Personalized Machine Learning

Julian McAuley

Contents

Illustrations

Tables

Notation

$u \in U$	users u in user set U
$i \in I$	items i in item set I
I_u	set of items rated by user u
U_i	set of users who have rated item i
$\|U\|$	number of users
$\|I\|$	number of items

INTRODUCTION TO PERSONALIZED MACHINE LEARNING

1
Introduction

Traditionally, *machine learning* encompasses a broad range of problems ranging from detecting objects in an image, to finding documents relevant to a given query, to predicting the next element in a sequence, among countless others. Roughly speaking, the field of machine learning approaches these problems by mining patterns in data in order to discover some underlying 'truth' about a document, image, or sequence.

Increasingly, though, there is a need to apply machine learning in settings where the 'correct' outcome is subjective, and depends on the context and characteristics of an individual user. As we browse online for movies to watch, products to buy, or romantic partners to connect with, we are likely engaging with a form *personalized* machine learning: that is the results are tailored to us specifically, based on the types of movies, products, or partners that *we specifically* are likely to engage with.

Much like traditional machine learning algorithms, *personalized* machine learning algorithms are at their heart essentially forms of pattern discovery. That is, predictions are made *for you* by analyzing the behavior of people *similar to you*. A feature such as 'people who liked this also liked' (**??**) is perhaps the most simple example of this type of personalized pattern discovery:[1] based on the contextual attribute of a user liking a particular item, recommendations are extracted based on users who share the same attribute. At the other end of the spectrum are complex deep learning approaches that learn 'black box' representations of users in order to make predictions, though these too at their heart rely on the intuition that 'similar' users (in terms of these complex representations) will have similar interaction patterns.

[1] Though strictly speaking maybe not one that we'd call 'machine learning.'

1.1 Purpose of This Book

In this book we seek to introduce *Personalized Machine Learning* by exploring the types of approaches used to solve the above problems, and construct a narrative around the underlying methods and design principles. As we explore in the next chapter, even in applications as diverse as song recommendation, or heart-rate profiling, there is a common set of techniques around which personalized machine learning systems are built.

By introducing this underlying set of principles, the book is intended is to teach readers how modern machine learning techniques can be improved by incorporating ideas from personalization and user modeling, and to guide readers in building machine learning systems where accurately modeling the users involved is key to success.

There is currently an abundance of models, datasets, and applications that seek to capture human dynamics or interactions. Examples pervade in diverse areas including web mining, recommender systems, fashion, dialog systems, personalized health, etc.

As such, there is an emerging set of common techniques that are used to capture the dynamics of 'users' in each of these settings. This book is designed to act as a reference point to explore and explain these techniques.

As a starting point, we will begin the book (Chapter 3) with a primer of machine learning, and especially supervised learning, that will bring readers up-to-speed on the basic techniques required later. Although this introductory material is likely familiar to many readers, we have a parcitular focus on user-oriented datasets, and show that even with 'standard' machine learning techniques, there is considerable scope for building personalized systems through careful feature engineering strategies that capture relevant user characteristics.

Following this, our main introduction point to personalized machine learning will be to explore personalized recommendation (Chapter 4). Recommendation technology has traditionally relied on personalization and user modeling, whether through simple similarity functions among users ('people like you also bought' etc.) or through more modern approaches, including approaches based on neural networks.

More recently, the need to account for personalization and to model users has encroached on a variety of other areas of machine learning. Exploring personalization and user modeling in these new areas—and more broadly giving readers the tools they need to design personalized approaches to them—is the main goal of this book.

Examples include:

Sequence Modeling and Time-Series Analysis

Natural Language Processing Modern NLP methods focus on interactive systems for language synthesis, dialog, question answering, etc. Within these applications there is a need to understand people's personalized writing styles, subjectivity, personal context (etc.). Understanding the variation among users is key to success in these applications.

Computer Vision A few 'traditional' computer vision problems depend on models of individual users (e.g. face recognition), though we are more interested in modern applications, such as personalized models of visual preferences, applications in fashion, and personalized GANs.

Fairness and Ethics Finally, there is growing recognition of the risks of blindly applying personalization and recommendation technologies without regard for the consequences of doing so. Examples include filter bubbles, 'concentration' effects (e.g. due to a lack of diversity of recommended content), or systematic biases against users from underrepresented groups. In response, there is a growing body of work that tries to understand and correct these types of bias

1.2 For Learners

Although this book is primarily intended as a guide to the specific topics of personalization, recommendation, and user modeling (etc.), it should also serve as a relatively gentle introduction to the topic of machine learning in general. Indeed, courses I've taught on recommender systems have routinely served as students' first exposure to the broad topic of machine learning. Such material is ideal as a starting point for learners seeking a more 'application oriented' view of machine learning compared to what is typically covered in introductory machine learning texts.

1.2.1 Expected Knowledge

While suitable as a first book on machine learning, the material in this book is intended for readers with some background in computer science, especially some familiarity with statistics, probability, linear algebra, and some experience working with structured datasets (e.g. *json*, *csv*).

- Programming in *Python*, which is used for all code examples.
- Basic string and data processing, file I/O etc.
- A basic knowledge of linear algebra, probability, etc.

For readers less familiar with the above, we give links on our online resources page (Section 1.4).

1.2.2 What is *Not* Covered

Regressors and classifiers When introducing basic machine learning concepts in Chapter 3, we limit ourselves to linear regression and linear classification (logistic regression), since these serve as building blocks for later methods we develop. Consequently, we ignore dozens of alternative regression and classification methods that are often the core of standard machine learning texts, though we discuss the merits of these alternatives in **??**

Dimensionality reduction We avoid a detailed treatment of dimensionality reduction and clustering algorithms, both staples of many introductory texts on machine learning. We note however that many of the techniques we explore when learning user representations are at their heart forms of dimensionality reduction. We choose to avoid a linear algebra-heavy presentation of 'traditional' dimensionality reduction techniques (principal component analysis, singular value decomposition, etc.), which in terms of actual implementation have little in common with the methods we develop.

Deep learning Although we cover various deep learning techniques (e.g. multilayer perceprtron-based recommendation in **??**, sequence models based on recurrent neural networks in Chapter 6, and models of visual preference based on convolutional neural networks in Chapter 8), by doing so we are merely scratching the surface of deep learning-based personalization. However, we note that deep learning approaches are merely one specific solution to the same problems that we visit using more traditional techniques. We survey deep learning-based personalization techniques in **??**.

Reinforcement and online learning We largely limit ourselves to traditional supervised learning problems, i.e., uncovering patterns and making predictions from historical collections of training data. In practice when deploying predictive models, data is obtained in a streaming setting and updates must be made in real time. This type of training regime is known as *online learning*, which we briefly cover in Section 4.11. We also avoid discussion of reinforcement learning algorithms, though touch upon their use for online advertising in **??**.

User interfaces and human computer interaction By design largely confined to machine learning approaches. That is, we are generally concerned

with building predictive systems that can estimate—as accurately as possible—how a particular user will respond to a given stimulus. By doing so, we can estimate preferences, predict future activities, retrieve relevant items, etc. This is complementary to a large body of work that explores personalization from the perspective of human computer interaction, where the primary concern is maximizing the quality of the user experience (ease of finding information, satisfaction, long-term engagement, etc.).

Deployment Likewise we largely avoid the 'systems building' side of personalized machine learning, such as concerns around deploying machine learning models on distributed servers (etc.), though we discuss high-level libraries and implementation best-practices throughout the book.

Of course, we are mindful of the dangers associated with 'black-box' approaches to machine learning, and want to avoid the pitfalls of blindly optimizing model accuracy, such filter bubbles, unwanted biases, or simply degraded user experience. In Chapter 9 we discuss these issues, as well as potential approaches to address them. However our discussion is still limited to machine learning solutions, i.e., we investigate *algorithmic* approaches to correct for biases, increase recommendation diversity, etc. We note that algorithmic solutions are only part of the picture, and that while having better algorithms is critical, it is also critical that those algorithms are appropriately *used*. Such questions fall under the umbrella of human computer interaction, and user interface design. Although these topics are outside the scope of this book, we discuss them briefly at the end of Chapter 9.

1.3 For Instructors

This book is inspired by my own experience teaching classes on recommender systems and web mining at UC San Diego. Courses on these topics have proved extremely popular, and are often chosen as learners' first machine learning courses.

One reason this topic acts as a good first contact with the machine learning curriculum is that it has a somewhat lower bar for entry than many machine learning courses, including (for example) courses on deep learning, or even many introductory machine learning classes. Partly this is due to the material being less dependent on deep and complex theory, and partly it is due to the ability to quickly build working solutions that are fairly representative of the

state-of-the-art, rather than mere proof of concepts. As such, a focus of this book is to quickly build working solutions, and covering a wide breadth of approaches, rather than diving too deep into the theory behind any one approach. I believe this will be useful in helping learners to understand the practical considerations behind building predictive systems based on user data, and will be complementary to the more theoretical treatment given in most introductory texts.

Another feature that has made this material popular among learners is the ability to work quickly with large, real-world datasets. The ability to work with collections of user data from *Amazon*, *Google*, *Steam* (etc.), on applications that are representative of real use-cases, has proved immensely valuable for students building their project portfolios or preparing for interviews. As such, each chapter is paired with project suggestions, each of which would be suitable as a major class project. These projects aim to synthesize the material from each chapter, with a particular focus on 'system building' considerations, design choices, and thorough model evaluation.

1.3.1 Course Plan and Overview

The content in this text is aimed at developing a quarter- or semester-long course, for students with some background in linear algebra, probability, and data processing. Chapters Chapter 3 and Chapter 4 cover core material upon which the remainder of the book builds. Chapters Chapter 6 to Chapter 8 are somewhat more orthogonal, such that components can be selected and combined as time or student background allows. A final chapter on bias, ethics, and fairness (Chapter 9) provides an opportunity to revisit earlier material through a new lens.

Each chapter is paired with homework and projects

Machine Learning Primer (Chapter 3) 2-3 weeks. Introduces the foundational concepts of machine learning, feature design, and evaluation, via a selection af datasets that capture user interactions. Exercises (Section 3.11) range from simple data manipulation to building a working machine learning pipeline (training, validation, etc.). Exercises in Chapter 3 are mainly concerned with feature design, including a project (Project 3.11.1) that involves experimenting with activity data involving temporal and geographical dynamics.

Recommender Systems (Chapter 4) 2-3 weeks. In Chapter 4 we introduce the core set of techniques used for recommendation. Traditional heuristics are presented along with machine learning approaches. Recommender systems are

used to develop the concept of a *user manifold* which is used throughout the following chapters to capture variation among users in various other settings. Exercises are mainly concerned with the basics of building practical recommendation approaches, and a project (Project 4.16.1) is concerned with building an end-to-end recommendation pipeline for a book recommendation scenario.

Content and Structure in Recommender Systems (Chapter 5) 1 week. Explores how to incorporate features (i.e., side information) into personalization (mostly recommendation) approaches, and explores personalization in settings with additional structure, such as socially-aware recommendation, news recommendation, etc. Some of these content-aware approaches (such as factorization machines) are revisited later the book when developing more complex models based on (e.g.) temporal or sequential dynamics. A project (Project **??**) consists of developing recommender systems for use in cold-start settings (i.e., capable of adapting to new users or new items).

Temporal and Sequential Models (Chapter 6) 1-2 weeks. Starts by revising some of the basic approaches to temporal and sequential modeling, such as autoregression and Markov chains; later explores how to personalize such models, and explores complex approaches based on recurrent networks. A project (Project **??**) compares various approaches to temporal recommendation.

Personalized Models of Text (Chapter 7) 1 week. After revising some of the basic predictive models of text (such as bag-of-words representations), we explore how text can be used to understand the dimensions of preferences. We revisit sequential modeling by exploring techniques that borrow from natural language to model interaction sequences. We also visit methods for text *generation*, which can be personalized in settings ranging from conversation to justification of machine predictions. A project (Project **??**) consists of building personalized systems for document retrieval.

Personalized Models of Visual Data (Chapter 8) 1 week. Explores applications involving visual data, ranging from personalized image search, to applications in fashion and design. Exercises and a project (Project 8.7) consist of building visually-aware recommendation systems for applications in fashion.

The Ethics of Personalized Machine Learning (Chapter 9) 1-2 weeks. The final chapter looks at the ethics and the various pitfalls of developing personalized machine learning systems. Examples include filter bubbles, extremification, and issues of bias and fairness. The chapter has a significant focus on

applied case-studies, and allows us to revisit several of the topics from previous chapters through a new lens. Exercises and a project (Section 9.10) consist of improving recommendation approaches in terms of gender parity and other fairness objectives.

1.4 Online Resources

To help readers with exercises, projects, and to collect resources such as datasets and additional reading materials, an online supplement is available to augment the material covered here with working code and examples:

`https://cseweb.ucsd.edu/~jmcauley/pml/`

The online supplement includes:

- Complete code examples covering the material in each chapter. These cover complete worked examples from which the code samples presented in each chapter are drawn. Additional code samples are included that reproduce various figures and examples presented throughout the book.
- Links to all datasets used in the book (as well as various other personalization datasets).
- Links to additional reading, mostly focused on introductory material useful to learners less familiar with some of the background material described in Section 1.2.1.

1.5 About the Author

Julian McAuley has been an Assistant Professor at UC San Diego since 2014, and an Associate Professor since 2019. Previously he was a postdoctoral scholar at Stanford University, after completing my undergraduate and graduate training at the University of New South Wales and the Australian National University.

Personalized Machine Learning is the main research theme of my lab at UCSD. Our research has pioneered the use of images and text in recommendation settings (e.g. McAuley et al. (2015a); McAuley and Leskovec (2013a)), with applications including fashion design, personalized ques-

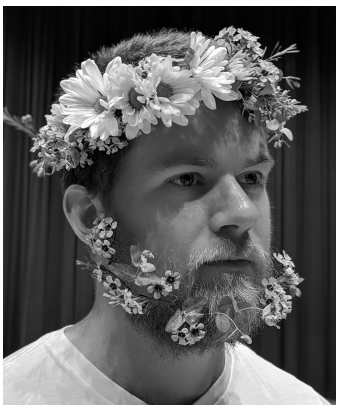


Figure 1.1: The Author

tion answering systems, and interactive dialog systems. Our lab has also studied personalization outside of typical recommendation settings, such as developing personalized models of heart-rate profiles Ni et al. (2019b), and systems for generating personalized recipes Majumder et al. (2019).

2

Personalization in Everyday Life

Other than introducing the techniques that underlie personalized machine learning systems, one of our goals in this book is to explore the wide range of practical applications where personalization is applied, to explore the history of the topic, and eventually to explore the broad consequences and ethical concerns associated with personalized machine learning.

2.1 The Importance of Personalization

2.1.1 Recommendation

Many of the examples we cover in this book will relate to *recommender systems*, and more broadly modeling user interactions on the web. Part of the reason for this focus is opportunistic: user interaction data is widely available, allowing us to build models on top of real data throughout the book.

Pedagogically, recommender systems are also appealing as an introduction to personalized machine learning as they allow us to quickly implement working systems that are close to the state-of-the-art. As we'll see, even widely-deployed systems turn out to be surprisingly simple, relying on simple heuristics and building on standard data structures (Section 4.4).

Ultimately though, our main reason for studying recommender systems is because they are a fundamental tool for modeling *interactions between users and items*. As such, the basic techniques developed when building recommender systems can be applied in a variety of situations where we want to predict how a user will respond to some stimulus. We explore a variety of related settings below.

2.1.2 Social and News-Media

2.2 Personalization Beyond Recommendation

Fundamentally, the value of *personalized machine learning* is to model datasets where *variation among individuals* captures a large fraction of the variability in a dataset. A high degree of variability among individuals manifests itself in a wide variety of settings. In recommender systems, users might vary due to subjective preferences, budgets, or demographic factors; in settings like personalized health, users may vary in terms of their physical characteristics, medical histories, or risk factors; or in settings involving natural language (or dialog) users may vary in terms of their writing styles, personalities, or their specific context.

Below we describe a few such examples, partly to highlight the wide range of settings where personalization is critical, but also to demonstrate the common set of ideas involved in modeling them.

2.2.1 Personalized Health

Personalized health

Note that most of these tasks are ones for which *non*-personalized models would not be effective

Estimating what symptoms a patient will exhibit on their next hospital visit is a canonical task in personalized health, with applications in (e.g.) preventative treatment. This task closely resembles the settings we explore when developing recommender systems, given the goal to estimate patients' interactions with certain stimuli (symptoms) over time Yang et al. (2014). As such, techniques for such tasks borrow ideas from recommender systems, especially temporal and sequential recommendation, as we develop in Chapter 6.

Beyond estimating patient symptoms, personalized machine learning techniques can be adapted to related tasks such as estimating the duration of surgical procedures (in order to more efficiently utilize hospital resources). Accurate prediction for such a task depends on both patient and surgeon characteristics. Successful models for this task have been built using traditional regression techniques, along with carefully-engineered patient and surgeon features Ng et al. (2017), resembling the types of personalization we explore in 'standard' regression and classification settings in Chapter 3.

Beyond regression, classification, and ranking tasks, a variety of problems in personalized health depend on more complex predictions, e.g. forecasting sequential data. For example, techniques to model the progression of heart-rate sequences in response to physical stimuli Ni et al. (2019b), or to estimate

the distribution kinetics of drugs (such as anesthetics) Ingrande et al. (2020), requires complex sequential models. Modern approaches to these types of sequential modeling tasks are often based on recurrent neural networks, which are capable of learning complex semantics from sequential data; we explore these types of personalized sequential models in **??**.

Many problems in personalized health also depend upon natural language data, for example modeling the characteristics of clinical notes, which again exhibit substantial variation as a function of the patient or physician. More complex task may even consist of generating notes automatically based on (e.g.) a radiology image Ni et al. (2020). Such applications build on techniques for personalized natural language processing and generation, as we develop in Chapter 7.

2.2.2 Computational Social Science

Often the goal of modeling user data is not merely to *predict* future events or interactions, but to *understand* the underlying dynamics at play. As such

Price sensitivity
Spotify maybe

2.2.3 Language Generation, Personalized Dialog, and Interactive Agents

Finally, given the new modalities via which people interact with predictive systems, there are new demands for personalization.

For example, personalization is critical in a broad range of settings involving natural language. User-generated language data exhibits substantial variability due to differences in writing style, subjectivity, etc. When dealing with such data, non-personalized models may struggle with this nuance.

For example, automated systems for dialog, whether in task-oriented settings or for open-domain 'chit-chat' can benefit from personalization, in order to generate responses that are more personalized or empathetic to tone or context of individual users Majumder et al. (2020).

2.3 The Risks and Ethics of Personalization

Along with the increasing ubiquity of personalized machine learning systems, there is a growing awareness of the risks associated with personalization. Some

of these issues have reached mainstream awareness, such as the idea that personalized recommendations can trap users in 'filter bubbles,' while other issues are considerably more subtle.

For instance, considering the specific case of recommender systems, a naïvely-implemented model can introduce issues including:

Filter bubbles Roughly speaking, recommendation algorithms rely on identifying specific item characteristics that are preferred by each user, and recommending items that most closely represent those characteristics. Without care, even a user with broad interests may be recommended only a narrow set of items that closely mimic their prior interactions.

Extremification Likewise, a system that identifies features that a user is interested in may identify items that are most representative of those features, e.g. a user who likes action movies may be recommended movies with *a lot* of action; in contexts such as social media and news recommendation this can lead to users being exposed to increasingly extreme content (the relationship between this and the previous issue is explained in Chapter 9).

Concentration Similar to the previous phenomenon, a user who has diverse interests may receive recommendations that only follow their most predominant interest (Section 9.2.1). In aggregate, this may lead to to a small set of items being over-represented among users' recommendations.

Bias Given that recommenders (and many other personalized models) ultimately work by identifying common patterns of user behavior, users in the 'long-tail' whose preferences don't follow the predominant trends may receive sub-par recommendations.

Along with a rising awareness of these issues has come a set of techniques designed to mitigate them. These techniques borrow ideas from the broader field of fair and unbiased machine learning, whereby learning algorithms are adapted so as not to propagate (or not to exacerbate) biases in training data, though the fairness goals are often quite different. Diversification techniques can be used to ensure that predictions or recommendations balance relevance with novelty, diversity, or serendipity; related techniques seek to better 'calibrate' personalized machine learning systems by ensuring that predicted outputs are balanced in terms of categories, features, or the distribution over recommended items (Section 9.3). Such techniques can mitigate filter bubbles by ensuring that model outputs aren't highly concentrated around a few items, and more qualitatively can increase the overall novelty or 'interestingness' of model outputs. Other techniques follow more directly from fair and unbiased

machine learning, ensuring that the performance of personalized models is not degraded for users belonging to underrepresented groups, or who have niche preferences (Section 9.6).

2.4 Techniques for Personalization

As mentioned in Chapter 1, one of the goals of this book is to establish a common narrative around the tools and techniques used to design personalied machine learning systems. Although we've shown that such systems are applied in domains as diverse as e-Commerce to personalized health, we find that the techniques used to implement these models follow a few common paradigms.

2.4.1 User Representations as Manifolds

One of the main ideas we'll revisit throughout this book—and which allows us to adapt ideas from recommender systems to other types of machine learning—is that of a *user manifold*. That is, most of the personalized methods we'll explore will involve *representations* of users that describe the common patterns of variation in their activities and interactions.

In the case of recommender systems, this 'user manifold' will be a vector that describes the principal dimensions that explain variance among user preferences (Figure 2.1). For example, we might discover that the principal dimensions that explain variance in preferences in a movie recommendation setting center around certain genres, actors, or special effects. In a text generation setting (**??**) user manifolds might describe patterns of variation in users' writing styles; when estimating users' heart-rate profiles (**??**), user manifolds might describe users' fitness levels or other patterns of variation in heart-rate dynamics; or in a personalized fashion setting (**??**), user manifolds might describe users' preferences toward specific visual characteristics.

Throughout the book, we'll revisit the idea of user manifolds, as a general-purpose means of capturing common patterns of variation among users. Some examples include:

- In Chapter 4, we'll use low-dimensional user representations to describe the dimensions of preferences and activities, which can be used to recommend items that users are likely to interact with.
- In Chapter 7, user representations can describe the topics users tend to discuss (e.g. when writing reviews), or individual characteristics of their writing styles.

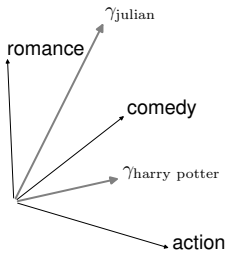


Figure 2.1 The basic idea behind recommender systems, and various other types of personalized machine learning, is to represent users by *low-dimensional manifolds* that describe the patterns of variance among their interactions.

- In Chapter 8, user representations will describe the visual dimensions that users are interested in, allowing us to rank, recommend, or generate images in a personalized way.
- Throughout various case studies, user representations will capture characteristics ranging from dietary preferences (Section 7.7), fitness profiles (Section 6.8), social trust (**??**), or fashion choices (Section 8.3).

2.4.2 Contextual Personalization and Model-Based Personalization

Although this book will predominantly cover methods that explicitly model user terms (as above), we will also cover a variety of models that deliberately avoid doing so.

Starting with trivial models such as 'people who bought X also bought Y,' many classical approaches for (e.g.) recommendation *leverage user data*, but do not include explicit parameters (i.e., a 'model') associated with a user. However, such models are still *personalized*, in the sense that different predictions will be made for each individual based on how they interact with the system. Simple machine learning techniques, such as those we develop in Chapter 3, where users are represented by a few carefully-engineered features, also follow this paradigm.

We'll distinguish between these two classes of approach using the terms *model-based* and *contextual* personalization. *Model-based* approaches learn an explicit set of parameters associated with each user, such as the 'user manifolds' described above; these models are typically intended to capture the predominant patterns of variation among users in a system, usually in terms

of a low-dimensional vector. In contrast, *contextual* (also sometimes called 'memory-based,' as in Chapter 4) approaches extract features from users' histories of recent interactions.

There are several settings in which contextual personalization may be preferable to explicitly modeling a user. When developing simple recommender systems in Chapter 4, and even more trivial personalized models in Chapter 3, we see that personalization can often be achieved with simple heuristics, or hand-crafted features or similarity measures. Such approaches may be desirable for a number of reasons: simple models may be more interpretable (and therefore preferable to expose to a user compared to 'black-box' predictions); or, we may lack adequate training data to learn complex representations from scratch.

FUNDAMENTALS OF PERSONALIZED MACHINE LEARNING

3

Machine Learning Primer

In this chapter, we'll cover the fundamental principles of machine learning (and in particular *supervised* learning), that will serve as a foundation for the remaining material in this book.

Although we'll only briefly touch upon *personalization* in this chapter, our examples will focus on the same types of user-oriented data that we'll visit in later chapters. In particular, we'll focus on datasets covering topics such as recommendation, sentiment, and predictive tasks involving (e.g.) demographic characteristics.

As such, the view we'll take on 'personalization' in this chapter will consist of *extracting features about individuals* in order to make predictions using traditional machine learning frameworks. Later, we'll draw a distinction between this type of method—where we extract *features* about users—and methods where we explicitly *model* each user. This will drive our discussion of *contextual* versus *model-based* personalization, though we'll discuss this distinction more precisely in Section 4.5. However, as we'll see in this chapter (and in various examples throughout the book), even traditional machine learning techniques, paired with appropriate feature extraction strategies, can lead to surprisingly effective models for personalized prediction.

Supervised Learning Most of the personalization techniques we'll explore throughout this book are forms of *supervised learning*. Supervised learning techniques assume that our prediction tasks (or our datasets) can be separated into two components:

labels (denoted y) that we would like to predict, and
features (denoted x) which we believe will help us to predict those labels

For example, given a sentiment analysistask (Chapter 7), our data might be

Supervised Learning approaches are those that seek to directly learn the relationship between the observed data X and the labels y. Most of the models in this book are forms of supervised learning, starting with regression and classification in this chapter, and continuing into later chapters as we build models to predict user activities. In contrast, *unsupervised learning* approaches seek to find patterns in the data X, but are not specifically concerned with predicting any label; examples include techniques for clustering and dimensionality reduction. Finally, *semi-supervised learning* approaches are somewhere in between, usually leveraging large datasets of *unlabeled* data to improve the performance of supervised models with a small number of labels.

Figure 3.1 Supervised, Semi-supervised, and Unsupervised Learning

(the text of) reviews from *Amazon*or *Yelp* and our labels would be the ratings associated with those reviews.

Given this distinction between features and labels in a dataset, the goal of a *supervised learning* algorithm is to infer the underlying function $f(x) \rightarrow y$ that explains the relationship between the features and the labels. Usually, this function will be *parameterized* by model parameters θ, i.e.,

$$f_\theta(x) \rightarrow y. \tag{3.1}$$

For example, in this chapter θ might describe which features are positively or negatively correlated (or uncorrelated) with the labels, or later θ might capture the preferences of a particular user in a recommender system (Chapter 4). Figure 3.1 explains how this type of approach relates to other types of learning.

Throughout this chapter, we will assume that we are given *labels* in the form of a vector y, and features in the form of a matrix X, so that each y_i is the label associated with the i^{th} datapoint, and x_i is a vector of features associated with that datapoint.

3.1 Linear Regression

Perhaps the simplest association we could assume between our features X and our labels y would be a *linear*relationship, i.e., that the relationship between X and y was defined as

$$y = X\theta, \tag{3.2}$$

or in our notation from Equation (3.1)

$$f_\theta(X) = X\theta. \tag{3.3}$$

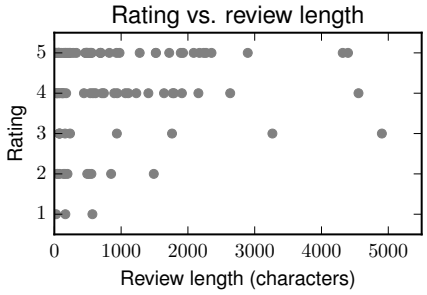


Figure 3.2: Ratings compared to review length (in characters), based on 100 reviews of fantasy novels from *Goodreads*

In the above, X is a matrix describing features extracted from the data, y is our vector of labels, and θ is a vector of unknowns which describe which features are *relevant* to predicting the labels.

Ignoring strict notation for now, a trivial example might consist of predicting a review's rating as a function of its length. To do so, let's consider a small dataset of 100 (length, rating) pairs from *Goodreads* Fantasy novels. Figure 3.2 plots the relationship between review length (in characters) and the rating.

From Figure 3.2, there appears to be an association between ratings and review length, i.e., more positive reviews tend to be longer. A very simple model might attempt to describe that relationship with a line, i.e.,

$$\text{rating} \simeq \theta_0 + \theta_1 \times (\text{review length}). \tag{3.4}$$

Note that the above is just the standard equation for a line ($y = mx + b$), where θ_1 is a slope and θ_0 is an intercept.

If we can identify a line that approximately describes this relationship, we can use it to estimate a rating from a given review, even though we may never have seen a review of some specific length before. In this sense, the line is a simple *model* of the data, as it allows us to predict labels from previously unseen features. To do so we formalize the problem of finding a line of best fit.

Specifically, we are interested in identifying the values of θ_0 and θ_1 that most closely matches the trend in Figure 3.2. To solve for $\theta = [\theta_0, \theta_1]$, we can write out the problem as a system of equations in matrix form:

$$y = X \cdot \theta, \tag{3.5}$$

where y is our vector of observed ratings and X is our matrix of observed features (in this case the reviews lengths). For the first few samples of our

Figure 3.3 Why is there a column of '1's in the feature matrix?
The first column of the feature matrix X in Equation (3.6), and in most feature matrices throughout this chapter, is a column of ones. To explain why we always have this feature, it is useful to expand the inner product $[1, \text{length}] \cdot [\theta_0, \theta_1]$ (e.g. as in Equation (3.6)) to confirm that it expands to the equation for a line $\theta_0 + \theta_1 \times \text{length}$. Without the constant term in our feature matrix, we would be implicitly assuming that the fitted line passes through $(0, 0)$.

Goodreads data we have:

$$\underbrace{\begin{bmatrix} 5 \\ 5 \\ 5 \\ 4 \\ 3 \\ 5 \\ \vdots \end{bmatrix}}_{y} \simeq \underbrace{\begin{bmatrix} 1 & 2086 \\ 1 & 1521 \\ 1 & 1519 \\ 1 & 1791 \\ 1 & 1762 \\ 1 & 470 \\ & \vdots \end{bmatrix}}_{X} \cdot \underbrace{\begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}}_{\theta}. \tag{3.6}$$

It is useful to compare Equations (3.4) and (3.6) to understand how the matrix expression above expands to include the slope ($\theta_1 \times$ (review length)) and intercept (θ_0) terms.

We would like to solve Equation (3.6) for θ. Naïvely, we might attempt to multiply both sides of the equation $y = \theta \cdot X$ by X^{-1}; however the inverse is not well-defined, since X is not a square matrix.

To obtain a square matrix, we (left) multiply both sides by X^T:

$$X^T y \simeq X^T X\theta, \tag{3.7}$$

resulting in a square (in this case 2×2) matrix $X^T X$. We can now multiply both sides by the inverse of this matrix:

$$(X^T X)^{-1} X^T y \simeq (X^T X)^{-1}(X^T X)\theta, \quad \text{or simply} \quad \theta = (X^T X)^{-1} X^T y. \tag{3.8}$$

The quantity $(X^T X)^{-1} X^T$ is known as the *pseudoinverse* of X.

Computing $\theta = (X^T X)^{-1} X^T y$ for our 100 ratings from Goodreads yields

$$\theta = \begin{bmatrix} 3.984 \\ 1.194 \times 10^{-4} \end{bmatrix}, \tag{3.9}$$

corresponding to the line

$$\text{rating} = 3.984 + 1.194 \times 10^{-4}(\text{review length}). \tag{3.10}$$

This line reflects a positive (albeit slight) trend between review length and

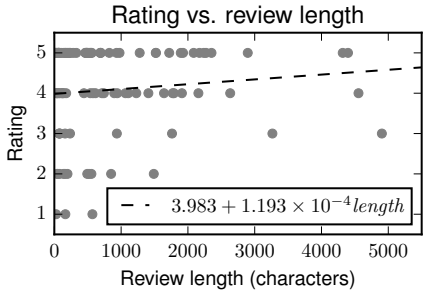


Figure 3.4: Line of best fit between ratings and review length.

ratings: for every additional character in a review, our estimate of the rating increases very slightly (by 1.194×10^{-4} points). This line of best fit is depicted in Figure 3.4. We will revisit how to interpret the parameters of linear models in Section 3.4.

More Complex Models The above reasoning generalizes to fitting more complex models than a simple line, for example we could imagine that a rating could be related to both the length of the review and the number of comments the review received:

$$\text{rating} \simeq \theta_0 + \theta_1 \times (\text{review length}) + \theta_2 \times (\text{n\_comments}). \tag{3.11}$$

The above process—finding a line of best fit that best approximates the relationship between our observed features X and labels y—describes the basic concept of *linear regression*.

Adding More Dimensions Just as Equation (3.4) corresponded to fitting a line in two dimensions, Equation (3.11) now corresponds to fitting a plane in three. But ultimately the procedure for fitting this model remains the same. We simply have an additional column in our feature matrix:

$$X = \begin{bmatrix} 1 & 2086 & 1 \\ 1 & 1521 & 1 \\ 1 & 1519 & 5 \\ 1 & 1791 & 1 \\ 1 & 1762 & 0 \\ & \vdots & \end{bmatrix}. \tag{3.12}$$

Solving $\theta = (X^T X)^{-1} X^T y$ yields

$$\theta = \begin{bmatrix} 3.954 \\ 7.243 \times 10^{-5} \\ 0.108 \end{bmatrix} \begin{matrix} \text{intercept} \\ \text{slope for length} \\ \text{slope for number of comments} \end{matrix}. \tag{3.13}$$

Interestingly, when we add this additional parameter θ_2, the values of θ_1 and θ_0 are different from those of the model we previously fit (compare Equations (3.9) and (3.13)). Critically, the slope associated with the length term (θ_1) is reduced in our new model. We discuss how to interpret these parameters in Section 3.4.

3.1.1 Regression in *sklearn*

Various libraries support the basic machine learning techniques described in this chapter, and indeed they can be implemented relatively straightforwardly via standard linear algebra operations. Here we describe the implementation in *scikit-learn* though other implementations follow similar interfaces.

First we load our dataset; here we read our sample (in this case a toy dataset of 100 reviews) in *json* format,[1] which results in a list of 100 dictionaries:

```
1 data = []
2 for l in open("fantasy_100.json"): # 100 reviews of fantasy novels
      from Goodreads
3     d = json.loads(l)
4     data.append(d)
```

Next we extract labels and features from the dataset. In this case we train a predictor to estimate ratings as a function of review length, as in **??**:

```
1 ratings = [d['rating'] for d in data] # The output we want to
      predict
2 lengths = [len(d['review_text']) for d in data] # The feature we'
      ll use for prediction
```

To regress on this data we must first construct our matrix of features X and our vector of labels y; note the inclusion of a constant feature in our feature matrix:[2]

```
1 X = numpy.matrix([[1,l] for l in lengths])
2 y = numpy.matrix(ratings).T
```

From here regressing is simply a matter of passing our features and labels to the appropriate model from *sklearn*. Having done so we extract the coefficients θ:

```
1 model = sklearn.linear_model.LinearRegression(fit_intercept=False)
2 model.fit(X, y)
3 theta = model.coef_
```

[1] *Json* is a structured data format, made up of key-value pairs (where values can in turn be lists or other json objects). See `https://www.json.org/`.

[2] Although in fact this can be excluded and θ_0 fit by the library by setting `fit_intercept=True`; here we include it manually as a matter of best-practice.

Finally, we can confirm manually that the pseudoinverse from Equation (3.8) yields the same result:

```
1 numpy.linalg.inv(X.T*X)*X.T*y
```

(in both cases we find that θ =?).

3.2 Evaluating Regression Models

When developing the linear models above, we were somewhat imprecise about what is meant by a 'line of best fit' (or generally a model of best fit). Indeed, the pseudoinverse is not a 'solution' to the system of equations given in Equation (3.6), but is merely an approximation (naturally, the line of best fit does not pass through all points exactly).

Here, we would like to be more precise about what it means for a model to be 'good'. This is a key issue when fitting and evaluating any machine learning model. One needs a way of quantifying how closely a model fits the given data. Given a desired measure of success, we can compare alternative models against this measure, and design optimization schemes that optimize the desired measure directly.

3.2.1 The Mean Squared Error

A commonly used evaluation criterion when evaluating regression algorithms is called the Mean Squared Error, or MSE. The Mean Squared Error between a model $f_\theta(X)$ and a set of labels y is defined as

$$\text{MSE}(y, f_\theta(X)) = \frac{1}{|y|} \sum_{i=1}^{|y|} (f_\theta(x_i) - y_i)^2, \tag{3.14}$$

in other words, the average squared difference between the model's predictions and the labels. Often reported is also the *Root* Mean Squared Error (RMSE), i.e., $\sqrt{\text{MSE}(y, f_\theta(X))}$; the RMSE is sometimes preferable as it is consistent in scale with the original labels.

With some effort, it can be shown that the linear model $f_\theta(X)$ that minimizes the MSE compared to the labels y is given by using the pseudoinverse as in Equation (3.8). We leave this as an exercise (**??**).

3.2.2 Why the Mean Squared Error?

Although the Mean Squared Error has a convenient relationship with the pseudoinverse, it may otherwise seem a somewhat arbitrary choice of error mea-

sure. For instance, it may seem more obvious at first to compute an error measure such as the Mean Absolute Error (or MAE):

$$\text{MAE}(y, f_\theta(X)) = \frac{1}{|y|}\sum_{i=1}^{|y|} |f_\theta(x_i) - y_i| \tag{3.15}$$

Or, why not count the number of times the model is wrong by more than one star? Or why not measure the mean *cubed* error?

To defend the mean squared error as a reasonable choice, we need to characterize what types of errors are more 'likely' than others. Essentially, the Mean Squared Error is assigning small penalties to small errors, and *very* large penalties to large errors. This is in contrast to, say, the Mean Absolute Error, which assigns penalties exactly proportional to how large the error is. What the Mean Squared Error therefore seems to be assuming is that small errors are *common* and large errors are very uncommon.

What we are talking about informally above is a notion of how errors are *distributed* under some model. Formally, we say that the labels are equal to our model's predictions, plus some error:

$$y = \underbrace{f_\theta(X)}_{\text{prediction}} + \underbrace{\epsilon}_{\text{error}}, \tag{3.16}$$

and that our error follows some probability distribution. Our argument above said that small errors are common and large errors are very rare. This suggests that errors may be distributed following a *bell curve*, which we could capture with a Gaussian (or 'Normal') distribution:

$$\epsilon \sim \mathcal{N}(0, \sigma^2). \tag{3.17}$$

The density function for a (zero mean) Gaussian distribution is given by

$$f'(x') = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x'}{\sigma}\right)^2} \tag{3.18}$$

(we use the notation f' and x' to avoid confusion with f and x elsewhere). So, the probability density of an error as large as $y_i - f_\theta(x)$ is given by

$$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y-f_\theta(x)}{\sigma}\right)^2}. \tag{3.19}$$

This density function is depicted in Figure 3.5.

3.2.3 Maximum Likelihood Estimation of Model Parameters

Having defined the density function above, we can now reason more formally about about what it means for a particular model to be a 'good' fit to the data.

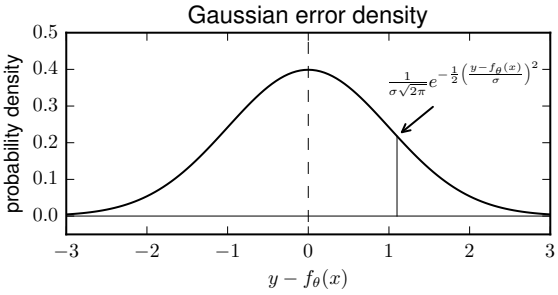


Figure 3.5: Gaussian error density.

In other words, we would like to ask how 'likely' a particular model is in terms of a given error distribution.

Specifically, the density function in Equation (3.19) gives us a means of assigning a probability (or likelihood) to a particular set of labels y, given features X and a model θ, under some particular error distribution (in this case a Gaussian):

$$\mathcal{L}_\theta(X|y) = \prod_{i=1}^{|y|} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i - f_\theta(x_i)}{\sigma}\right)^2}. \tag{3.20}$$

Essentially, we want to choose θ so as to maximize this likelihood. Our goal intuitively is to choose a value of θ that is consistent with this error distribution, i.e., a model that makes many small errors and few large ones.

Precisely, we would like to find $\arg\max_\theta \mathcal{L}_\theta(X|y)$. This procedure (finding a model θ that maximizes the likelihood under some error distribution is known as *Maximum Likelihood Estimation*. We solve by taking logarithms and removing irrelevant terms (π, σ):

$$\begin{aligned}
\arg\max_\theta \mathcal{L}_\theta(X|y) &= \arg\max_\theta \ell_\theta(X|y) && (3.21)\\
&= \arg\max_\theta \log \prod_{i=1}^{|y|} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i - f_\theta(x_i)}{\sigma}\right)^2} && (3.22)\\
&= \arg\max_\theta \sum_i \log e^{-\frac{1}{2}\left(\frac{y_i - f_\theta(x_i)}{\sigma}\right)^2} && (3.23)\\
&= \arg\max_\theta -\sum_i (y_i - f_\theta(x_i))^2 && (3.24)\\
&= \arg\min_\theta \sum_i (y_i - f_\theta(x_i))^2 && (3.25)\\
&= \arg\min_\theta \frac{1}{|y|} \sum_i (y_i - f_\theta(x_i))^2 && (3.26)
\end{aligned}$$

Figure 3.6 The MSE and the MLE
The argument we made in Section 3.2.2 explained our motivation behind the choice of the Mean Squared Error (MSE): by choosing the MSE as our error metric, we are implicitly assuming that our model's errors follow a Gaussian distribution. This assumption is explained by the fact that minimizing the mean squared error maximizes the likelihood of the observed errors under a Gaussian error model.

Note crucially in the above equation that *the maximum likelihood solution for θ under our Gaussian error model is precisely the Mean Squared Error*. This demonstrates the relationship between the Mean Squared Error and Maximum Likelihood Estimation (which we summarize in Box 3.6).

The above arguments may seem like just a mathematical curiosity, and indeed in practice we will often minimize the Mean Squared Error without scrutinizing the decision to do so. But this relationship between error functions and probabilities will come up regularly when we develop models for classification (Section 3.8), sequence mining (Chapter 6), and recommender systems (Chapter 4). To summarize a few key points:

(i) When we optimize a certain error criterion, we are often making implicit assumptions about how errors are distributed.
(ii) Sometimes, a model will poorly fit a dataset because these assumptions are violated. Understanding the underlying assumptions gives us a chance to diagnose them and attempt to correct it (Section 3.2.5).
(iii) In many of the models we fit later (including when we develop classifiers in **??**), we will use this style of probabilistic language, i.e., we will talk about some observed data having high *likelihood* under some model. Fitting such models will use this same strategy of selecting a model which maximizes this likelihood.

3.2.4 The R^2 Coefficient

Having motivated our choice of the Mean Squared Error at some length, it is worth asking how low the MSE should be before we consider our model to be 'good enough'?

This quantity turns out not to be well-defined: the Mean Squared Error will depend on the scale and variability of our data, and the difficulty of our task. For example, predicted ratings on a 5-point scale would likely have lower MSEs than predicted ratings on a 100-point scale; on the other hand, this might not be the case if ratings on a 100-point scale were highly concentrated (e.g. nearly all ratings were in the 92-95 range). Finally, the MSE in either set-

ting could be higher simply due to a lack of available features that allow us to predict ratings accurately.

As such, we would like a calibrated measurement of model error. As we just argued, the MSE is related to the *variance* of the data: this relationship is easy to see as follows:

$$\bar{y} = \frac{1}{|y|} \sum_i y_i \tag{3.27}$$

$$\text{var}(y) = \frac{1}{|y|} \sum_i (y_i - \bar{y})^2 \tag{3.28}$$

$$\text{MSE}(y|X) = \frac{1}{N} \sum_i (y_i - f(x_i))^2 \tag{3.29}$$

In other words, the Mean Squared Error would be equal to the variance if we had a trivial predictor that always estimated $f(x_i) = \bar{y}$.[3] Thus the variance might be used as a way of normalizing the Mean Squared Error:

$$\text{FVU}(y|X) = \frac{\text{MSE}(y|X)}{\text{var}(y)} \tag{3.30}$$

This quantity, known as the *Fraction of Variance Unexplained* essentially measures the extent to which the model *explains variability* in the data, as compared to a predictor which always predicts the mean (i.e., one which explains no variability at all).

This quantity will now take a value between 0 and 1: 0 being a perfect classifier (MSE of zero), and 1 being a trivial classifier.[4]

Often, one reports the R^2 *coefficient*, which is simply one minus the FVU:

$$R^2 = 1 - \frac{\text{MSE}(y|X)}{\text{var}(y)}, \tag{3.31}$$

which now takes a value of 1 for a perfect predictor, and 0 for a trivial predictor. The name 'R^2' comes from a different way of deriving the same quantity, in terms of the correlation between the predictions and the labels.[5]

3.2.5 What to do if Errors *Aren't* Normally Distributed?

The arguments we made above characterized the relationship between the Mean Squared Error and the normal (Gaussian) distribution. In summary, the MSE

[3] Note that this is the best we could do if using a trivial predictor of the form $f(x_i) = \theta_0$ (**??**).

[4] The FVU *could* be greater than 1, if our classifier were *worse* than a trivial one.

[5] We omit this alternate derivation for now, but revisit the idea of correlation briefly in Section 4.3.4.

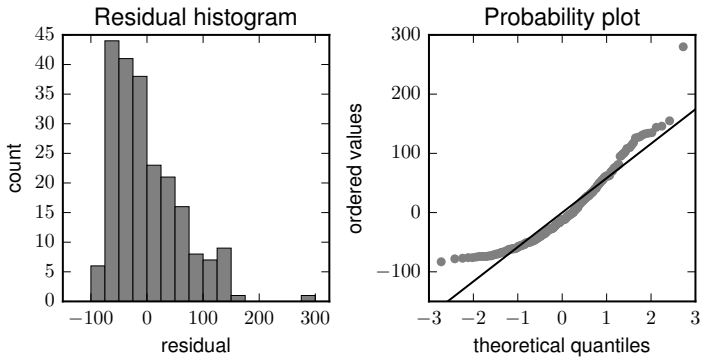


Figure 3.7 Histogram of observed residuals (left), and residuals compared to theoretical quantiles under a normal distribution.

is a reasonable choice so long as our model errors are expected to be centered around zero, and not to have large outliers.

But what can we do if these assumptions do not hold? First, we consider how to validate the assumptions in the first place. Recall that our assumptions assume that the residuals,

$$r_i = (y_i - f_\theta(x_i)) \tag{3.32}$$

follow a normal distribution. To begin with, a simple plot may reveal whether the residuals follow the desired overall trend.

Figure 3.7 (left) shows a histogram of residuals r_i for a simple prediction task, in which we estimate review lengths as a function of user gender (covered later in Section 3.3.2). Although the plot has a slight bell shape, it deviates from the normal distribution in several key ways, for instance:

- The residuals do not appear to be centered around zero. In fact the *average* residual is zero,[6] though largest bins in the histogram are somewhat below zero.
- The are some large outliers (i.e., extremely long reviews whose length was underpredicted).
- There are no small outliers, and there is almost no 'left tail,' i.e., the model never significantly overpredicts.

Although the histogram in Figure 3.7 allows us to quickly assess whether

[6] In fact, the average residual of this type of linear regression model is *always* zero (see **??**).

the residuals follow a normal distribution, this can be visualized more precisely by comparing the theoretical quantiles of a normal distribution to the observed residuals, as in Figure 3.7, right.[7] The plot essentially compares the (sorted) residuals to those we would expect if we were to sample the same number of values from a normal distribution: if our residuals followed a normal distribution, plotting these quantities against each other would result in a straight line. Again, the plot basically reveals that there is an unusual outlier, and that residuals are missing the left tail (i.e., overpredictions) that would be expected. Note that this same type of diagnostic tool can be used to compare our residuals against any distribution in the same way.

While the above is merely a diagnostic for determining *whether* the residuals followed a normal distribution, the more difficult question is how these discrepancies can be corrected. Some general guidelines are as follows:

Remove outliers The normal distribution (and thus the MSE) is especially sensitive to outliers due to how it penalizes large errors. To the extent that extremely long reviews do not conform to the usual behavior of the data, we could simply discard them before training.

Choose an error distribution less sensitive to outliers For example, the Mean Absolute Error assigns a smaller penalty to large mispredictions, so outliers will have a smaller effect on the model.

Choose a skewed distribution In this example we are predicting lengths, which by definition are bounded below (at length zero) but not above. Thus there will be a long tail of underpredctions, but not large overpredictions. We might account for this by modeling the data using a skewed probability distribution (such as a Gamma distribution.

Fit a better model Note that the diagnostic in Figure 3.7 is a function of the *errors*, rather than the original data. Thus, for example, if we had a feature that allowed us to correctly predict the length of the unusually long review, the errors may again become more consistent with a normal distribution.

Again, most of the time the Mean Squared Error is a safe and reasonable choice, and can be used without too much scrutiny. Nevertheless it is useful to have a sense of its underlying assumptions so that one can detect when they have been violated.

[7] This type of diagnostic plot can be generated easily with a library function, e.g. this one was generated with `scipy.stats.probplot`.

3.3 Feature Engineering

Along with the simple linear function relating features to labels as in **??** come significant limitations in terms of what kinds of relationships can be modeled with linear regression techniques. When trying to model asymptotic, periodic, or other non-linear relationships between features and labels, it is not yet clear how this can be accomplished given the limitations of this type of model.

As we shall see, complex relationships can be handled within the framework of linear models, so long as we exercise care by appropriately transforming our features (and labels). In practice, the success or failure of our models will often depend on carefully processing our data to help the model uncover the most salient relationships. This process of *feature engineering* proves critical even when developing deep learning models based on images or text: in spite of the vague promise of learning complex non-linear relationships automatically, extracting meaningful signals from data is often a matter of careful engineering, rather than selecting a more complex model.

3.3.1 Simple Feature Transformations

The first model we fit in Equation (3.4) revealed a positive association between review length and ratings. However, fitting the data with a line (Figure 3.4) does not seem to fit the data very accurately. Fitting the data with a line seems limiting, given that the trend may be better captured by a polynomial or asymptotic function (since the rating can not grow above five stars).

Naïvely, we might think that this is a fundamental limitation of linear models. Note however that the assumption of linearity in θ (Equation (3.2)) does not prevent us from fitting (for example) a polynomial function. The polynomial equation

$$\text{rating} \simeq \theta_0 + \theta_1 \times (\text{review length}) + \theta_2 \times (\text{review length})^2 \tag{3.33}$$

is linear in θ, even though we have transformed the input features in X.

This idea can be applied straightforwardly to fit polynomial functions, as shown in Figure 3.8.[8]

[8] Actually, these curves were generated using the feature $\frac{\text{length}}{1000}$, as the matrix inverse $(X^T X)^{-1}$ becomes numerically unstable given large values of length^3.

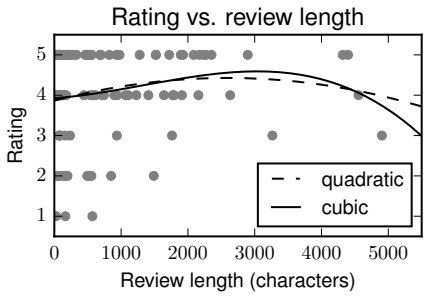


Figure 3.8: Quadratic and cubic polynomials of best fit.

3.3.2 Binary and Categorical Features: One-Hot Encodings

So far we have dealt with regression problems where we have both real-valued inputs (features X), and real-valued outputs (labels y). What can we do in cases where features are binary or categorical?

As an example, let's consider whether the length of a user's review can be predicted by (or more simply, is related to) their gender. To do so, we'll look at a different dataset (of a few hundred beer reviews) that includes the gender of its users.

That is, we'd like a model of the form:

$$\text{length} \simeq \theta_0 + \theta_1 \times \text{gender}. \tag{3.34}$$

Obviously, gender (represented in this dataset as a string) is not a numerical quantity, so we need some appropriate *encoding* of the gender variable.

For the moment, let's treat gender as a binary variable. We'll relax this assumption in a moment to allow for a non-binary gender variable (and allow for the possibility that the gender is missing, as it can be in this dataset), but for the moment let's encode the gender variable as:

$$\text{Male} = 0; \quad \text{Female} = 1. \tag{3.35}$$

Alternately, this is just a binary indicator specifying whether this user is female. This encoding, although only one of a few we might have used, allows us to fit a linear model and estimate the values of θ_0 and θ_1. The model we fit (after removing users who did not specify a gender) is

$$\text{length (in words)} \simeq 127.07 + 8.76 \times (\text{user is female}). \tag{3.36}$$

With a little thought, we can interpret the model parameters as indicating that, on average, females write slightly longer reviews (by 8.76 words) compared to males. Note that 127.07 is not the population average, but rather the average for males (whose gender feature is zero); we will revisit how to interpret the parameters of these models in Section 3.4.

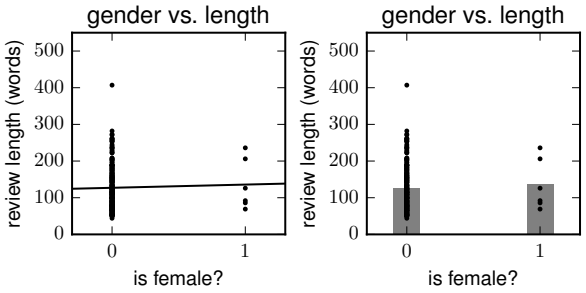


Figure 3.9: Gender versus review length (beer data). Visualized via a line of best fit (left) and a bar plot (right).

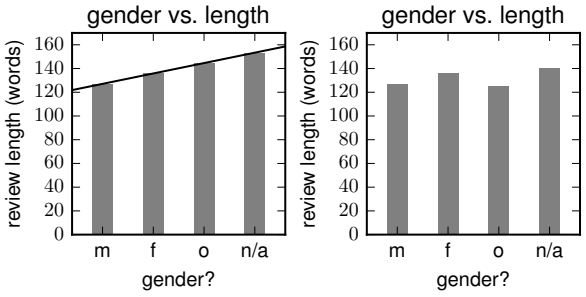


Figure 3.10: Categorical features with a naïve sequential encoding (left), and a one-hot encoding (right).

A scatter plot of the data (i.e., the encoded gender attribute and the review lengths), as well as the line of best fit above is depicted in Figure 3.9. Note that although we have fit the data with a line (Figure 3.9, left), the actual feature values only occupy two points (0 and 1); thus the fit is perhaps better represented with a bar plot (Figure 3.9, right).

Categorical Features

In practice, the gender attribute may assume more than binary labels in some datasets. To accommodate this, we might naïvely imagine extending our encoding from Equation (3.35) to include additional values:

$$\begin{aligned} \text{Male} &= 0; \\ \text{Female} &= 1; \\ \text{Other} &= 2; \\ \text{Not specified} &= 3; \\ \text{etc.} & \end{aligned} \tag{3.37}$$

Again we fit the same model as in Equation (3.34). Doing so we might obtain a fitted model like the one in Figure 3.10 (left).

Note that the model fit in Figure 3.10 (left) implicitly makes some dubious assumptions. For example, because the model is linear, it assumes that the

difference between 'male' and 'female' lengths is *the same as* the difference between 'female' and 'other' lengths. [9]

This assumption is not supported by the data, and in fact would be different if we simply reordered our indices in Equation (3.37). Rather, we would like to associate different predictions to members of each group, as in Figure 3.10 (right). This can be achieved via a different encoding:

$$\begin{array}{ll} \text{Male} = & [0,0,0] \\ \text{Female} = & [0,0,1] \\ \text{Other} = & [0,1,0] \\ \text{Not specified} = & [1,0,0] \\ \text{etc.} & \end{array} \tag{3.38}$$

We can quickly confirm that the model would make predictions as follows:

$$\begin{array}{ll} \text{Male:} & y = \theta_0 \\ \text{Female:} & y = \theta_0 + \theta_1 \\ \text{Other} & y = \theta_0 + \theta_2 \\ \text{Not specified} & y = \theta_0 + \theta_3 \\ \text{etc.} & \end{array} \tag{3.39}$$

That is, θ_0 is the prediction for males, θ_1 is the *difference* between females and males, etc. Note that we now have four parameters to estimate four values, as opposed to two parameters as in Equation (3.37). As such, the model has sufficient flexibility to make different estimates for each group, as in Figure 3.10 (right).

This type of encoding, in which we have a separate feature dimension for each category, is called a *one-hot* encoding.

Note that to represent four categories in Equation (3.38) we only used *three*-dimensional features (or in general, for N categories, we could use an $(N-1)$-dimensional encoding. Possibly this seems slightly confusing compared to using a four-dimensional feature vector (e.g. Male $= [0,0,0,1]$, etc.). Two reasons for using an $(N-1)$ dimensional feature vector are as follows:

- Doing so is not necessary; together with θ_0, the representation in Equation (3.38) uses four parameters to predict four values, so adding an additional dimension would add no more expressive power to the model and would be redundant.
- Doing so could possibly be harmful. While adding a redundant feature seems harmless, in practice it means the system of equations in **??** would no longer have a unique solution, as the matrix $X^T X$ would be uninvertable (**??**).

[9] That is, males receive the prediction θ_0; females receive $\theta_0 + \theta_1$; other receives $\theta_0 + 2\theta_1$, etc.

Similarly, a *multi-hot* encoding can be used in cases where an instance can belong to multiple categories simultaneously, for example for a 'race' feature, a user may associate with multiple racial groups. Note that this is equivalent to a concatenation of several binary features.

3.3.3 Missing Features

Often datasets will have features that are missing, for example the underlying data used for the example in Section 3.3.2 consisted of a gender attribute that many users may leave unspecified.

When dealing with binary or categorical features we dealt with these missing values quite straightforwardly—we simply treated 'missing' as an additional category.

But if a continuous feature, such as a user's age or income, were missing, we must think harder about how to handle it. Trivially, we might simply discard instances with missing features, though this strategy will harm model performance if it means discarding a substantial fraction of our data.

Alternately we might replace the missing entries by the average (or mode) value for that feature; this strategy is known as 'feature imputation.' This may be more effective than discarding the feature, but may also introduce some bias, as (for example) users who choose to leave a feature unspecified may be quite different from the average or mode.

To avoid the above issues, we would like a strategy that uses features when they are available, but makes separate predictions for those users when they are not. This can be achieved via the following strategy: for any feature x which is sometimes missing, replace it by two features x' and x'' as follows:

$$x' = \begin{cases} 1 & \text{if feature is missing} \\ 0 & \text{otherwise} \end{cases}, \quad x'' = \begin{cases} 0 & \text{if feature is missing} \\ x & \text{otherwise} \end{cases}. \tag{3.40}$$

Following this parameters can be fit within a model as usual:

$$y = \theta_0 + \theta_1 x' + \theta_2 x''. \tag{3.41}$$

The above representation may seem somewhat arbitrary, but makes sense once we expand the expression for missing and non-missing features. E.g. when a feature is available predictions are made according to

$$y = \theta_0 + \theta_2 x, \tag{3.42}$$

whereas when a feature is missing predictions are made according to

$$y = \theta_0 + \theta_1. \tag{3.43}$$

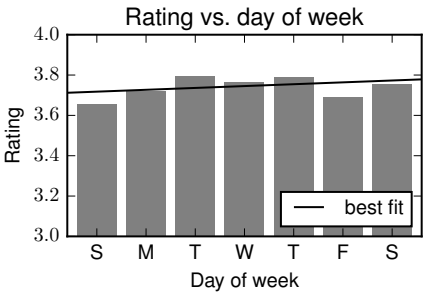


Figure 3.11: Ratings as a function of the weekday, and line of best fit.

This achieves the desired effect: when the feature is available we predict as normal, and when the feature is unavailable we predict using a learned value (θ_1). Note that this strategy is very similar to feature imputation, but rather than using a heuristic imputation strategy, the model will directly learn what is the best prediction to impute.

3.3.4 Temporal Features

Temporal features may make excellent predictors in various settings. Outcomes such as ratings, clicks, purchases (etc.) are often influenced by factors such as the day of the week, the season, or long-term trends that span several years.

Let's explore an example in which we try to predict the rating of a book on Goodreads based on the day of the week that it was entered. Average ratings for each weekday[10] are shown in Figure 3.11.

As before, we might try to describe this relationship using a line, i.e., to fit a model of the form

$$\text{rating} = \theta_0 + \theta_1 \times (\text{day of week}). \tag{3.44}$$

For this equation to make sense, we need to map the day of the week to a numeric quantity. A trivial encoding might assign numbers sequentially, e.g.

$$\text{Sunday} = 1; \quad \text{Monday} = 2; \quad \text{Wednesday} = 3; \quad \text{etc..} \tag{3.45}$$

Fitting Equation (3.44) using this representation yields the line of best fit depicted in Figure 3.11, which reveals a slight upward trend as the days of the week progress.

The linear trend in Figure 3.11 seems a fairly poor fit to the data; we might think about fitting more complex functions like polynomials to better capture

[10] Again based on a small sample of reviews from the Fantasy genre.

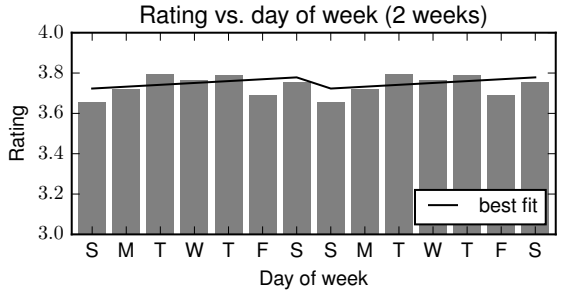


Figure 3.12: If we consider that our weekly measurements are periodic, we realize that fitting periodic data with a linear trend seems unrealistic.

the observed data. But consider that our model is essentially periodic: Sunday (represented by a 1) follows Saturday (represented by a 7), though we could just as easily have represented Wednesday as 1 and Tuesday as 7. These choices seem arbitrary, but they impact our model in unexpected ways.

The above point is perhaps clearer if we visualize our model's predictions over a period of two weeks, as in Figure 3.12.

We'll revisit the critical role of temporal features in the context of recommendation in **??**, and in

3.3.5 Transformation of Output Variables

Finally, just as we saw how to transform features in Section 3.3.1, we can also transform our *output* variables.

For example, let's consider fitting a model to determine whether resubmitted posts on *reddit* receive lower numbers of upvotes, i.e.,

$$\text{upvotes} = \theta_0 + \theta_1 \times (\text{submission number}) \tag{3.46}$$

(where the 'submission number' is '1' for an original resubmission, '2' for the first resubmission, etc.) This model, along with the observations on which it is based, are shown in Figure 3.13 (left).

Although the line of best fit indicates a slight downward trend, it does not appear to correspond closely to the overall shape of the data. Possibly we might imagine that the data follows an exponentially decreasing trend, e.g. every time you resubmit a post, you can expect to receive half as many upvotes.

Again, one might assume that this type of trend is something that cannot be captured by a linear model. But in fact we can possibly address this by transforming the output variable y. For example, consider fitting

$$\log_2(\text{upvotes}) = \theta'_0 + \theta'_1(\text{submission number}) \tag{3.47}$$

Now, a unit change in the prediction corresponds to a post receiving *twice*

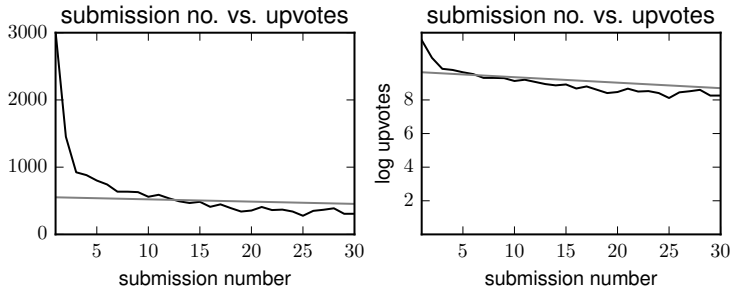


Figure 3.13 Number of upvotes versus submission number on *reddit*. The left plot shows the original data (with averaged upvote counts), the right plot shows the logarithm of the number of upvotes. Lines of best fit for both plots are included.

as many upvotes. While this is still a linear model, the model corresponds to fitting

$$\text{upvotes} = 2^{\theta_0' + \theta_1' (\text{submission number})}. \tag{3.48}$$

The transformed data and line of best fit are shown in Figure 3.13 (right).

Arguably, this second line better captures the overall trend, and does not have the same issues with outliers. In fact, if we transform the fitted values from Equation (3.47) back to their original scale via Equation (3.48), the transformed values actually have a Mean Squared Error about 10% *lower* than the model from Equation (3.46), indicating that the transformed data more closely follows a linear trend compared to the untransformed data.

3.4 Interpreting the Parameters of Linear Models

When analyzing the linear models developed so far, we have already talked about interpreting their parameters in terms of general trends, correlation, differences between groups, etc.

While is tempting to casually interpret the meaning of various features, we must be careful and precise about doing so.

First, we should be precise about the interpretation of our slope and intercept terms. For example, when we modeled rating as a function of review length (Equation (3.10)), we stated that under our model, ratings increased fractionally (1.194×10^{-4}) for every character of a review.

This interpretation makes sense given a model containing only a single features, but as soon as we incorporate multiple features we must be more careful. Consider e.g. the model from Equation (3.13), in which we included both the

Figure 3.14 Interpreting the parameters of linear models
Given a linear model $y = X\theta$ we should interpret a parameter θ_k as follows:
For every unit change in x_{ik}, our prediction of the output y_i would increase by θ_k if all other features remain fixed
It is important to note that we are talking about the model's *prediction* (rather than an actual change in the label), which could change if different features were included. And we must include the condition that other features remain constant. Without which we would fail to account for the correlations among different features.

length and number of comments as predictors. We could no longer state that under this model, that the rating increases (by 7.243×10^{-5}) for every character in the review. Precisely, we must interpret the parameters as follows: *Our prediction of the rating increases by 7.243×10^{-5} for every character in the review, assuming the other features remain unchanged*. This definition is stated precisely in Figure 3.14.

Critically, features like review length and number of comments may be highly *correlated*, such that we would rarely see longer reviews without also seeing more comments. For example, when incorporating features based on polynomial function (as in Equation (3.33)), or when dealing with one-hot encodings (as in Equation (3.37)), a feature generally *cannot* change without the other features changing.

Second, we should be clear when interpreting parameters that we are talking about *predictions* under a particular model rather than actual changes in the output variable y_i. These predictions can change as we include additional features, and a feature that had previously been predictive may be less predictive in the presence of another (as we saw in Equation (3.13)); likewise, we should be careful not to conclude that (e.g.) length is *not* correlated with the output, simply because another feature is *better* correlated.

Finally, we should be careful not to make statements about the *causal* effect of features on the output variable. Our line of best fit does not state that long reviews 'cause' positive opinions any more than it states that positive opinions cause long reviews.

3.5 Fitting Models with Gradient Descent

When solving regression problems in the previous section, we looked for *closed form* solutions. That is, we set up a system of equations (Equation (3.2)) in θ, and attempted to solve them for θ (albeit approximately via the pseudoinverse).

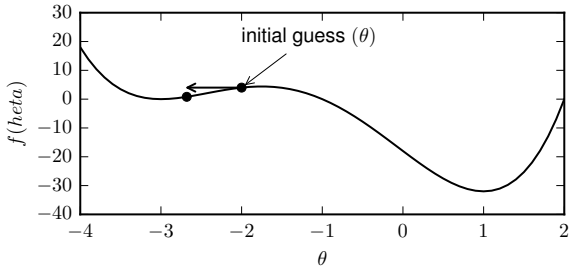


Figure 3.15: Gradient descent demonstration.

As we begin to fit more complex models including in Sections 3.7.2 and 3.8, a closed-form solution may no longer be available.

Gradient descent is an approach to search for the minimum value of a function, by iteratively finding better solutions based on an initial starting point. The process (depicted in Figure 3.15) operates as follows:

(i) Start with an initial guess for θ
(ii) Compute the derivative $\frac{\partial}{\partial\theta} f(\theta)$. Here $f(\theta)$ is the MSE (or whatever criterion we are optimizing) under our model θ.
(iii) Update our estimate of $\theta \coloneqq \theta - \alpha \cdot f'(\theta)$
(iv) Repeat Steps (ii) and (iii) until convergence.

During each iteration, the process now follows the path of steepest descent, and will gradually arrive at a minimum of the function f_θ.[11]

The above is a simple description of the procedure that omits many details. We omit some of the specifics for now and save them until we implement the procedure in practice, though some of the main issues include:

- Given the starting point in Figure 3.15, the algorithm would only achieve a *local* rather than a *global* optimum. To address this we could investigate ways to come up with a better initial 'guess' of θ, or later, we could investigate alternative optimization approaches less susceptible to local minima.
- The step size α (Step (iii) above) must be chosen carefully. If α is too small, the procedure will converge very slowly; if α is too large, the procedure may 'overshoot' the minimum value and obtain a *worse* solution during the next iteration. Again, other than carefully tuning this parameter, we could investigate optimization methods not dependent on choosing this rate (see e.g. quasi-Newton methods such as L-BFGS Liu and Nocedal (1989)).

[11] Assuming the function is 'well-behaved,' e.g. the objective is bounded below, and the function is differentiable everywhere; though these are rarely issues when dealing with error functions specifically.

- 'Convergence' as defined in Step (iv) is not well-defined. We might define convergence in terms of the change in θ (or $f_\theta(X)$) during two successive iterations, or alternately we may terminate the algorithm once we stop making progress on held-out (validation) data, as in **??**.

3.5.1 Linear Regression via Gradient Descent

To solidify the ideas above, let's consider the specific example of minimizing the Mean Squared Error of a linear model, i.e.,

$$f_\theta(y; X) = \frac{1}{|y|} \sum_{i=1}^{|y|} (x_i \cdot \theta - y_i)^2. \tag{3.49}$$

The derivative $f'(\theta)$ can be computed as follows:

$$\frac{\partial f}{\partial \theta_k} = \frac{1}{|y|} \sum_{i=1}^{|y|} 2x_{ik}(x_i \cdot \theta - y_i). \tag{3.50}$$

Note that the above is a *partial* derivative in θ_k, which must be computed for each feature dimension $k = \{1 \dots K\}$. The above derivative is more obvious after expanding $x_i \cdot \theta = \sum_{k=1}^{K} x_{i,k}\theta_k$.

3.6 Non-linear Regression

So far, we have limited our discussion to models of the form $y = X\theta$, mostly because these offered us a convenient (closed form) solution to finding lines of best fit in terms of θ.

However, this type of model has several limitations that we might wish to overcome, such as:

- We cannot incorporate simple constraints on our parameters, such as that a certain parameter should be positive, or that one parameter is larger than another (which might be based on domain knowledge of a certain problem).
- Although we can manually engineer non-linear transforms of our features (as we did in Section 3.3.1, we cannot have the model learn these non-linear relationships automatically.
- The model cannot learn complex *interactions* among features, e.g. that length is correlated with ratings, but only if the user is female.[12]

[12] To be precise, the linear model does consider relationships among features in the limited sense that parameters for one feature will change in the presence of other correlated features (Section 3.4); and, the model *could* capture relationships between (e.g.) gender and length if

The above goals can potentially be realized if we are allowed to transform model *parameters*: for instance, we could ensure that a particular parameter was always positive by fitting

$$\theta_k = \log(1 + e^{\theta'_k}) \tag{3.51}$$

(this is known as a 'softplus' function; note that this function smoothly maps $\theta'_k \in \mathbb{R}$ to $\theta_k \in (0, \infty)$); or if we wanted one feature to be larger than another (e.g. $\theta_k > \theta_j$) we could simply add the positive quantity above to another feature:

$$\theta_k = \theta_j + \log(1 + e^{\theta'_k}). \tag{3.52}$$

Although it goes beyond our discussion here, fitting these types of non-linear parameterizations (and especially transformations that deal with complex combinations of parameters) is largely the goal of *deep learning*.

We will see examples of non-linear models in later chapters (e.g. Section 6.6,Section 4.5) though for the moment we briefly discuss

```
y = tf.constant(y, shape=[len(y),1]) # Convert to column vector
M = len(X[0]) # Feature dimensionality

# Regularized MSE, described using tensorflow matrix operations
def MSE(X, y, theta, lambda):
  return tf.reduce_mean((tf.matmul(X, theta) - y)**2) +
         lambda * tf.reduce_sum(theta**2)

# Initialize theta
theta = tf.Variable(tf.constant([0.0]*M, shape=[M,1]))

# Adam Optimizer
optimizer = tf.train.AdamOptimizer(0.01)
objective = optimizer.minimize(MSE(X,y,theta,1.0))
```

3.6.1 Case Study: Image Popularity on Reddit

Lakkaraju et al. (2013) used regression algorithms to estimate the success of content (e.g. number of upvotes) on *reddit*.

Presumably, one of the biggest predictors of success is the quality of the content itself. Predicting whether submitted content is of high quality (e.g. whether an image is funny or aesthetically attractive) is presumably incredibly challenging. To control for this high-variance factor of content quality, Lakkaraju et al. (2013) study *resubmissions*, i.e., content (images) that has been submitted

we were to manually engineer a feature describing this relationship. Our point here is about whether the model can learn these relationships automatically.

multiple times. This way, if one submission is more successful than another (of the same image), the difference in success cannot be attributed to the content itself, and must arise due to other factors such as the title of the submission or the community it was submitted to.

Having controlled for the effect of the content itself, the goal is then to distinguish between features that capture the specific dynamics of reddit itself, versus those that arise due to the choice of title (i.e., how the content is 'marketed'). Various features are extracted that model the reddit community dynamics, such as the following:

- One of the largest predictors of successful content is simply whether it has been submitted before (as we saw in Figure 3.13, which is based on the same dataset); this is captured via an exponentially decaying function.
- However, the above effect might be mitigated if enough time has passed between resubmissions (by when the original submission is forgotten, or the community has enough new users); this is captured using a feature based on the *inverse* of the time delta between submissions.
- Resubmissions might still be successful if they are resubmitted to largely non-overlapping communities (subreddits).
- Submission success may correlate with the time of day. For example, submissions may be most successful during the highest-traffic times of day, or alternately they may be more successful if submitted when there is less competition.

Whereas community effects are somewhat reddit-specific, a question of broader interest is to determine the effect that a particular choice of *title* has on the success of a submission. Understanding the characteristics of a successful title can have broader implications when marketing content (e.g. an advertising campaign) to a new market (for example).

Features are extracted to capture the dynamics of a submission's title, including:

- Titles should differ from those previously used in submissions of the same content.
- Titles should align with the expectations of the community the content is submitted to. Interestingly, Lakkaraju et al. (2013) find that there is a 'sweet spot,' in the sense that titles should roughly follow the linguistic style of previous successful submissions in the same community, but should not be *too* similar, to the point that they are not novel compared to previous submissions. We'll discuss text similarity measures more in Chapter 7.

- Successful titles might have other features, in terms of length, sentiment, linguistic style, etc.

Ultimately, all of the above features are combined into a regression model that estimates the score (number of upvotes minus number of downvotes) that a particular submission will receive.

Due to the way that features are combined, the model is not linear in the parameters, so optimization proceeds by gradient descent (as in Section 3.5). The method is evaluated in terms of the R^2 coefficient, with experiments revealing that community and textual features both play a key role in prediction. Finally, it is shown that the method can be used 'in the wild' to predict the success of actual reddit submissions.

3.7 The Learning Pipeline

By now we have covered many of the individual components that go into building a predictive model: model fitting (**??**), feature engineering (**??**), and evaluation (**??**). Bringing these components together still requires filling in some additional details. How can we know whether our model will work well when deployed (i.e., on *new* data), and what steps can be taken to ensure this? How can we decide between various alternatives in terms of feature design, and meaningfully compare those alternatives against each other? Collectively these steps are part of the *pipeline* of machine learning.

3.7.1 Generalization and Overfitting

So far, when discussing model evaluation in Section 3.2, we've considered training a model to predict labels y from a dataset X; we've then evaluated the model by comparing the predictions $f(x_i)$ to the labels y_i. Critically, we're using the *same* data to train the model as we're using to evaluate it.

The risk in doing so is that our model may not *generalize* well to new data. For example, when fitting a model relating review length to ratings (as in **??**), increasing the degree of the polynomial would continue to lower the errors of the predictor; alternately, we could have modeled review length using a one-hot encoding (so that there was a different predicted value for every length). In spite of their low MSEs (or high R^2 coefficients), these models would not seem to meaningfully capture the *trends* in the data; rather, they are effectively 'memorizing' meaningless details in the observed data.

To consider an extreme case, imagine fitting a vector y using only *random*

features. The code below fits a vector of 50 observations using 1, 10, 25, and 50 random features, and then prints the R^2 coefficient of each model:

```
1 y = numpy.random.rand(50)
2 mod = linear_model.LinearRegression()
3 for n in [1,10,25,50]:
4     X = numpy.random.rand(50,n)
5     mod.fit(X,y)
6     print(mod.score(X,y))
```

Here, the R^2 coefficients take values of 0.07, 0.25, 0.35, and 1.0—once we include 50 random features, we can fit the data perfectly. Of course, given that our features were *random*, this 'fit' is not meaningful, and the model has merely discovered random correlations between the observed data and labels.

The above points to two issues that need to be addressed when training a model:

(i) We should not evaluate a model on the same data that was used to train it. Rather we should use a held-out dataset (i.e., a *test* set).
(ii) Features that improve performance on the training data will not necessarily improve performance on the held-out data.

Evaluating a model on held-out data gives us a sense of how well we can expect that model to work 'in the wild.' Thus we are measuring how well our model will *generalize* to new data.

Overfitting Fundamentally, if our model works well on training data but not on held-out data, it must mean that certain characteristics of the training data are somehow not representative of the held-out data. This could occur for various reasons. One possibility is that our held-out data is drawn from a different distribution from the training data. For instance, if we had held-out sales data from the most recent month, and trained on data from the previous eleven, it is possible that the most recent month of observations follow a different trend, or occurs during a different season, etc. In principle, one might address the above simply by ensuring that the training and test sets are (non-overlapping) *random samples* of the data, such that both the training and test data will be drawn from the same distribution.[13]

Even if the training and held out data are independently drawn samples from the same distribution, we may still observe significantly degraded performance on our held-out data. In such cases we are said to be *overfitting*.

A trivial demonstration of overfitting is shown in Figure 3.16. Here we show

[13] Though if our goal is to *forecast* next month's sales, using the most recent data as our held out sample may be the most appropriate decision.

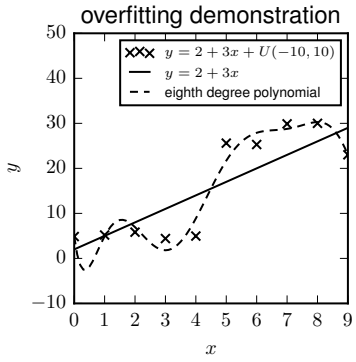


Figure 3.16: Overfitting demonstration. A high-degree polynomial fits the observed data accurately, but is unlikely to generalize well.

a dataset that follows a line, subject to some random perturbation. While a high-degree polynomial can fit the data very closely, we would not expect this complex function to generalize well to new data. This again demonstrates the principle of Occam's Razor, as we saw in Section 3.7.2, where we would expect the simpler model to better fit new data. We are said to have *overfit* when we fit a model that is highly accurate on the training data, but that does not generalize well.

Note that we *expect* any model to perform somewhat worse when applied to new data compared to its training performance: in fact this is one of our 'theorems' about model performance that we present in **??**. Rather our goal when tuning the parameter λ (Equation (3.55)) using our validation set is to minimize this gap, typically by sacrificing training accuracy in order to improve generalization performance.

Underfitting Just as we overfit by fitting a model whose good performance on a training set does not generalize to a held-out set, we *underfit* when our model is insufficiently complex to capture the underlying dynamics in a dataset. Again, this can occur for a variety of reasons. If we select too simple a model, e.g. a linear function to capture the data in **??** (which doesn't seem to follow a linear trend), no choice of parameters will lead to good training *or* held-out performance. Or, as above, if we tune λ (Equation (3.55)) poorly (λ too large in the case of underfitting), we will sacrifice training performance unnecessarily by choosing too simple a model.

3.7.2 Model Complexity and Regularization

So far we have talked vaguely about what it means for a model to be 'too complex' and suggested that we should choose a model that is complex enough

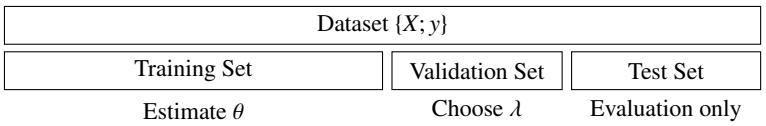


Figure 3.17 Basic roles of training, validation, and test sets.

to fit the data, but simple enough not to overfit. This idea is often referred to as *Occam's Razor*, a philosophical principle which states that among several alternate hypotheses that explain some phenomenon, one should favor the simplest.

However for these notions to be useful we must be precise about what it means for a model to be 'complex.' We would like to define complexity in terms of the parameters θ, such that given a fixed set of features and labels, we could select the 'simplest' θ that adequately explains (or models) the data.

We'll discuss two candidate notions of 'simplicity' as follows:

- A simple model is one in which all terms are about equally important, i.e., one in which all terms θ_k are about equally large.
- A simple model is one that includes only a few terms, i.e., in which only a few values θ_k are non-zero.

Below we'll argue that these two notions of 'complexity' are captured by the following expressions:

$$\Omega_2(\theta) = ||\theta||_2^2 = \sum_k \theta_k^2 \tag{3.53}$$

$$\Omega_1(\theta) = ||\theta||_1 = \sum_k |\theta_k|, \tag{3.54}$$

i.e., the sum of squares and sum of absolute values.

In order to fit a model which simultaneously explains the data but is not overly complex (corresponding to our goal above), we write down a new objective that combines our original accuracy objective from **??** with one of the complexity expressions above:

$$\underbrace{\frac{1}{|y|}\sum_{i=1}^{|y|}(x_i \cdot \theta - y_i)^2}_{\text{accuracy}} + \underbrace{\lambda \sum_k \theta^2}_{\text{model complexity}} \tag{3.55}$$

Note that generally speaking the term θ_0, i.e., the *offset* term, should not be included in the regularizer, i.e., our regularizer should be $\sum_{k=1}^{K} \theta_k^2$ or $\sum_{k=1}^{K} |\theta_k|$. That is, our underlying assumption that few parameters are non-zero, or that

parameters are small, should not apply to the offset term. If we were to include the offset term when regularizing, we would generally select a model which made systematically smaller (in magnitude) predictions.

Validation Sets We now require a protocol for choosing the best value of the trade-off parameter λ in Equation (3.55). If we were to select λ based on the accuracy on the training set, we would always select $\lambda = 0$, and Ideally, we want to choose the value of λ that will result in the best performance on the held-out test set. However recall that we cannot use the test set to compare alternative models: the test set is supposed to represent true held-out performance, and strictly speaking test performance should only be examined *after* we have selected our best model.

As such, we need a third partition of our data which can be used to select among alternative models. This *validation set* in some sense mimics the test set, in that it is not used to fit the model, but is used to give us an *estimate* of what we expect the test performance to be under certain modeling conditions.

As such, a typical pipeline will consist of three partitions of our data, whose roles are summarized as follows:

- The **training set** is used to optimize the parameters of a specific model. In the type of models we fit in this book, this usually refers to the parameters that can be fit via gradient ascent/descent (i.e., θ in this chapter).
- The **validation set** is used to select among model alternatives. 'Model alternatives' may simply mean different values of λ in Equation (3.55), but could also mean different feature representation strategies, etc. We discuss a few alternative uses below. Typically we select the model with the highest accuracy/lowest error on the validation set.
- The **test set** is used to evaluate held-out model performance; ideally it should not be used to make any modeling decisions, but should only be used to report performance.

The relationship between these three sets is shown in Figure 3.17, and some overall guidelines for building a validation set are shown in Figure 3.19.

Why does the ℓ_1 norm induce sparsity? Figure 3.18 demonstrates (for a simple two-parameter model) why the ℓ_1 norm induces sparsity (i.e., few non-zero paramaters) while the ℓ_2 norm results in a more uniform parameter distribution. Models with equivalent ℓ_1 norm lie along a diamond-shaped contour (Figure 3.18, left) whereas models with equivalent ℓ_2 norm lie on a circle (Figure 3.18, right). Models with an equivalent mean squared error lie along an ellipse. When balancing the error and the regularizer as in Equation (3.55), the

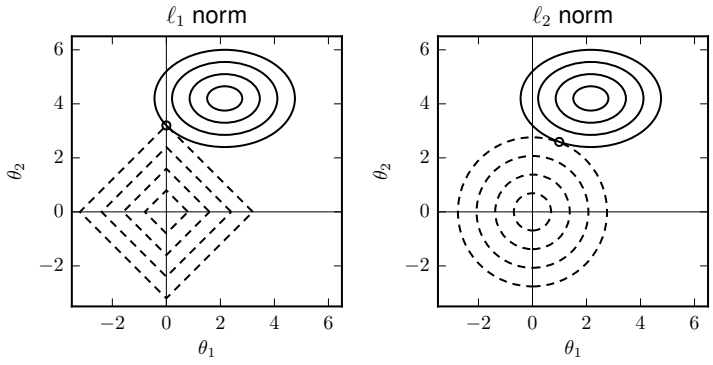


Figure 3.18 Demonstration of the regularization effect of the ℓ_1 (left) versus ℓ_2 norms. Dashed lines indicate models with equivalent norms (ℓ_1 or ℓ_2); solid lines indicate models with equivalent mean-squared errors. The selected model in either condition is circled.

Figure 3.19 Guidelines for building training, validation, and tests sets

- Training, validation, and test sets should (generally speaking) be non-overlapping, random samples of a dataset. Some exceptions apply, for example when fitting temporal models in Chapter 6, we might build a test set out of the *most recent* observations in order to get a sense of how well a model would work *now* rather than how well it would work *on average*.
- The size of our training set may be driven by modeling as well as practical concerns. Our training set should be large enough that we can reasonably expect to fit our model on the data (as a guideline, we might hope to have an order of magnitude more training examples than model parameters); likewise if we have a simple model with just a few parameters we need not train on millions of observations.
- Likewise the size of our validation and test sets should be large enough that we can be reasonably confident of our results. We briefly touch upon measuring significance in **??**

best model will correspond to the point where the boundaries intersect. In the case of the ℓ_1 norm these curves intersect on *on one of the vertices of the diamond*; for the ℓ_2 norm they do not. The former case corresponds to a model with only a few non-zero parameters. For a more rigorous explanation of this phenomenon see e.g. Friedman et al. (2001).

'Theorems' Regarding Training, Testing, and Validation Sets To solidify the roles of training, validation, and test sets, below we outline some theorems guiding the relationships among these sets, as the regularization hyperparameter λ changes.

Note that these are 'theorems' in the sense that they will be true in gen-

Figure 3.20 Example train, validation, and test curves, demonstrating the relationships between each type of error.

eral, but only in the limit given large enough datasets, and that our training, validation, and test sets are drawn from the same distribution (etc.). As such, these theorems should mostly be regarded as guidelines to 'sanity check' the correctness of your model pipeline:

- The training error increases as λ increases; typically for large λ it will asymptote to some value, e.g. a linear model might asymptote to the error of a trivial predictor (i.e., the variance of the label).
- The validation and test errors will be at least as high as the training error; intuitively, the algorithm will not work better on 'unseen' data than it did on training data.
- When λ is too small, a too-complex model achieves low training error, but high validation/test errors. In this case, the model is said to be *overfitting*.
- When λ is too large, a too-simple model has high training, validation, and test error. In this case, the model is said to be *underfitting*.
- Generally, there should be a 'sweet spot' between under- and over-fitting, which is determined using our validation set. This point (marked in **??** with an 'x') corresponds to the model we expect to yield the best generalization performance on the test set.

3.7.3 Guidelines for Model Pipelines

Having introduced the conceptual details of a model pipeline, it is worth finishing with some practical advice on how to combine the various pieces and how to set the various tunable components of a model pipeline:

- If you do *not* see a 'sweet spot' between under- and over-fitting (as in the theorems above), it could mean that you have not adequately explored the range of regularization coefficients. For example, if you observe monotonically increasing validation errors, it may mean you have not considered sufficiently small values of λ. Alternately, given (e.g.) a simple linear model with only a few parameters, it may simply mean that your model is *not capable* of overfitting to a particular dataset.
- Regularization parameters such as λ do not have an absolute scale, and will vary depending on factors ranging from the model's tendency to overfit, to the specific scale of the features x and labels y. As a rough guideline, it is useful to consider setting *lambda* by considering several different orders of magnitude (as we do in **??**), before honing in on a narrower range of values.
- When implementing iterative models (such as approaches based on gradient descent, as in **??**), the validation set can be used as a condition to cease further iteration. That is, we need not train models until convergence: if we are making no further improvements on the validation set (say, for a predetermined number of iterations), there is little reason to continue optimizing our model on the training set. Ideally, the model parameters θ might be chosen from whichever iteration yields the best *validation* performance.

Uses of a Validation Set The core use of our validation set is to estimate model hyperparameters such as λ above. Beyond regularization coefficients, 'hyperparameters' more broadly refer to any tunable model components that do not get optimized during the training phase. For example, when building models from text in Chapter 7, the number of words in our dictionary (from which we build features) would be an example of a hyperparameter, and could be chosen using our validation set to select the model that will generalize the best.

Our validation set can also be used when training a model. For instance, when we fit models via gradient descent (as in Section 3.5), we might periodically evaluate the model's validation performance every few gradient iterations; rather than waiting until the model reaches a particular convergence threshold, we can simply terminate once performance is no longer improving on the validation set.

3.7.4 Significance Testing

Although not a focus of this book, it is worth briefly exploring how we can formally determine whether the performance of one model is 'better' than another, in terms of a formal statistical framework. So far, we have compared

models in terms of their mean-squared errors though as we explored in **??**, the mean squared error was chosen based on an underlying assumption that model errors are randomly distributed (under a Gaussian distribution).

Significance testing refers to the overall process of determining whether a statistical measurement would have been likely to have occurred due to chance alone (under the assumptions of some particular model). For example, if an established restaurant on *Yelp* has a rating of 4.3 stars based on 50 reviews, and a new restaurant has a rating of 4.5 stars based on four reviews, would you conclude that the new restaurant is better rated? Or would you conclude that the higher initial rating is likely to have occurred due to chance? Significance tests allow us to formalize these questions.

Formally, a *p-value* measures the probability that a result as (or more) extreme as the one we actually observed could have occurred due to chance (under some statistical model). E.g. if we assume that users' ratings follow a Gaussian distribution, with what probability would the ratings of two restaurants deviate by more than 0.2 stars? In the case of this specific measurement, this probability would depend on (a) the *magnitude* of the difference between the two averages; (b) the *size* of the two samples (e.g. a difference of 0.2 stars might be significant of both restaurants had 50 ratings, but not if they had four); and (c) the *variance* of the two samples (e.g. if the two samples had highly concentrated ratings we might more quickly conclude that the difference was significant).[14]

When making comparisons between models, we will typically use a p-value to measure whether one model has residuals $(f(x) - y)$ that are closer to zero than another (i.e., we are testing whether one model's predictions are closer to the labels than the other's). To do so we are measuring the difference in *variance* between two samples.

We will compute this quantity via an *F-test*. Other tests could also be used to compare the performance of two models, such as a likelihood ratio test, each of which has different underlying assumptions. Below we'll compare the performance of two models for estimating a rating, using our beer review data from **??**:

$$\text{model 1:} \quad \text{rating} = \theta_0 \tag{3.56}$$

$$\text{compared to model 2:} \quad \text{rating} = \theta_0 + \theta_1 \times \text{ABV} \tag{3.57}$$

One of the assumptions of this particular test is that one of the to models has a subset of the parameters of the other. As such we are really measuring whether the additional parameters significantly improve the model's perfor-

[14] This specific probability would be measured via a t-test.

mance (i.e., whether adding a term based on the ABV improves the performance of a model including only θ_0).

First we generate features and labels for the two models (assuming the data has already been read, shuffled, etc.):

```
X1 = [[1] for d in data]
X2 = [[1, d['beer/ABV']] for d in data]
y = [d['review/overall'] for d in data]
```

Next we fit the two models (on half of the data), and compute their residuals (on the other half):

```
model1 = sklearn.linear_model.LinearRegression(fit_intercept=False
    )
model1.fit(X0[:250], y[:250])
residuals1 = model1.predict(X1[250:]) - y[250:]
model2 = sklearn.linear_model.LinearRegression(fit_intercept=False
    )
model2.fit(X2[:250], y[:250])
residuals2 = model2.predict(X2[250:]) - y[250:]
```

The actual F statistic depends on the sum of squared residuals, the number of parameters in each model, and the size of the sample:

```
rss1 = sum([r**2 for r in residuals1]) # sum of squared residuals
rss2 = sum([r**2 for r in residuals2])
k1,k2 = 1,2 # Number of parameters of each model
n = len(residuals1) # Number of samples
```

Finally we compute the F statistic and estimate the associated p-value using a method from `scipy`:

```
F = ((rss1 - rss2) / (k2 - k1)) / (rss2 / (n-k2))
scipy.stats.f.cdf(F,k2-k1,n-k2)
```

The obtained p-value is close to zero, indicating that the result (that ABV improves predictive performance) is statistically significant.[15]

Note that this is just one example of a significance test, that works for a particular situation (albeit a fairly common one); in different situations alternative tests may be required, see e.g. Wasserman (2013) for a more comprehensive presentation.

In spite of the importance of rigorously demonstrating the significance of claimed model improvements, we will mostly avoid further discussing significance testing throughout the remainder of this book. Generally speaking, these types of tests are designed for small-sample contexts (e.g. surveys or clinical trials, etc.); on the types of large datasets we consider, even small differ-

[15] One can experiment with less-predictive features or a smaller test set to see how these influence the estimated p-value.

ences between models will tend to yield extremely small (highly significant) p-values.

3.8 Classification

So far, we have considered supervised learning tasks in which the output variable y is a real number, i.e., $y \in \mathbb{R}$. Often, we will deal with problems with binary or categorical output variables, for example we might be interested in problems such as:

- Will a user click on a product or advertisement? (binary outcome)
- What category of object does an image contain? (multiclass)
- What product is a user most likely to purchase next? (multiclass)
- Which of two products would a user prefer? (binary)

3.8.1 Logistic Regression

We will focus on a form of classifier that extends the ideas behind regression to classification problems.

Logistic Regression sets up classification using a probabilistic framework; the predictions $X\theta$ that we used when building regressors are transformed into *probabilities* of observing a particular label.

Ultimately, logistic regression is just one of dozens of classification schemes; we describe it here rather than other popular alternatives (such as Support Vector Machines Cortes and Vapnik (1995), or Random Forest Classifiers Ho (1995)) mainly because logistic regression more closely matches the approaches we will develop in later chapters.

In short, logistic regression associates a probability of the dataset under some model, which is fully differentiable and can be optimized using gradient-based approaches (as we saw in Sections 3.2.2 and 3.5). This same type of modeling approach will be used throughout this book, when building Recommender Systems in Chapter 4, or generating fashionable outfits in Chapter 8.

Fitting a logistic regressor When fitting regular linear regressors, we wanted a model f_θ whose estimates $f_\theta(X_i)$ were as close as possible to the (real-valued) labels y_i. When adapting a linear regression algorithm to classification, we might instead seek models that associate positive values of $X_i \cdot \theta$ with positive labels ($y_i = 1$), and negative values of $X_i \cdot \theta$ with negative labels ($y_i = 0$).

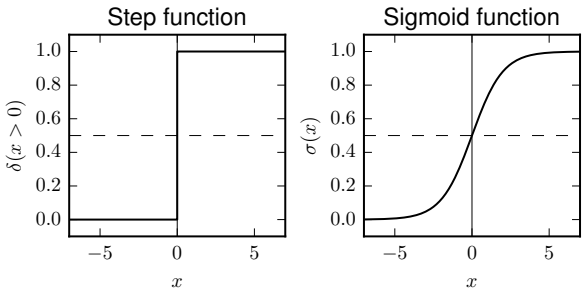


Figure 3.21: The sigmoid function (right), and the step function (left) that it approximates.

If we could do so, we could write down the *accuracy* associated with a particular model:

$$\frac{1}{|y|}\sum_{i=1}^{|y|} \underbrace{\delta(y_i = 0)\delta(X_i \cdot \theta \leq 0)}_{\text{label is positive and prediction is positive}} + \overbrace{\delta(y_i = 1)\delta(X_i \cdot \theta > 0)}^{\text{label is negative and prediction is negative}} \tag{3.58}$$

(here δ is an indicator function that returns 1 if the argument is true, 0 otherwise). The equation above, in spite of slightly confusing notation, is merely counting the number of times we correctly predict a positive score for positively labeled data, and a negative score for negatively labeled data.

We now simply desire from our classifier θ that it maximizes the accuracy in Equation (3.58). Unfortunately, directly optimizing the accuracy in Equation (3.58) is NP-hard (see e.g. Nguyen and Sanner (2013)). To get a sense for why it is difficult, consider that the function in Equation (3.58) is essentially a *step* function (Figure 3.21, left), i.e., it is flat (deravitive zero) almost everywhere; it is therefore not amenable to techniques like gradient ascent.

So, to optimize the accuracy approximately, we would like a function that is *similar to* Equation (3.58), but is more straightforward to optimize.

Logistic Regression achieves this goal by converting the outputs of a linear function $X_i \cdot \theta$ to *probabilities* via a smooth function. Our intuition is that large values of $X_i \cdot \theta$ should correspond to high probabilities, and small (i.e., large negative) values of $X_i \cdot \theta$ should correspond to small probabilities.

This goal can be achieved via the *sigmoid function*:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3.59}$$

This function, depicted in Figure 3.21, maps a real value to the interval $(0, 1)$, and passes through 0.5 when $x = 0$. Thus it can be interpreted as a probability:

$$p_\theta(y_i = 1|X_i) = \sigma(X_i \cdot \theta) = \frac{1}{1 + e^{-X_i \cdot \theta}} \tag{3.60}$$

Now, as a smooth surrogate for the expression in Equation (3.58), we can instead optimize

$$\mathcal{L}_\theta(y|X) = \prod_{y_i=1} p_\theta(y_i = 1|X_i) \times \prod_{y_i=0} (1 - p_\theta(y_i = 0|X_i)) \tag{3.61}$$

$$= \prod_{y_i=1} \frac{1}{1 + e^{-X_i \cdot \theta}} \times \prod_{y_i=0} \frac{e^{-X_i \cdot \theta}}{1 + e^{-X_i \cdot \theta}}. \tag{3.62}$$

The above expression is a *likelihood function*, much like we saw in Equation (3.20). Intuitively, for this expression to be maximized we want positive instances ($y_i = 1$) to be associated with high probabilities, and negative instances ($y_i = 0$) with low probabilities.

Our goal is to maximize the above function, i.e., to find $\arg\max_\theta L_\theta(y|X)$. Short of a closed form solution, our approach is as follows:

- Take the logarithm $\ell_\theta(y|X)$, since $\arg\max_\theta \mathcal{L}_\theta(y|X) = \arg\max_\theta \log(\mathcal{L}_\theta(y|X))$.
- *Subtract* a regularizer $\lambda||\theta||_2^2$ (or similar).[16]
- Compute the gradient of $\ell_\theta(y|X) - \lambda||\theta||_2^2$, and optimize via gradient ascent.

We compute the gradient below as follows:

$$\ell_\theta(y|X) = \sum_{y_i=1} \log\left(\frac{1}{1 + e^{-X_i \cdot \theta}}\right) + \sum_{y_1=0} \log\left(\frac{e^{-X_i \cdot \theta}}{1 + e^{-X_I \cdot \theta}}\right) - \lambda||\theta||_2^2 \tag{3.63}$$

$$= \sum_i -\log(1 + e^{-X_i \cdot \theta}) + \sum_{y_i=0} -X_i \cdot \theta - \lambda||\theta||_2^2 \tag{3.64}$$

$$\frac{\partial \ell}{\partial \theta_k} = \sum_i x_{ik} \frac{e^{-X_i \cdot \theta}}{1 + e^{-X_i \cdot \theta}} - \sum_{y_i=0} x_{ik} - \lambda\theta_k \tag{3.65}$$

$$= \sum_i x_{ik} \left(1 - \sigma(X_i \cdot \theta)\right) - \sum_{y_i=0} x_{ik} - \lambda\theta_k \tag{3.66}$$

(note the summation indices change between Equations (3.63) and (3.64), since both terms in Equation (3.63) have the same denominator).

Summary Our development of logistic regression above is representative of the overall approach we'll take later when developing models that estimate interactions, clicks, purchases, etc.:

- Rather than estimating an outcome directly, we associate a *probability* with

[16] Note that we subtract the regularizer rather than adding it as we did in **??** since we are maximizing rather than minimizing our objective; since we still want low complexity, we seek to maximize $-\lambda||\theta||_2^2$.

each outcome. Associating the outcome with probability allows us to associate discrete (e.g. $y_i \in \{0, 1\}$) outcomes with a continuous function ($f(x) \in (0, 1)$); this is accomplished via a transformation (such as the sigmoid function) which maps a real-valued output into the correct range.

- The model should associate positive (1) labels with high probabilities, and negative labels (0) with low probabilities. Likewise we can associate a probability to the entire dataset by taking a product of probabilities (or a sum of log-probabilities, as in **??**).
- Ultimately the procedures above allow us to associate the quality of a model (paramaterized by θ) with a continuous function whose value we should try to maximize; we optimize the model via gradient ascent.

3.8.2 Other Classification Techniques

In this introductory chapter, we have only discussed a single classification technique: Logistic Regression. Our choice to explore this particular technique was largely a practical one: the idea of associating a probability with a particular outcome (as in **??** and estimating that probability via a differentiable function (to facilitate gradient ascent) will appear repeatedly as we develop more and more complex models. Likewise, the choices of training and regularization strategies were largely guided by what will prove useful later.

However the specific techniques we've explored are only one set of approaches to build classifiers, and

Support Vector Machines While logistic regressors optimize a probability associated with a set of observed labels, they do not explicitly minimize the number of *mistakes* made by the classifier. Support Vector Machines (SVMs) Cortes and Vapnik (1995) replace the sigmoid function in **??** with an expression that assigns zero cost to correctly classified examples, and a positive cost to incorrectly classified examples (in proportion to the confidence of the prediction $\theta \cdot x$). This distinction is fairly subtle: while *every* sample will influence the optimal value of θ for a logistic regressor, the solution found by an SVM is entirely determined by a few samples closest to the classification boundary, or those that are mislabeled. Conceptually it is appealing for a classifier to focus on the most 'difficult' samples in this way, though note that in many cases (and notably when building recommender systems) our goal is to optimize ranking performance rather than classification accuracy (see e.g. **??**), such that giving special attention to the most ambiguous examples is not necessarily desirable.

Decision Trees Decision trees classify instances based on a sequence of binary decisions, each of which deals with a specific feature. Each node of the tree separates the data based on such a decision, with leaf nodes being responsible for determining an outcome. Decision trees straightforwardly facilitate learning non-linear classifiers that capture complex interactions among features, e.g. we can straightforwardly learn that a low price is associated with a positive review for young people, while a high price is associated with a positive review for older people: such an association is difficult for a linear classifier to learn as neither the 'age' nor 'price' feature is individually correlated with the outcome. Extensions such as *random forests* Ho (1995) (an ensemble of decision trees) remain popular forms of classification.

Multilayer Perceptrons So far, we have only described *linear* classifiers, which assume a simple relationship between between features and predictions. Although we argued in **??** that such limitations can be overcome by careful feature engineering, ideally we might like to learn such feature transformations automatically.

We exclude SVMs and decision trees from the remainder of this book mostly because they have little in common methodologically with the approaches we bulid in later chapters. We briefly introduce multilayer perceptrons in **??** when describing their use within deep learning-based recommendation techniques. As we try to reiterate throughout the book, multilayer perceptrons and various other state-of-the-art models are simply architectural choices that offer alternate ways to optimize the same objectives that we approach through simpler models. Having introduced the overall objectives, and the fundamentals of gradient based optimization approaches, adapting them to alternate architectures is (relatively) straightforward.

3.9 Evaluating Classification Models

So far, when developing classifiers, we have focused on maximizing the alignment between the labels and the model's outputs. E.g. in the case of logistic regression, we want the predicted probability $p_\theta(y_i = 1|x_i)$ to be as close as possible to the label y_i. Implicitly, when doing so, we are trying to maximize the model's *accuracy*:

$$\text{accuracy}(f_\theta|y, X) = \frac{1}{N}\sum_{i=1}^{|y|}\delta(f_\theta(x_i) = y_i), \tag{3.67}$$

where δ is an indicator function,[17] and $f_\theta(x_i)$ is the binarized output of the model (e.g. in the case of logistic regression, $f_\theta(x_i) = \delta(x_i \cdot \theta > 0)$). Equivalently we are minimizing the *error*, i.e.,

$$\text{error}(f_\theta|y, X) = 1 - \text{accuracy}(f_\theta|y, X). \tag{3.68}$$

To motivate the difficulty of properly evaluating classifiers, consider the following classification task. We saw in Figure 3.9 that there was a slight relationship between gender and review length; now, let's see if we can develop a simple classifier that attempts to predict gender based on review length:

```
1 X = [[1, len(d['review/text'])] for d in data]
2 y = [d['user/gender'] == 'Female' for d in data]
3
4 mod = sklearn.linear_model.LogisticRegression()
5 mod.fit(X,y)
6 predictions = mod.predict(X) # Binary vector of predictions
7 correct = predictions == y # Binary vector indicating which
      predictions were correct
8 print(sum(correct) / len(correct))
```

Surprisingly, the classifier produced by this code is 98.5% accurate! This result might seem implausible, but turns out to be an issue with the error measure itself. Counting the number of negative labels in the dataset

reveals that the data is 98.5% male (i.e., 98.5% negative labels). Not only does this reveal that the accuracy is unlikely to be an informative metric in this case, but it reveals that *our goal of optimizing the accuracy* caused us to learn a trivial classifier—the model simply predicts zero everywhere.

The above example demonstrates the problem with naïvely computing (or optimizing) the accuracy. Several situations where we might need more nuanced evaluation measures include:

- Datasets whose labels are highly imbalanced, such as the example above.
- Situations where different types of errors have different associated costs. E.g. failing to detect dangerous luggage is a more severe mistake than an erroneous positive identification.
- When we use classifiers for search or retrieval (as we will often do when developing recommender systems), we often care about the ability of the model to confidently identify a few positive instances (e.g. those surfaced on a results page), and are not interested in its overall accuracy.

Below we will develop error measures designed to handle each of these scenarios.

[17] $\delta(x) = 1$ if the argument x is true, 0 otherwise.

3.9.1 Balanced Metrics for Classification

The basic issue with the example presented above was that we allowed one of the two labels to dominate the classifier's objective. Although in some cases we may justifiably want a classifier that focuses more on the dominant label, in the example above we would likely prefer a solution that had reasonable accuracy *per class*.

To achieve this we need evaluation metrics that consider the two classes (positive and negative, or female and male in our example) separately. To do so, we consider each of the four possible outcomes in terms of our prediction and label:

$$TP = \text{True Positives} = |\{i|y_i \wedge f_\theta(x_i)\}| \tag{3.69}$$

$$TN = \text{True Negatives} = |\{i|\neg y_i \wedge \neg f_\theta(x_i)\}| \tag{3.70}$$

$$FP = \text{False Positives} = |\{i|\neg y_i \wedge f_\theta(x_i)\}| \tag{3.71}$$

$$FN = \text{False Negatives} = |\{i|y_i \wedge \neg f_\theta(x_i)\}|. \tag{3.72}$$

From these, we can define errors (or accuracies) that consider each of the two classes in isolation:[18]

$$TPR = \text{True Positive Rate} = \frac{|\text{true positives}|}{|\text{labeled positive}|} = \frac{TP}{TP + FN} \tag{3.73}$$

$$TNR = \text{True Negative Rate} = \frac{|\text{true negatives}|}{|\text{labeled negative}|} = \frac{TN}{TN + FP} \tag{3.74}$$

$$FPR = \text{False Positive Rate} = \frac{|\text{false positives}|}{|\text{labeled negative}|} = \frac{FP}{FP + TN} \tag{3.75}$$

$$FNR = \text{False Negative Rate} = \frac{|\text{false negatives}|}{|\text{labeled positive}|} = \frac{FN}{FN + TP}. \tag{3.76}$$

Note that it is trivial to optimize any one of these criteria in isolation (e.g. we can achieve a True Positive Rate of 1.0 simply by always predicting positive). As such, we would normally optimize a criterion which considers both positive and negative labels together. One such measure is the *Balanced Error Rate*, which simply takes the average of the False Positive and False Negative rates:

$$\text{BER}(f_\theta|y, X) = \frac{1}{2}(FPR + FNR) = 1 - \frac{1}{2}(TPR + TNR). \tag{3.77}$$

In our motivating example, this now attributes half of the error to the 'Female' (positive) class and half of the error to the 'Male' (negative) class.

Note that an appealing quality of the Balanced Error Rate is that (unlike the

[18] Various other terms exist for these expressions, e.g. the terms *sensitivity*, *recall*, *hit rate*, and *true positive rate* are largely interchangeable.

accuracy), it can no longer be minimized via trivial solutions: always predicting 'True,' or always predicting 'False,' or predicting at random, will result in a BER of 0.5.

3.9.2 Optimizing the Balanced Error Rate

Having argued that the Balanced Error Rate may be preferable to the accuracy if we wish to avoid trivial solutions, we next ask how to train a classifier to avoid producing trivial solutions in the first place.

Intuitively, the degenerate solutions we saw in **??** arose due to an imbalance in our training data (i.e., a high ratio of positive or negative labels). Trivially, we might correct this by *re-sampling* our training data: i.e., sampling either a fraction of our negative negative instance, or sampling negative instances (with replacement) until we have an equal number of positive and negative instances.

While the above is a common and reasonably effective strategy, the same goal can be achieved more directly simply by *weighting* the positive and negative instances. Note that in our objective for logistic regression:

$$\sum_{y_i=1} \log\left(\frac{1}{1+e^{-X_i\cdot\theta}}\right) + \sum_{y_1=0} \log\left(\frac{e^{-X_i\cdot\theta}}{1+e^{-X_I\cdot\theta}}\right), \tag{3.78}$$

the two summations (over $y_i = 1$ and $y_i = 0$) essentially reward the model for correctly predicting positive instances and negative instances. The issue with the above objective is that it will be imbalanced, and one of the two terms can dominate the expression in the event that positive or negative instances are over-represented in our dataset.

To address this, we can normalize the two expressions by the number of samples in the positive and negative classes:

$$\frac{|y|}{2|\{i|y_i=1\}|}\sum_{y_i=1} \log\left(\frac{1}{1+e^{-X_i\cdot\theta}}\right) + \sum_{y_1=0} \log\left(\frac{e^{-X_i\cdot\theta}}{1+\frac{|y|}{2|\{y|y_i=0\}|}e^{-X_I\cdot\theta}}\right). \tag{3.79}$$

By doing so the left- and right-hand expressions have equal importance, such that *all* positively labeled instances have the same importance as *all* negative instances. In other words by normalizing the two expressions correspond to the True Positive *Rate* and True Negative *Rate*, as in Equation (3.77). Note that in addition to normalizing by the number of samples, both sides are multiplied by $\frac{|y|}{2}$; this is not strictly necessary but is done by convention such that the total 'weight' of all instances is still $|y|$.

The above can be accomplished in sklearn with the `class_weight='balanced'` option as follows:

```
1 X = [[1, len(d['review/text'])] for d in data]
2 y = [d['user/gender'] == 'Female' for d in data]
3
4 mod = sklearn.linear_model.LogisticRegression(class_weight='
      balanced')
5 mod.fit(X,y)
```

Note that the same idea can be applied to problems including more than two categories, and that one can choose different weighting schemes, e.g. to assign any desired relative weights to true positives versus true negatives.

3.9.3 Error Metrics for Ranking

Often, the goal of training a classifier is not merely to generate exhaustive sets of 'true' and 'false' instances. For example, if we wanted to identify relevant webpages in response to a query, or to recommend items that a user is likely to purchase, in practice it may not matter whether we can identify *all* relevant webpages or products; rather, we might care more about whether we can surface *some* relevant items among the first page of results returned to a user.

Note that the type of classifiers we've developed so far can straightforwardly be used for *ranking*. That is, in addition to outputting a label ($y_i = x_i \cdot \theta > 0$ in the case of logistic regression), they can also output *confidence scores* (i.e., $x_i \cdot \theta$, or equivalently $p_\theta(y_i = 1|x_i) > 0.5$). Thus, in the context of finding relevant webpages or products above, our goal might be to maximize the number of relevant items returned among the few most confident predictions. Furthermore, we might be interested in how the model's accuracy changes as a function of confidence; e.g. even if the model's accuracy is low overall, is it accurate for the top 1%, 5%, or 10% of most confident predictions?

Precision and Recall *Precision* and *Recall* assess the quality of a set of retrieved results in terms of two related objectives. Informally, *precision* measures the rate at which those items 'retrieved' by the model (i.e., those predicted to have a positive label by the classifier) are in fact labeled positively; *recall* measures what fraction of all positively-labeled items our classifier predicted as having a positive label. For example, in a spam filtering setting (where positively-labeled items are spam e-mails), *precision* would measure how often e-mails marked as spam are in fact spam, whereas *recall* would measure what fraction of all spam was filtered.

Formally precision and recall are defined as follows:

$$\text{Precision} = \frac{|\{\text{relevant items}\} \cap \{\text{retrieved items}\}|}{|\{\text{retrieved items}\}|} \tag{3.80}$$

$$\text{Recall} = \frac{|\{\text{relevant items}\} \cap \{\text{retrieved items}\}|}{|\{\text{relevant items}\}|} \tag{3.81}$$

Alternately it is straightforward to verify that these expressions can be rewritten in terms of the number of true-positives, false-positives, and false-negatives, as in **??**:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.82}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.83}$$

Finally we briefly show how these quantities can be computed for a given predictor (such as the one from **??**):

```
predictions = mod.predict(X) # binary vector of predictions

numerator = sum([(a and b) for (a,b) in zip(predictions,labels)])
nRetrieved = sum(predictions)
nRelevant = sum(y)

precision = numerator / nRetrieved
recall = numerator / nRelevant
```

F_β Score Note that neither precision nor recall are particularly meaningful if reported in isolation. For instance, it is trivial to achieve a recall of 1.0 simply by using a classifier that returns 'true' for every item (in which case, all relevant documents are returned); such a classifier would of course have low precision. Likewise, a precision of close to 1.0 can often be achieved by returning 'true' only for a few items about which we are extremely confident; such a classifier would have low recall.

As such, to evaluate a classifier in terms of precision and recall, we likely want a metric which considers both, or otherwise to place additional constraints on our classifier (as we see below).

The F_β score achieves this by taking a weighted average of the two quantities:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \text{precision} + \text{recall}}. \tag{3.84}$$

In the case of $\beta = 1$ (which is normally called simply the 'F-score'), Equation (3.84) simply computes the harmonic mean of precision and recall, which is low if either of precision or recall are low.

Otherwise, if $\beta \neq 1$, the F_β score reflects a situation where one cares about recall over precision by a factor of β.[19]

There are several situations where one might care about recall more than precision, or vice versa. For instance, considering the motivating examples from the start of this section, in a baggage-handling scenario we would likely care primarily about recall, and would be willing to sacrifice precision to achieve it; or, in a search or recommendation setting, we may be happy retrieving only a few items, so long as some are relevant (i.e., high precision but low recall).

Precision and Recall @ K One of our motivating examples when defining precision and recall considered cases where we may only have a fixed budget of results that can be returned to a user. In particular, we might be interested in evaluating the precision and recall when our classifier returns only its K most confident predictions. To do so, we begin by sorting the labels y_i according to their associated confidence scores (i.e., $x_i \cdot \theta$):

confidence $X_i \cdot \theta$:	$\cdots$	0.49	0.42	0.38	0.16	0.02	-0.02	-0.05	-0.05	-0.08	-0.10	$\cdots$
label y_i:	$\cdots$	True	True	True	True	False	True	False	False	True	True	$\cdots$

(3.85)

Such tuples of confidence scores and labels can be generated as follows (in this case for a logistic regressor as in Section 3.9):

```
1 confidences = mod.decision_function(X) # real vector of
      confidences
2
3 sortedByConfidence = list(zip(confidences,y))
4 sortedByConfidence.sort(reverse=True) # sorted as in
      Equation (3.85)
```

Note that when evaluating our model's K most confident predictions, we are no longer interested in whether the actual scores are greater or less than zero (i.e., whether the classifier would output 'true' or 'false'): we are only interested in the *labels* among the top-K predictions.

The *Precision @ K* and *Recall @ K* now simply measure the precision and recall for a classifier which returns only the K most confident predictions. That is, Precision @ K measures what fraction of the top-K predictions are actually labeled 'true'; Recall @ K measures the fraction of all relevant documents that are returned among the top K. The main difference to note (compared to the definition in Section 3.9.3) is that the number of 'retrieved' documents is always K; that is, the 'retrieved' documents are always the K most confident, whether or not the classifier actually predicts a positive label (i.e., $x_i \cdot \theta > 0$).

[19] Without going into detail, this motivation leads to the specific formulation in Equation (3.84) Van Rijsbergen (1979).

Unlike precision and recall, precision@K and recall@K can be reported in isolation as they cannot be optimized by trivial solutions. Precision@10, for example, is an effective measure of a classifier's ability to return reasonable results among a page of 10 retrieved items.

ROC and Precision/Recall Curves Another holisitc measure of a classifier's performance is to report the *relationship* between precision and recall, or true between and false positives.

For example, the relationship between the number of True Positives and False Positives is known as the *Receiver Operating Characteristic* (ROC). It is so named because of its use in evaluating the performance of radar receiver operators: as an operator's threshold for detection decreases, both their true positive and false positive rates (TPR and FPR) will simultaneously increase; thus we might evaluate a classifier by evaluating the TPR and FPR as we change the classifier's detection threshold.

The *precision recall curve* is developed following a similar line of reasoning: as we lower a classifier's detection threshold, the precision will decrease while the recall increases; thus we might evaluate a classifier by examining the relationship between precision and recall as the threshold changes.

To generate these curves, we sort the predictions of our classifier by confidence (much as we did in Equation (3.85)), which corresponds to gradually considering lower thresholds; at each step, we compute the precision and recall (i.e., we compute the precision and recall @ k for each value of k). Together these values form the precision recall curve:

```
1 for i in range(1,len(sortedByConfidence)+1):
2     retrievedLabels = [x[1] for x in sortedByConfidence[:i]]
3     precision = sum(retrievedLabels) / len(retrievedLabels)
4     recall = sum(retrievedLabels) / sum(y)
5     xPlot.append(recall)
6     yPlot.append(precision)
```

Plotting these x and y coordinates results in the plot in Figure 3.22 (right); the ROC curve can be generated similarly.

We revisit evaluation techniques based on ranking in **??**, when we explore evaluation strategies for recommender systems.

3.10 Implementing the Learning Pipeline

Below we briefly show how to practically apply the process from **??** to select a model based on training, validation, and test samples.

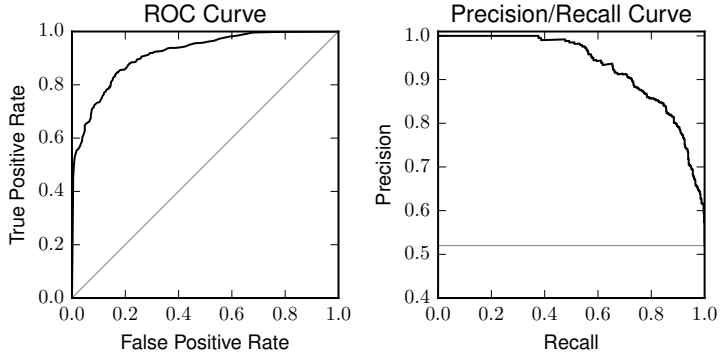


Figure 3.22 Examples of Receiver-Operating Characteristic (left) and Precision Recall (right) curves.

The actual features for this task are based on a sentiment analysis from Chapter 7, in which we predict ratings based on words in a review; for the sake of demonstrating a model pipeline it is useful to consider a problem with high-dimensional features, such that the model is prone to overfitting if not carefully regularized (in this case, we consider 1,000-dimensional features on a dataset with only 5,000 samples).

First, we randomly shuffle the dataset and split it into non-overlapping training, validation, and test portions:

```
random.shuffle(data)
X = [feature(d) for d in data]
y = [d['review/overall'] for d in data]
Ntrain,Nvalid,Ntest = 4000,500,500
Xtrain,ytrain = X[:Ntrain],y[:Ntrain]
Xvalid,yvalid = X[Ntrain:Ntrain+Nvalid],y[Ntrain:Ntrain+Nvalid]
Xtest,ytest = X[Ntrain+Nvalid:],y[Ntrain+Nvalid:]
```

Next, we consider regularization coefficients λ ranging between $\lambda = 10^{-3}$ and $\lambda = 10^4$. For each value, we train a model on training set and evaluate its accuracy on the validation set; during each step, we keep track of the best-performing model in terms of validation accuracy:

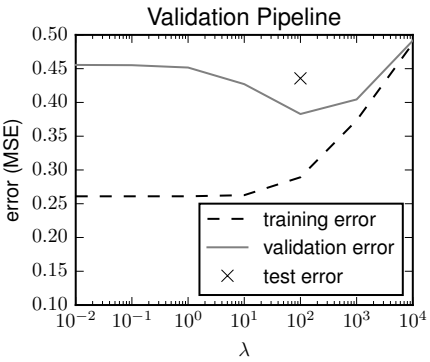


Figure 3.23: Training, validation, and test error on a real pipeline.

```
bestModel = None
bestVal = None
for l in [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]:
    model = sklearn.linear_model.Ridge(l)
    model.fit(Xtrain, ytrain)
    predictValid = model.predict(Xvalid)
    MSEvalid = sum((yvalid - predictValid)**2)/len(yvalid)
    print("l = " + str(l) + ", validation MSE = " + str(MSEvalid))
    if bestVal == None or MSEvalid < bestVal:
        bestVal = MSEvalid
        bestModel = model
```

Finally, we evaluate the best-performing model (in terms of validation performance) on the test set. Note that this is the first and only time we use the test set:

```
predictTest = bestModel.predict(Xtest)
MSEtest = sum((ytest - predictTest)**2)/len(ytest)
```

Figure 3.23 shows the training, validation, and test performance found during the above steps; note the similarity to the hypothetical curves in **??**.

3.11 Exercises

Exercises

3.1

3.2 Using the *GoodReads* data (as in **??**), train a simple predictor that estimates ratings from review length, i.e.,

$$\text{star rating} = \theta_0 + \theta_1 \times (\text{review length in characters}).$$

Compute the values θ_0 and θ_1, and the Mean Squared Error of your predictor.

3.3 Re-train your predictor so as to include a second feature based on the number of comments, i.e.,

$$\text{star rating} = \theta_0 + \theta_1 \times (\text{length}) + \theta_2 \times (\text{number of comments})$$

Compute the coefficients and MSE of the new model. Briefly explain why the coefficient θ_1 in this model is different from the one from Exercise **??**.

3.4 More easy questions

3.5 – Some question about generating precision / recall, and ROC curves

3.6 Show that $\theta_0 = \bar{y}$ is the best possible solution for a trivial predictor (i.e., $y = \theta_0$) in terms of the Mean Squared Error (hint: write down the MSE of this trivial predictor and take its derivative).

3.7 Repeat **??**, but this time show that the best trivial predictor in terms of the Mean *Absolute* Error (Equation (3.15)) is given by taking the *median* value of y.

3.8 In Equation (3.26) we motivated the choice of the MSE by explaining its relationship to a Gaussian error model. Likewise, show that minimizing the MAE is equivalent to maximizing the likelihood if errors follow a Laplace distribution (the Laplace distribution has probability density function $\frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right)$).

3.9 (Hard) In Equation (3.8) we saw how to compute a line of best fit via the pseudoinverse, $\theta = (X^T X)^{-1} X^T y$; show that the parameters that minimize the Mean Squared Error are found by taking the pseudoinverse, i.e., that $\arg\min_\theta \frac{1}{|y|}\sum_{i=1}^{|y|}(x_i \cdot \theta - y_i)^2 = (X^T X)^{-1} X^T y$ (that is, find the stationary point where $\frac{\partial \text{MSE}}{\partial \theta} = 0$).

3.10 (Hard) When minimizing the Mean Squared Error with a linear model as in **??** 3.9, show that the residuals $r_i = (y_i - x_i \cdot \theta)$ have average $\bar{r} = 0$.

3.11 Naïvely, to build a classifier we might simply train a *regressor* by treating labels as numerical quantities (e.g. predicting -1/+1). Perform a simple experiment to demonstrate that this naïve model does not work as well as logistic regression. That is, select a dataset (such as the one you used in Exercise **??**), a few features, a label to predict, and an appropriate classifier evaluation metric, to show that the naïve classifier is outperformed by logistic regression.

– Compare a model that includes θ_0 in the regularizer to one that doesn't

3.11.1 Project 1: Taxicab Tip Prediction

Throughout the chapter, we've seen various strategies for dealing with features of different types. For our first project, let's look into building a prediction pipeline to estimate tip amounts from taxicab trips. For this project you might make use of publicly-available data such as the *NYC Taxi and Limousine Commission Trip Record Data*.[20]

(i) First, conduct *exploratory analysis* of the data. Just as we have done throughout the chapter, plot the relationship between the output variable (tip amount), and various features that you think might be related to this outcome.

(ii) Based on the above analysis, consider what features might be useful for prediction. Consider, for example, features associated with the time of the trip, the start and end location, and the duration/distance of the trip.

(iii) How should the above features be represented or transformed? For example, how can the timestamp be represented to capture variation at the level of time of day, day of week, or even the time of year (**??**); how might you represent the start and end locations? Are there any useful derived features that are useful for prediction, e.g. speed = distance/duration?

(iv) Is it useful to transform the *output* variable (**??**)? For example, rather than predicting the tip *amount*, it may make more sense to predict the tip *percentage*.

(v) Are there any alternatives to formulating the task as a regression problem? For example, you could cast the problem as *classification* by estimating whether a tip will be above or below the median; this may be less sensitive to outliers. Discuss the advantages and disadvantages of various formulations as well as what evaluation metrics you might use.

[20] `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`

4

Recommender Systems

In Chapter 3, when revising regression and classification, our only means of providing *personalized* predictions was to extract features associated with user characteristics (e.g. age, location, gender). The success of such models largely depends on our ability to extract features that adequately explain the variation in the labels we are trying to predict. While effective in a number regression or classification scenarios, when modeling interactions in recommendation scenarios, it is less clear what features are predictive of users' actions, and less likely that those features could be collected in the first place. Consider for example:

- What features would be useful to predict what movies a user would be likely to watch? 'Obvious' features such as user demographics may explain only a small fraction of the variation in interactions and preferences.
- How would you identify the types of features that would be useful for an obscure or unusual domain? For example what features would you collect to recommend baby toys, toaster ovens, or temporary tattoos?
- Are such features likely to be available? In practice, we will often know *nothing* about a user, other than their interaction history.
- How can we use machine learning to make predictions when *no* features are available?

Recommender systems are a fundamental tool to try and make predictions in such scenarios. At their core, recommender systems are concerned with understanding *interactions* between users and items. That is, we are concerned with fitting models that take a user u and item i as input in order to estimate an interaction label y (such as a purchase, click, or rating):

$$f(u, i) \rightarrow y. \tag{4.1}$$

Superficially, solving such a prediction task seems no different than the regres-

sion or classification scenarios we explored in Chapter 3: naïvely we might imagine collecting some appropriate user or item features and applying the techniques we've already developed. However as we study below (**??**), certain characteristics of this scenario render traditional regression and classification approaches ineffective, and demand that we explore new approaches specifically designed to capture the dynamics of interaction data. Specifically:

- Most of the techniques we'll develop in this chapter discard features altogether, and make predictions purely on the basis of historical interactions. This owes partly to the difficulty of collecting useful features, and also to the complex semantics that underlie people's preferences and behavior.
- As such, rather than having parameters associated with features as we did in Chapter 3, the models we'll develop here have parameters associated with individual *users* (or items). This shall be our introduction to the idea of *model-based* personalization, as we discussed in Section 2.4.2.
- To model users (and items) we'll introduce the concept of *latent spaces* in Section 4.5. whereby we automatically discover hidden dimensions that explain the variation in people's opinions—without necessarily knowing exactly what the dimensions correspond to.

Ultimately, the models we develop in this chapter are quite different from those in previous chapters, eschewing explicit features in favor of techniques more closely related to dimensionality reduction and pattern mining.

Roughly speaking, recommender systems operate by finding common patterns and relationships among users and items, so that recommendations for a user can be harvested from others who have similar interactions patterns. We first explore such approaches based on simple set-based similarity heuristics in **??**, before exploring machine-learning based approaches in **??**.

Our discussion of recommender systems will form the basis of many of the models we develop throughout the remainder of this book. Although in this chapter we'll build predictive models based purely on interaction histories, later we'll show how similar models can be extended by incorporating features (Chapter 5) and temporal information (Chapter 6). Later, as we further develop personalized models of text (Chapter 7) and images (Chapter 8) this same notion of modeling users via latent spaces will appear repeatedly.

4.1 Basic Setup and Problem Definition

The typical modality of the data we are trying to model might consist of sequences of historical interactions between users and items, for example, we

might have a collection of movie ratings such as:

$$\begin{array}{llll} (\text{Julian}, & \textit{The Godfather}, & 4, & \text{Jan 4 2019}) \\ (\text{Julian}, & \textit{Pulp Fiction}, & 3, & \text{Jan 6 2019}) \\ (\text{Laura}, & \textit{Seven Samurai}, & 5, & \text{Jan 8 2019}) \\ (\text{Laura}, & \textit{The Godfather}, & 4, & \text{Jan 11 2019}) \\ & \vdots & & \end{array} \tag{4.2}$$

which might further be anonymized in terms of user IDs, item IDs, and sequential timestamps:[1]

$$\begin{array}{llll} (264, & 547, & 4, & 1546588800) \\ (264, & 82, & 3, & 1546761600) \\ (3473, & 231, & 5, & 1546934400) \\ (3473, & 547, & 3, & 1547193600) \\ & \vdots & & \end{array} \tag{4.3}$$

Such a data format is ubiquitous across many popular recommendation datasets and tasks, including (for example) the popular *Netflix Prize* dataset (Section 6.2.1). Such data may include 'side information,' such as reviews, demographic information about the users, or metadata about the movies, but often it may not. In fact, in the simplest form it may not even include ratings or timestamps.

Thus in essence the data we are trying to work with simply describes *interactions* among users and content. Such interactions could describe clicks, purchases, ratings, likes, (etc.). In other settings the interactions could describe social connections among users, or even 'interactions' among compatible clothing items (**??**).

Given interaction data such as that above, we would now like to ask questions such as:

- How will Laura rate Pulp Fiction?
- Given that Laura liked the The Godfather, what other movies will she like?
- What movie is Laura likely to rate next?

Answering these questions seems difficult, as we seemingly know very little about the users and items involved. However we do know, for instance, that both Laura and Julian recently watched The Godfather, and gave it similar ratings; from this we could begin to reason that may exhibit similar preferences with regard to other movies also.

[1] The timestamp shown here is known as the *unix time*, representing the number of seconds since January 1970 (in UTC); such a representation is often useful as it allows straightforward comparison between timestamps.

Reasoning about these types of questions, and modeling these types of interaction data, are the main goals of recommender systems.

How is Recommendation Different from Regression or Classification? In Chapter 3, we saw several techniques that seem like they could already be used to predict outcomes like ratings and purchases. For example, predicting a rating (e.g. of a movie) seems like a traditional regression task, and we can imagine various user and movie features that might be associated with ratings. As such, naïvely we might try to extract user and movie features and fit a linear model of the form

$$\text{rating}(\text{user}, \text{movie}) = \langle \underbrace{\phi(\text{user}, \text{movie})}_{\text{user and movie features}}, \theta \rangle. \tag{4.4}$$

User features might include attributes like the user's age, gender, location, or other demographic features that might be associated with rating patterns; movie features could capture the length, MPAA rating, budget, or presence of certain actors (etc.). Assuming user and movie features can be collected independently, and since the model is linear, this could be rewritten as

$$\text{rating}(\text{user}, \text{movie}) = \langle \underbrace{\phi^{(u)}(\text{user})}_{\text{user features}}, \overbrace{\theta^{(u)}}^{\text{user parameters}} \rangle + \langle \underbrace{\phi^{(i)}(\text{movie})}_{\text{movie (item) features}}, \overbrace{\theta^{(i)}}^{\text{movie parameters}} \rangle. \tag{4.5}$$

When written this way, we can see that the prediction of the rating is the sum of two *independent* predictions: one for the user (say $f(u)$) and one for the item (say $f(i)$). If we were to make recommendations based on these predictions, for example by recommending whichever unseen movie a user would give the highest rating to, i.e.,

$$\underset{i \in \text{unseen movies}}{\arg\max}\ f(u) + f(i), \tag{4.6}$$

our recommendation *for every user* would simply be whichever movie had the highest predicted rating $f(i)$. In other words, every user would simply be recommended movies which had features associated with high ratings.

Critically, such a model does not *personalize* its recommendations to individual users. Even if the model achieved a reasonable Mean Squared Error in terms of predicting ratings, it would not be an effective recommender system.

To overcome this limitation, a model must in some fashion capture *interactions* between users and items, e.g. how compatible is a user with a particular movie. We discussed this idea a little in **??**, but investigate the idea in more detail here.

In summary, modeling *interections* between users and items is the main goal

Figure 4.1 Recommender Systems Compared to Other Types of ML
The main distinguishing feature of recommender systems compared to other types of machine learning is their goal of explicitly modeling *interactions* between users and consumed items based on historical patterns. This feature allows the models to understand which items are *compatible* with which users, and thus to make different recommendations to each user in a personalized way.

of recommender systems and is the main characteristic that differentiates them from other types of machine learning (Figure 4.1).

4.2 Notation and Representation

Basic notation is established in **??**. There are several ways we can represent the interaction data described above. Formally, we might simply describe the dataset as a collection of tuples (u, i, r, t), or $r_{u,i,t} \in \mathbb{R}$ indicating that a user u entered the rating r for item i at time t.

But conceptually it is easier to think about these data in terms of sets or matrices. Set representations will be useful when establishing similarity between users in terms of sets of items they have consumed (or likewise similarity between items in terms of sets of users who have consumed them); matrix representations will be useful when developing models based on the concept of matrix factorization (or dimensionality reduction).

Activities as Sets For our simplest recommendation model, we can describe users in terms of the sets of items they have interacted with, e.g. for a user u:

$$I_u = \text{set of items consumed by } u; \tag{4.7}$$

likewise, we can describe items in terms of the sets of users who have interacted with them:

$$U_i = \text{set of users who consumed item } i \tag{4.8}$$

Activities as Matrices Alternately, we can represent a dataset of user/item interactions via matrices.

$$R = \underbrace{\left[\begin{array}{ccccc} 5 & \cdot & \cdot & 2 & 3 \\ \cdot & 4 & 1 & \cdot & \cdot \\ \cdot & 5 & 5 & 3 & \cdot \\ 5 & \cdot & 4 & \cdot & 4 \\ 1 & 1 & \cdot & 4 & 5 \end{array}\right]}_{\text{items}} \Bigg\} \text{users.} \tag{4.9}$$

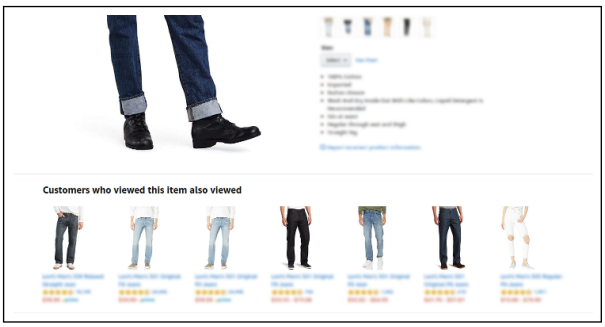


Figure 4.2: An example of neighborhood-based recommendation ('People who viewed X also viewed Y').

Each row of R represents a single user, and each column represents a single item. Then a particular entry r_{ui} indicates the rating a particular user u gave to an item i. Note that the vast majority of entries in such a matrix would typically be missing (most users do not rate most items); indeed, the missing entries are exactly the quantities that we would like to predict.

Naturally, the set and matrix representations can be written in terms of each other, for example, our set representation is equivalent to:

$$I_u = \{i | R_{u,i} \neq 0 \tag{4.10}$$

Both our set and matrix representations of interactions possibly seem limited—neither conveys the timestamps associated with the ratings (or any other side-information), and the set-based representation doesn't even encode users' ratings. Nevertheless they are useful for reasoning about the basic principles behind recommender systems (which will become the building blocks behind more sophisticated approaches), and in fact these simple techniques are among the most widely deployed.

4.3 Neighborhood-based Approaches to Recommendation

Perhaps the simplest (and most ubiquitous) approaches to recommendation are based on some notion of 'similarity' among items. That is, an item is recommended to a user because it is similar to one that they they have recently clicked, liked, or consumed.

'People who viewed X also viewed Y' (or 'people who bought X also bought Y', etc.) features (Figure 4.2) are familiar examples of such similarity-based recommenders. Items are recommended to a user on the basis of how *similar* they are to item the user is currently browsing.

For such a recommender to be effective depends on choosing an appropriate

Figure 4.3 Memory-based and model-based recommender systems
The neighborhood-based recommendation approaches we study in Section 4.3 make recommendations by writing down similarity functions that take as input user and item interaction histories. There is no *model* (i.e., no parameters) associated with each user or item; the original data is used as an input to the similarity functions. Such systems are referred to as *memory-based* recommenders. In contrast, the techniques in **??** learn *representations* of each user and item, such that the original data isn't used directly when making a prediction at test time. Such systems are referred to as *model-based* recommenders.

similarity function. The similarity function in Figure 4.2 is based on view data (presumably clicks); but even then, by what metric should we consider patterns of clicks to be 'similar'? Do we literally just count the number of users who have clicked on both items? Or do we need some kind of normalization? Or should we consider temporal recency?

Appropriately designing such similarity functions, and recommending on the basis of such similarity, is the task of so-called *neighborhood-based* recommender systems—items are recommended due to being in the *neighborhood* of (i.e., similar to) other items.

4.3.1 Defining a Similarity Function

Defining a recommender like the one in Figure 4.2 essentially requires that we define a similarity function among items:

$$\text{sim}(i, j), \tag{4.11}$$

and then given a query item i, recommend a set of items j that maximize the given similarity.

Let's consider a small toy example, with four items, and sets of users (or rather user IDs) who have consumed each:

$$\begin{aligned} U_1 &= \{1, 3, 4, 8, 12, 15, 17, 24, 35, 39, 41, 43\} \\ U_2 &= \{2, 3, 4, 5, 9, 12, 13, 16, 19, 24, 27, 31\} \\ U_3 &= \{4, 5, 9, 12\} \\ U_4 &= \{4, 9\} \end{aligned} \tag{4.12}$$

(recall that in our notation U_1 represents the set of users who have bought item 1). Naïvely, we might assume that a recommender like the one depicted in Figure 4.2 is simply counting the number of users who purchased both items in common. In our set notation this would be:

$$\text{sim}(i, j) = |U_i \cap U_j|. \tag{4.13}$$

Computing some similarities under this model we would find:

$$\text{sim}(1,2) = 4; \quad \text{sim}(2,3) = 3; \quad \text{sim}(3,4) = 2; \quad \text{etc.} \tag{4.14}$$

i.e., we would rate items 1 and 2, or items 2 and 3, as being *more similar* than items 3 and 4.

We should examine whether these relative scores seem reasonable. Items 1 and 2 are *popular* items, which most users did *not* purchase in common; whereas items 3 has *half* of its users in common with item 4. If we were to build recommenders on this basis, we might recommend (for example) a popular album as being highly similar to a popular pair of jeans, simply on the basis that they have many users in common. In general, such a system would tend to identify popular items (such as items 1 and 2 in Equation (4.12)) as being similar. 'Niche' items with fewer associated purchases (such as items 3 and 4 in in Equation (4.12)) would rarely be recommended.

In most cases, this is not the outcome we want; such a system would make generic recommendations of popular items, that likely would not seem specific to the context of a given query item. Presumably, we should improve our similarity function so that it has some appropriate normalization to account for item popularity, as we will see below.

This toy example is intended to demonstrate that 'similarity' is not something easy to define, and that different definitions have implicit assumptions with non-obvious consequences.

4.3.2 Jaccard Similarity

Our first attempt at correcting the above issue is to normalize similarity scores in a way that considers the popularity of each item. The *Jaccard Similarity*, or 'intersection over union,' does so by computing

$$Jaccard(i, j) = \frac{|U_i \cap U_j|}{|U_i \cup U_j|}. \tag{4.15}$$

This similarity function is perhaps best visualized by a Venn diagram such as that in Figure 4.4. The Jaccard similarity takes a value between 0 (when U_i and U_j do not overlap at all, and thus have no intersection) and 1 (when the intersection is equal to the union, i.e., the items were consumed by *exactly* the same set of people).

To demonstrate the Jaccard similarity in action, let's consider computing the similarity among items in terms of past purchases from *Amazon.com*. We'll

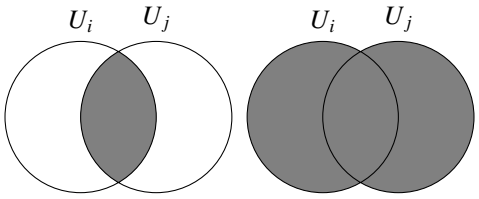


Figure 4.4: The similarity between two items i and j can be computed in terms of the intersection (left) and union (right) between the sets of users U_i and U_j who have consumed each.

consider Amazon's publicly-available dataset of around 900,000 reviews from the *Musical Instrument* category.[2]

We first build some data structures to store the sets of items consumed by each user (or the sets of users who have consumed each item), i.e., I_u and U_i:

```
usersPerItem = defaultdict(set)
itemsPerUser = defaultdict(set)

for d in dataset:
    user,item = d['customer_id'], d['product_id']
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
```

We can also implement the Jaccard similarity straightforwardly:

```
def Jaccard(s1, s2):
    numerator = len(s1.intersection(s2))
    denominator = len(s1.union(s2))
    return numerator / denominator
```

Now, *recommendation* consists of finding the items with the highest Jaccard similarity compared to some given query (i.e., "people who bought X also bought Y"):

```
def mostSimilar(i, K): # Query item i, and number of results K to
    return
    similarities = []
    users = usersPerItem[i] # Users who have purchased i
    for j in usersPerItem: # Compute similarity against each item
        if j == i: continue
        sim = Jaccard(users, usersPerItem[j])
        similarities.append((sim,j))
    similarities.sort(reverse=True) # Sort to find the most
        similar
    return similarities[:K]
```

Finally, let's examine some recommendations, e.g. of the *'AudioQuest LP record clean brush'* (product ID *B0006VMBHI*). The 5 most similar items (i.e.,

[2] Available from `https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt`

`mostSimilar('B0006VMBHI', 5)`) are:

Shure SFG-2 Stylus Tracking Force Gauge
Shure M97xE High-Performance Magnetic Phono Cartridge
ART Pro Audio DJPRE II Phono Turntable Preamplifier
Signstek [...] Long-Playing LP Turntable Stylus Force Scale Gauge Tester
Audio Technica AT120E/T Standard Mount Phono Cartridge

All of the recommended items are also related to record players (which make up only a fraction of items in the category), which seem semantically reasonable given the query.

And in just a few lines of code, and using a (reasonably large) real-world dataset, we have implemented our first recommender system! Our solution is simple (and our implementation is *very* inefficient[3]), but nevertheless quickly produced reasonable recommendations. Perhaps not surprisingly, these simple types of similarity-based recommendations drive many of the most high-profile recommender systems on the Web (see **??**).

4.3.3 Cosine Similarity

The Jaccard similarity captures our intuition about what items ought to be 'similar' to each other, but is only defined if interactions are represented as *sets*. We would like more nuanced similarity measures for data where feedback is associated with each interaction; for example we might not regard two users as 'similar' if both had watched the *Harry Potter* movies, in the event that one of them had liked the series and the other had disliked it.

The *Cosine Similarity* achieves this by representing users' (or items') interaction histories in terms of *vectors* rather than in terms of *sets*. An example is shown in Figure 4.5 in which we have three items (i_1, i_2, and i_3) and two users (u_1 and u_2) who have each interacted with two of them.

In our previous (set) representation we would write $I_{u_1} = \{i_2, i_3\}$ and $I_{u_2} = \{i_1, i_3\}$ to describe the sets of items that u_1 and u_2 have interacted with. In vector representation we simply represent the users as $u_1 = (0, 1, 1)$ and $u_2 = (1, 0, 1)$; note that these vectors are equivalent to rows of our interaction matrix R (Equation (4.9)), i.e., R_{u_1} and R_{u_2}.

The Cosine Similarity (in this case between two users u_1 and u_2) is now defined in terms of the *angle* between the vectors u_1 and u_2. Recall that the

[3] In particular it is not necessary to iterate over *all* items, but rather one can quickly compute a candidate set of only those items that could potentially have a non-zero Jaccard coefficient; see Exercise **??**.

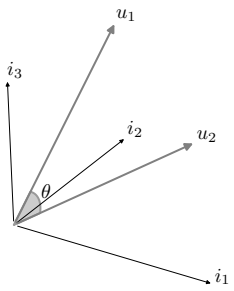


Figure 4.5: The Cosine Similarity is defined in terms of the angle between two vectors, here describing users u_1 and u_2.

angle between two vectors a and b is defined as:

$$\theta = \text{Cos}^{-1}\left(\frac{a \cdot b}{|a| \cdot |b|}\right); \quad \text{or} \quad \text{Cos}(\theta) = \frac{a \cdot b}{|a| \cdot |b|}. \tag{4.16}$$

The angle θ measures the extent to which the two vectors point in the same direction; in the case of interaction data, the angle will range between 0° (if the two users have interacted with exactly the same set of items) and 90° (if the interaction vectors are orthogonal, i.e., if the users have interacted with non-overlapping sets of items). The actual cosine similarity is the cosine of this angle, e.g. between two users u and v is:[4]

$$\text{Cosine Similarity}(u, v) = \frac{I_u \cdot I_v}{|I_u| \cdot |I_v|}. \tag{4.17}$$

For (binary) interaction data the cosine of the angle is now between 1 (when the angle is zero, and the interactions are identical) and 0 (when the interactions are orthogonal).

It is instructive to compare Equations (4.15) and (4.17). In the case of binary interaction data, both expressions take values of 1 when the interactions are identical, and 0 when the interactions are non-overlapping. The numerator $I_u \cdot I_v$ in Equation (4.17) is equivalent to $|I_u \cap I_v|$, as in Equation (4.15). The two differ only in their denominators, both of which are essentially forms of normalization based on the size of the sets I_u and I_v (and both demoninators will take the same value when I_u and I_v are equal).

Of course the cosine similarity is more interesting once we consider numerical interactions, i.e., interactions associated with *feedback*, rather than just binary (0/1) data. For example, consider data where each interaction is associated with a 'thumbs-up' or 'thumbs-down' rating. We might represent this via an interaction matrix R such that $R_{u,i} \in \{-1, 0, 1\}$ (where $-1/1$ indicates

[4] Again it can be straightforwardly defined for *item* similarity by interchanging users and items.

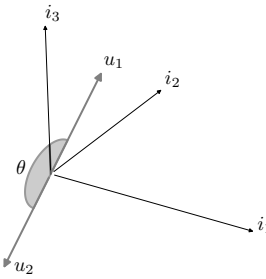


Figure 4.6: The Cosine Similarity for two users who rated the same items, but with opposite sentiment polarity.

thumbs-up/thumbs-down, and 0 indicates an item the user hasn't interacted with).

In this case, the Jaccard similarity would not be well-defined. But the cosine similarity can still be computed on rows (or columns) of R. An example is shown in Figure 4.6. Here, user u_1 has interacted with two items (i_2 and i_3), and liked both; u_2 has interacted with the *same* items, but disliked both. The two user vectors now point in opposite directions, i.e., they have an angle of 180° and a cosine similarity of -1.

With some effort we can adapt our code above for the Jaccard similarity to implement the cosine similarity. Here we use an auxiliary data structure (`ratingDict`) that retrieves ratings for a given user/item pair:

```
1 def Cosine(i1, i2):
2     # Between two items
3     inter = usersPerItem[i1].intersection(usersPerItem[i2])
4     numer = sum([ratingDict[(u,i1)]*ratingDict[(u,i2)] for u in
          inter])
5     norm1 = sum([ratingDict[(u,i1)]**2 for u in usersPerItem[i1]])
6     norm2 = sum([ratingDict[(u,i2)]**2 for u in usersPerItem[i2]])
7     denom = math.sqrt(norm1) * math.sqrt(norm2)
8     if denom == 0: return 0 # If one of the two items has no
          ratings
9     return numer / denom
```

Doing so (for the same query item), the top recommendation remains the same (Shure SFG-2 Stylus Tracking Force Gauge). Among the next few recommendations, there are many ties (i.e., identical cosine similarities); upon inspection these turn out to be items with only a single (overlapping) interaction. In this case such items are preferred by the cosine similarity (and not the Jaccard) since the denominator grows quickly for items with many associated interactions (whereas the union term in Equation (4.15) grows more slowly, assuming many of the interactions overlap).

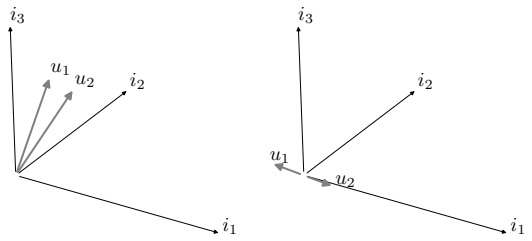


Figure 4.7: Pearson Correlation. Two users have rating vectors that point roughly in the same direction (left); after subtracting the average from each, they point in opposite directions (right).

Which similarity metric is 'better'? In the above example the Jaccard similarity seemed to work 'better' than the Jaccard similarity, but our argument about the difference between the two is somewhat imprecise. Note that this argument largely applies to this specific dataset (or even the specific query item we chose). Ultimately, these similarity measures are essentially *heuristics*; whether one is 'better' than another depends on our own assumptions and intuition about what similarity ought to mean. This is in contrast to the machine learning approaches we saw in previous chapters in which we had a specific objective (i.e., a measure of success) that we were trying to optimize. We'll revisit this question when we examine model-based recommender systems in Section 4.5.

4.3.4 Pearson Similarity

We motivated the Jaccard similarity by considering binary interaction data (i.e., sets), and the cosine similarity by considering polarized interactions like 'thumbs-up's and 'thumbs-down's. But how would these similarity measurements operate on numerical feedback scores such as star ratings (as in Equation (4.9))?

Consider the users represented in **??** (left). Here two users have rated the same items (i_2 and i_3); user u_1 rated them 3 and 5 stars (respectively); user u_2 rated them 5 and 3.

According to the Jaccard similarity (based only on interactions), we would consider the two users to be identical (Jaccard similarity of 1); according to the cosine similarity, we would regard them as being very similar, since the angle between the two vectors is small. However one could argue that these users are polar opposites of each other: if we consider 5 stars to be a *positive* rating and

3 stars is a *negative* rating, then these two users indeed have opposite opinion polarity.

Our definition of the cosine similarity does not account for this interpretation, essentially because it depends on the interactions already having explicitly positive or negative polarity. To might correct this by appropriately normalizing our ratings: if we subtract the average for each user (4 stars for both u_1 and u_2), we find that their ratings are each 1 star above or below their personal average. After doing so, the example becomes very similar to the one from Section 4.3.3 (see Figure 4.7, right).

This is essentially the idea captured by the Pearson Correlation. The *Pearson Correlation Coefficient* is a classical measurement for assessing the relationship between two variables, i.e., whether they trend in the same direction, regardless of scale and constant differences between them. The Pearson Correlation between two vectors x and y is defined as

$$\text{Pearson Correlation}(x, y) = \frac{\sum_{i=1}^{|x|}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{|x|}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{|y|}(y_i - \bar{y})^2}}. \tag{4.18}$$

Compare this definition to that of the cosine similarity in Equation (4.17): the only difference is that from each measurement we subtract the mean ($\bar{x}$ or $\bar{y}$) of the corresponding vector.

When applying this concept to rating data, we should be careful not to regard unobserved ratings (i.e., missing values of $R_{u,i}$ in Equation (4.9)) as zeros—doing so would distort our estimate of the user mean. Thus we might define the similarity between two users u and v (or similarly, items) only in terms of items they have both interacted with:

$$\text{Pearson Similarity}(u, v) = \frac{\sum_{i \in I_u \cap I_v}(R_{u,i} - \bar{R}_u)(R_{v,i} - \bar{R}_v)}{\sqrt{\sum_{i \in I_u \cap I_v}(R_{u,i} - \bar{R}_u)^2}\sqrt{\sum_{i \in I_u \cap I_v}(R_{v,i} - \bar{R}_v)^2}}. \tag{4.19}$$

Note that our choice to define the Pearson Similarity by considering only *shared* items is somewhat arbitrary; we could instead have considered *all* items rated by each user in the denominator. Using our definition (which appears in e.g. Sarwar et al. (2001)), we regard users as maximally similar if they have rated shared items in the same way; if we considered all items rated by each user in the denominator, we would regard users as less similar if they had also rated some different items. Remember that these similarity functions are merely heuristics—neither option should be considered more 'correct,' but rather we should choose the definition that suits our intuition (or generates the most satisfactory results) in a particular situation.

Figure 4.8 Summary of Similarity Measures.
Our comparison of the Jaccard, Cosine, and Pearson similarity can be summarized as follows:

- The *Jaccard Similarity* computes the similarity between *sets*. The basic idea is to find items that have been purchased in common by many users (or similarly, users who have purchased many items in common); the set union is used to normalize the quantity, so that the measure does not overly favor large sets.
- The *Cosine Similarity* instead represents interactions as vectors (basically, rows or columns of our interaction matrix R). Similarity is then computed in terms of the angle between vectors for two items (or users). This definition allows similarity to be computed for numerical interaction data, especially if a polarity (i.e., positive or negative) is associated with each interaction.
- Finally, the *Pearson Similarity* was motivated by the idea that numerical feedback may need to be properly calibrated in order to associate a polarity to each score. For instance, a rating of '3.5' might be positive for one user but negative for another. The Pearson Similarity calibrates this polarity simply by subtracting the average for each item (or user); after this calibration the definition is similar to that of the cosine similarity.

Our code for the cosine similarity above can easily be adapted to implement the Pearson Similarity (noting the details above). Here we have an additional data stucture (`itemAverages`) recording the mean rating for each item:

```
1  def Pearson(i1, i2):
2      # Between two items
3      iBar1,iBar2 = itemAverages[i1], itemAverages[i2]
4      inter = usersPerItem[i1].intersection(usersPerItem[i2])
5      numer = 0
6      denom1 = 0
7      denom2 = 0
8      for u in inter:
9          numer += (ratingDict[(u,i1)] - iBar1)*(ratingDict[(u,i2)] -
               iBar2)
10     for u in inter: # Alternately could sum over usersPerItem[i1
           ][i2]
11         denom1 += (ratingDict[(u,i1)] - iBar1)**2
12         denom2 += (ratingDict[(u,i2)] - iBar2)**2
13     denom = math.sqrt(denom1) * math.sqrt(denom2)
14     if denom == 0: return 0
15     return numer / denom
```

Fitting the Pearson Similarity given the query item from Section 4.3.2 does not produce particularly satisfactory results; using $U_i \cap U_j$ in the denominator of **??** results in many items with a similarity of 1.0 (usually just due to a single overlapping interaction); using U_i and U_j separately in the denominator of **??** generates more meaningful results, though they come from a broad category of items that do not seem closely related.

Possibly these results are unsatisfactory simply because ratings of (e.g.) a record cleaning brush are not due to factors that meaningfully transfer to other items. One likely purchses a record cleaning brush for its utility, rather than because of their personal preference toward such items. If variability in ratings is primarily due to build quality, or effectiveness (for example), then the Pearson Similarity might identify other items with 'similar' build quality or effectiviness, but those may not be semantically similar items. In this particular example, the Jaccard similarity—which defines similarity in terms of *what* was purchased—seems more appropriate than the Pearson Similarity, which defines similarity in terms of preferences.

Again though, possibly this measure is simply not suitable for this dataset or this query item. Let's try again on another dataset, this time from the Amazon *Video Games* category. Given the query *One Piece: Pirate Warriors*, the five most similar items in terms of Pearson Similarity[5] are:

Full Metal Alchemist: The Broken Angel
Monster Rancher 4
FINAL FANTASY X X-2 HD Remaster
BlazBlue: Continuum Shift EXTEND Limited Edition
Killzone 3 (etc.)

These recommendations look more reasonable. In addition to being for similar platforms (e.g. PlayStation) most are reasonably similar in terms of genre and style (e.g. Japanese, based on anime, etc.). Seemingly, for this type of time, features like style and genre better explain variation in ratings, making the Pearson Correlation more effective.

Finally, these similarity measures needn't be used so directly for recommendations as we have done here (i.e., simply retrieving the most similar item given a query). In practice they might simply be subroutines that guide more complex algorithms. For example, to recommend items to a user we might first find similar users, and recommend items that many of those users liked, rather than simply relying on item-to-item similarity directly (see e.g. **??**).

4.3.5 Using Similarity Measurements for Rating Prediction

In Section 4.5, we will contrast the similarity-based recommendation approaches above with machine learning (or 'model-based') approaches which directly seek to predict ratings as accurately as possible.

However these two goals (measuring similarity versus predicting ratings)

[5] Again using U_i and U_j separately in the denominator

are not at odds with each other, and indeed one can use a measure of similarity as a means of predicting ratings.

The essence of such an approach is that the rating a user will give to an item can be estimated from ratings that user has given to *similar* items (again, for some appropriate definition of 'similarity'). One such definition (from Sarwar et al. (2001)) predicts the rating as a weighted sum of other items the user has rated:

$$r(u,i) = \frac{\sum_{j \in I_u} R_{u,j} \cdot \text{Sim}(i,j)}{\sum_{j \in I_u} \text{Sim}(i,j)}, \tag{4.20}$$

where $\text{Sim}(i,j)$ could be any item-to-item similarity function such as those above. Note here that $r(u,i)$ is a prediction whereas $R_{u,i}$ is a historical rating.

The intuition behind the above equation is simply that the most similar items should be the most relevant when predicting future ratings, so the user's past ratings of those items are given the highest weights. Again though this is just a heuristic for predicting ratings and could be defined differently. For example we could write the same definition in terms of user similarity:

$$r(u,i) = \frac{\sum_{v \in U_i} R_{u,j} \cdot \text{Sim}(u,v)}{\sum_{v \in U_i} \text{Sim}(u,v)}, \tag{4.21}$$

or, we, could possibly improve performance slightly by weighting deviations from the average rating, rather than ratings directly:

$$r(u,i) = \bar{R}_i + \frac{\sum_{j \in I_u} (R_{u,j} - \bar{R}_j) \cdot \text{Sim}(i,j)}{\sum_{j \in I_u} \text{Sim}(i,j)}, \tag{4.22}$$

Using our video game data from Section 4.3.4, and following the prediction function from Equation (4.22) (with the Jaccard Similarity as our similarity function), the Mean Squared Error of predicted ratings compared to the true labels is 1.786, compared to 1.838 when always predicting the mean.

Code to implement the rating prediction model of Equation (4.22) is included below. Here we use the Jaccard similarity, though any item-to-item similarity metric could be used in its place:

```
def predictRating(user,item):
  ratings = [] # Collect ratings over which to average
  sims = [] # and similarity scores
  for d in reviewsPerUser[user]:
    j = d['product_id']
    if j == item: continue # Skip the query item
    ratings.append(d['star_rating'] - itemAverages[j])
    sims.append(Jaccard(usersPerItem[item],usersPerItem[j]))
  if (sum(sims) > 0):
    weightedRatings = [(x*y) for x,y in zip(ratings,sims)]
    return itemAverages[item] + sum(weightedRatings) / sum(sims)
  else:
    # User hasn't rated any similar items
    return ratingMean
```

4.4 Case Study: Amazon.com Recommendations

In a 2003 paper Linden et al. (2003), researchers described the techniques underlying *Amazon.com*'s recommendation technology. The paper described systems similar to that in Figure 4.2, e.g. "Customers who bought items in your shopping card also bought."

The first recommendation method the paper describes is based on the Cosine Similarity (Section 4.3.3). Interestingly, cosine similarity is defined between *users*, rather than between items as we did in Section 4.3.2; the goal is then to recommend items that have previously been purchased by similar customers. The paper discusses the issues of scaling this type of similarity computation to the large number of *Amazon* users, and discusses an alternative strategy to *cluster* users based on a user-to-user similarity metric; similar customers to a given user by determining the user's cluster membership (which is cast as a classification problem).

Although Linden et al. (2003) goes into little detail about the specifics of what is implemented on Amazon.com, their work does stress the key point that real-world, large-scale recommenders need not be based on complex models. Rather, primary considerations including building models that are simple but scale well.

Following Linden et al. (2003), a follow up paper was published describing more modern recommendation techinques on Amazon.com Smith and Linden (2017). The paper starts by describing minor modifications to the algorithms described above. For instance, they describe an item-to-item based approach which is more reminiscent of that in Section 4.3.2, and describe how most of the computation for this type of problem can be done offline. Smith and Lin-

Figure 4.9 Memory-based vs. Model-based Approaches
There are a variety of reasons why one might choose a model-based or memory-based approach. We summarize a few of the advatages and disadvantages as follows:

Training and Inference Complexity Model-based approaches often require (expensive) offline training; on the other hand, once trained, recommendations can potentially be retrieved quickly, e.g. by retrieving a nearest neighbor or a maximum inner product in parameter space. In contrast,

Interpretability Often, simple recommendations such as those in Figure 4.2 may be preferable simply because they are easy to explain to a user. In contrast, machine learning-based recommendations may make users uncomfortable due to their 'black box' nature (see **??**).

Accuracy Model-based systems are appealing because they directly optimize a desired error measure. On the other hand, error measures that are tractible may not be those that relate meaningfully to user satisfaction, and may distract from qualitative improvements.

den (2017) also stresses the importance of choosing a good similarity heuristic, and discusses some strategies for doing so. Finally, Smith and Linden (2017) discusses the importance of considering temporal factors when designing recommenders, which is our main focus in Chapter 6.

4.5 Model-based Approaches to Recommendation

In contrast to 'memory-based' recommendation approaches, *model-based* approaches seek to learn parameterized representations of users and items, so that recommendations can be made in terms in terms of the learned parameters. Model-based approaches are typically cast in terms of supervised learning, so that the goal is to predict ratings, purchases, clicks (etc.) as accurately as possible.

4.5.1 The Netflix Prize

In 2006, *Netflix* released a dataset of 100,000,000 movie ratings (across 17,770 users and around 480,000 users). Their dataset took exactly the form described in Section 4.1, i.e., it consisted purely of (user, item, rating, timestamp) tuples. Associated with the dataset was a competition Bennett et al. (2007) to reduce the RMSE (on a test set with withheld ratings) by 10% compared to Netflix's existing solution. The first team to do so would win a $1,000,000 prize.

The history of the competition is itself fascinating. Early leaders joined forces to develop ensemble approaches, and the winning teams were nearly tied in a nail-biting finish. The competition also led to a broader discussion around

Figure 4.10 Lessons from the Netflix Prize
Add text on "Lessons from the Netflix Prize Challenge", maybe in box.

the value of such high-profile competitions, as well as the question of whether narrowly reducing a mean-squared error actually improves recommendations. Finally it also led to a lawsuit against Netflix following de-anonymization of the competition data Narayanan and Shmatikov (2006).[6]

But other than the history surrounding the Netflix Prize, the dataset and high-profile competition spawned a great deal of research on recommender systems in general, and in particular the specific problem of rating prediction. and in particular, solutions based on matrix factorization.

4.5.2 Matrix Factorization

The basic assumption made by model-based recommenders is that there is some underlying *structure* among the interactions we are trying to predict. Put differently, model-based recommender systems are essentially a form of dimensionality reduction.

In simple terms, we assume that users' opinions, or the properties of the items they consume, can be efficently summarized. Do you tend to like action movies (and is this an action movie)? Do you tend to enjoy movies with a high budget, certain actors, or a long runtime? To the extent that purchases, clicks, or ratings can be explained by such factors, the goal of model-based recommendation is to discover them.

Considering data of the form in Section 4.1, this seems a difficult process: we have no knowledge of the necessary *features* that would be needed to discover these important factors (i.e., we don't know which movies are action movies, which movies have a long runtime, etc.). But surprisingly one can still uncover these underlying dimensions. Consider the interaction (e.g. click) data depicted below:

$$R = \begin{bmatrix} 1 & \cdot & 1 & \cdot & \cdot \\ 1 & 1 & 1 & \cdot & \cdot \\ \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & 1 & \cdot \end{bmatrix} \begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{matrix} \qquad (4.23)$$
$$\begin{matrix} i_1 & i_2 & i_3 & i_4 & i_5 \end{matrix}$$

[6] This too is an interesting story, as the competition data, with anonymized user and item IDs, at first glance appears to be sufficiently anonymized.

The matrix appears to decompose roughly into two 'blocks': if we wrote

$$\begin{array}{ll} \gamma_{i_1} = [1,0] & \gamma_{u_1} = [1,0] \\ \gamma_{i_2} = [1,0] & \gamma_{u_2} = [1,0] \\ \gamma_{i_3} = [1,0] \;\; ; & \gamma_{u_3} = [1,0] \;\; , \\ \gamma_{i_4} = [0,1] & \gamma_{u_4} = [0,1] \\ \gamma_{i_5} = [0,1] & \gamma_{u_5} = [0,1] \end{array} \tag{4.24}$$

Then we could summarize the matrix R in Equation (4.23) by writing

$$R_{u,i} = \gamma_u \cdot \gamma_i. \tag{4.25}$$

The two blocks in R might conceivably correspond to some feature in the data, e.g. male and female users who buy men's and women's clothing. If so, the values in Equation (4.24) would correspond to genders, though we discovered these factors automatically simply because they *summarized the structure of the matrix*, rather than needing to rely on observed features.

Matrix Factorization follows this same idea, again by looking for underlying structure that explains observed interactions.

Essentially, our goal is to describe a matrix in terms of lower-dimensional factors, i.e.,

$$\underbrace{\begin{bmatrix} \\ \text{R} \\ \\ \end{bmatrix}}_{N_u \times N_i} = \underbrace{\begin{bmatrix} \\ \gamma_U \\ \\ \end{bmatrix}}_{N_u \times K} \times \underbrace{\begin{bmatrix} \gamma_I^T \end{bmatrix}}_{K \times N_i}. \tag{4.26}$$

That is, we are assuming that the matrix R, of dimension $N_u \times N_i$ (the number of users times the number of items), can be approximated by a 'tall' matrix γ_U and a 'wide' matrix γ_I. Now, a single entry $R_{u,i}$ can be estimated by taking the corresponding row of γ_U and column of γ_I:

$$R_{u,i} = \gamma_u \cdot \gamma_i, \tag{4.27}$$

as in Equation (4.25). $\gamma_u \in \mathbb{R}^K$ is now a latent vector that describes a user, and $\gamma_i \in \mathbb{R}^K$ describes an item.

Examples of such vectors are depicted in Figure 4.11. Intuitively, γ_u might be thought of as describing the 'preferences' of the user u, whereas γ_i describes the 'properties' of item i. Then, the user u will like (e.g. give a high rating or interact with) the item i if their preferences are *compatible* with the item's properties (i.e., they have a high inner product). The latent dimensions, $\gamma_{\cdot,1}$, $\gamma_{\cdot,2}$ (etc.) now describe those latent factors that best explain variability in R. For example, if such a model were trained on the Netflix dataset they might measure the extent to which a movie is a comedy or a romance, or the quality

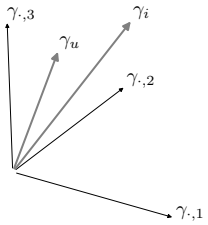


Figure 4.11: Representation of a user u and item i in a latent-factor model.

of its special effects. Again though, these factors are *latent* and are discovered purely so as to maximally explain the observed interactions; models based on the principle of matrix factorization are commonly referred to as *latent factor models*.

4.5.3 Fitting the Latent Factor Model

So far we have described the intuition behind modeling interactions in terms of latent user and item factors, but have not yet described how to fit a model based on this principle. That is, we would like to choose γ_U and γ_I so as to fit the interaction data most closely, e.g. by minimizing some loss such as the Mean Squared Error, following the setting of the Netflix Prize:

$$\arg\min_{\gamma} \frac{1}{|R|} \sum_{(u,i)\in R} (f(u,i) - R_{u,i})^2, \tag{4.28}$$

where $f(u,i)$ is our prediction function $f(u,i) = \gamma_u \cdot \gamma_i$.

Note that this would be straightforward if the matrix R were fully observed (i.e., every user rated every item). In this case, we could rely on the Singular Value Decomposition (SVD). Briefly, this decomposition allows us to factorize a real $M \times N$ matrix according to

$$R = U\Sigma V^T, \tag{4.29}$$

where U, V, and Σ eigenvectors and eigenvalues of $R^T R$ and RR^T. Critically, if we keep just the K eigenvectors corresponding to the largest eigenvalues, this corresponds to the *best possible rank K factorization in terms of the MSE* Golub and Reinsch (1971).

However, the Singular Value Decomposition is defined only for *fully observed* matrices, rather than partially observed interactions as in R. Even if this could be addressed (e.g. via a data imputation strategy on the missing values),

it would not be practical to compute the SVD on a matrix that could potentially have millions of rows and columns.

Instead, we seek a solution based on gradient descent. When minimizing the Mean Squared Error, the solution is similar to the one we saw in Section 3.5. Note that as usual we should be careful to split interactions $(u, i) \in R$ into training, validation, and test sets, and to include a regularizer to avoid overfitting, as we describe below.

User and Item Biases

Before describing the gradient-descent-based solution in detail, we first suggest some steps to improve the solution.

Although a simple solution of the form $r(u, i) = \gamma_u \cdot \gamma_i$ seems to capture the types of interactions we want, it is difficult to regularize. Consider adding a simple ℓ_2 regularizer such as

$$\Omega(\gamma) = \lambda \left(\sum_{u=1}^{|U|} \sum_{k=1}^{K} \gamma_{u,k}^2 + \sum_{i=1}^{|I|} \sum_{k=1}^{K} \gamma_{i,k}^2 \right) \tag{4.30}$$

This regularizer (for large λ) will encourage the parameters to be close to zero; as such the predictions $\gamma_u \cdot \gamma_i$ will also be pushed toward zero, and the system will systematically underpredict ratings.

There are several ways this could be avoided. Trivially, we might simply subtract the mean ($\bar{R}$) from all ratings before training so that they are centered around zero. Alternately, recall as in **??** that we were careful to exclude the intercept term θ_0 from our regularizer. Although our current model lacks such an intercept term, we can straightforwardly add one:

$$r(u, i) = \alpha + \gamma_u \cdot \gamma_i. \tag{4.31}$$

Note that we still regularize as in Equation (4.30).

Although the offset term α corrects the problem of systematically underpredicting ratings, it retains a similar issue at the level of individual users or items. Again, the regularizer pushes γ_u and γ_i toward zero, and therefore pushes predictions toward α. But individual users or items may tend to systematically give much higher (or lower) ratings than α, meaning that our regularizer again encourages us to systematically under (or over) predict.

Again we correct for this by adding additional bias terms, this time at the level of individual users or items:

$$r(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i. \tag{4.32}$$

β_u now encodes the extent to which user u's ratings trend higher or lower than

α, and β_i encodes the extent to which item i tends to receive higher or lower ratings than α.[7]

Adding these bias terms introduces an additional $|U| + |I|$ parameters to the model. Whether these terms should be included in the regularizer Ω is arguable: on the one hand they are similar to offset terms, which we would normally not regularize (for the same reason we do not regularize α); on the other hand, failing to regularize them may lead to overfitting. In practice, the terms may simply be included in the regularizer:

$$\lambda\Omega(\beta,\gamma) = \left(\sum_{u=1}^{|U|}\beta_u^2 + \sum_{k=1}^{K}\gamma_{u,k}^2 + \sum_{i=1}^{|I|}\beta_i^2 + \sum_{k=1}^{K}\gamma_{i,k}^2\right). \tag{4.33}$$

Alternately one could regularize β and γ with different stregths (since we generally expect β to have larger values:

$$\lambda_1\Omega(\beta) + \lambda_2\Omega(\gamma) = \lambda_1\left(\sum_{u=1}^{|U|}\beta_u^2 + \sum_{i=1}^{|I|}\beta_i^2\right) + \lambda_2\left(\sum_{u=1}^{|U|}\sum_{k=1}^{K}\gamma_{u,k}^2 + \sum_{i=1}^{|I|}\sum_{k=1}^{K}\gamma_{i,k}^2\right), \tag{4.34}$$

though doing so is difficult as it results in multiple regularization constants to tune.

Gradient Update Equations

Finally, the objective we wish to minimize (on a training set of interactions $\mathcal{T}$) is

$$\text{obj}(\alpha;\beta;\gamma|\mathcal{T}) = \frac{1}{\mathcal{T}}\sum_{(u,i)\in\mathcal{T}}(r(u,i) - R_{u,i})^2 + \lambda\Omega(\beta,\gamma) \tag{4.35}$$

$$= \frac{1}{\mathcal{T}}\sum_{(u,i)\in\mathcal{T}}(\alpha + \beta_i + \beta_u + \gamma_i + \gamma_u - R_{u,i})^2 + \lambda\Omega(\beta,\gamma). \tag{4.36}$$

Assuming the regularizer takes the form given in Equation (4.33), the partial derivatives (for α, β_u, and $\gamma_{u,k}$) are given by:

$$\frac{\partial\text{obj}}{\partial\alpha} = \sum_{(u,i)\in\mathcal{T}} 2(r(u,i) - R_{u,i})^2$$

$$\frac{\partial\text{obj}}{\partial\beta_u} = \sum_{i\in I_u} 2(r(u,i) - R_{u,i})^2 + 2\beta_u$$

$$\frac{\partial\text{obj}}{\partial\gamma_{u,k}} = \sum_{i\in I_u} 2\gamma_{i,k}(r(u,i) - R_{u,i})^2 + 2\gamma_{u,k}.$$

[7] Note that we are careful not to refer to α as an 'average' rating, and indeed in general once we fit the model $\alpha \neq \bar{R}$, just as the offset θ_0 is not the average $\bar{y}$ in a linear regression model.

Note the change of summation in the last two terms: the derivative for user u is based only on items I_u that they consumed (in the training set). Derivatives for β_i and $\gamma_{u,k}$ can be computed similarly and are left as an exercise (**??**).

Other Considerations for Gradient Descent When we first introduced gradient descent in Section 3.5, we noted some potential issues in terms of local minima, learning rates etc. It is worth revisiting some of those issues in light of the more complex model we are fitting here:

- The problem in Section 4.5.3 is certainly non-convex and has many local minima.[8] Surprisingly though, this problem is not prone to 'spurious' local optima, and if carefully implemented should converge to a global optimum Ge et al. (2016).
- Nevertheless the problem is sensitive to initialization. For example one if multiple columns of γ_U and γ_I are initialized to the same value, they will have identical gradients and will remain in 'lock step' during successive iterations. This can normally be avoided simply by randomized initialization.
- Rather than computing the full gradient as in Section 4.5.3, alternate approaches such as stochastic gradient descent, or alternating least squares, may converge faster or require less memory.[9] See e.g. Bottou (2010); Yu et al. (2012).

4.6 Implicit Feedback and Ranking Models

So far, our discussion of (model-based) recommender systems has focused on predicting real-valued outcomes such as ratings, using objectives based on the Mean Squared Error. That is, we have described model-based recommendation in terms of *regression* approaches.

Just as we developed separate approaches for neighborhood-based recommendation when considering click, purchase, or rating data, here we consider how our regression-based approaches should be adapted to handle binary outcomes (such as clicks and purchases).

Naïvely, we might imagine that we could adapt our regression-based approaches to handle binary outcomes much in the same way we that developed

[8] The proof can roughly be sketched as follows. The objective is smooth, and given any global optimum γ_U, γ_I, any permutation applied to both (i.e., $\gamma_U\pi$ and $\gamma_I\pi$) will result in an *equivalent* local optimum.

[9] Alternating Least Squares notes that the optimization problem in Section 4.5.3 has a closed form if either γ_U or γ_I is fixed; optimization proceeds by alternately fixing one term and optimizing the other.

logistic regression. That is, we could pass the model output (Equation (4.32)) through a sigmoid function, such that positive interactions are associated with high probabilities, and negative interactions are associated with low probabilities.

However, when dealing with click or purchase data, we should consider that items which haven't been clicked or purchased are not necessarily *negative* interactions—in fact items that *haven't* been clicked or purchased are exactly the ones we intend to recommend.

Several techniques have been proposed to handle recommendation in this context. Often this setting is referred to as *one-class* recommendation, as only the 'positive' class (clicks, purchases, listens, etc.) is observed.

Also called implicit feedback recommendation

4.6.1 Instance Re-weighting Schemes

One category of methods for dealing with implicit feedback data attempts to reweight instances as having various 'confidences' of being positive or negative.

Hu et al. (2008) considers cases where positive instances are associated with 'confidence' measures $r_{u,i}$, which could measure e.g. the number of times a user listened to a song; in Hu et al. (2008) the confidence values encode how many times a user has watched a particular program. Negative instances still have $r_{u,i} = 0$, such that the model essentially captures the fact that negative instances are necessarily associated with low confidence, whereas confidence may vary substantially among positive instances.

Ultimately, the goal is still to predict a binary outcome $p_{u,i}$, and the model is trained to predict

$$p_{u,i} = \begin{cases} 1 & \text{if } r_{u,i} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.37}$$

. The form of the predictor is similar to that of Section 4.5.3, i.e., latent user and item factors are use to predict $p_{u,i}$ using a (regularized) Mean Squared Error. The main difference is that the MSE is weighted according to the confidence of each observation:

$$\arg\min_{\gamma} \sum_{(u,i)\in\mathcal{T}} c_{u,i}(p_{u,i} - \gamma_u \cdot \gamma_i)^2 + \lambda\Omega(\gamma), \tag{4.38}$$

where $c_{u,i}$ is a weighting function associated with each observation, which ultimately is a monotone transform of $r_{u,i}$, e.g.

$$c_{u,i} = 1 + \alpha r_{u,i}; \quad \text{or} \quad c_{u,i} = 1 + \alpha \log(1 + r_{u,i}/\epsilon), \tag{4.39}$$

where α and ϵ are tunable hyperparameters. Note that the transform $c_{u,i}$ ensures that negative instances have small but non-zero weight, whereas positive instances receive increasingly higher weight according to their associated confidence.

Pan et al. (2008) approach the problem in a similar way, also fitting a function of the form in Equation (4.38), though their weighting scheme is applied to negative instances. Several schemes are proposed, two of which are as follows:

$$c_{u,i} = \alpha \times |I_u|; \quad \text{or} \quad c_{u,i} = \alpha(m - |U_i|). \tag{4.40}$$

The first (which they call 'user oriented' weighting) suggests that a negative instance should be weighted higher if the corresponding *user* has interacted with many items; the second assumes that a negative instance should be weighted higher if the corresponding *item* has few associated interactions.

Although the schemes above are ultimately simple heuristics for reweighting the model we developed in Section 4.5.3, experiments in Hu et al. (2008) and Pan et al. (2008) show that these scheme outperforms models that try to predict $p_{u,i}$ (or $r_{u,i}$) directly.

4.6.2 Bayesian Personalized Ranking

While the above reweighting schemes demonstrate the importance of treating 'negative' and 'positive' feedback carefully in implicit-feedback settings, they ultimately optimize *regression* objectives, and therefore still seek to assign 'negative' scores to unseen instances.

A potential objection to such an approach is that the unseen instances are exactly the ones we *want* to recommend, and thus we should not encourage a model to assign them a negative score. A weaker assumption might state that while unseen instances should have *lower* scores than positive instances, they need not have negative scores. That is, items which we know a user likes are more positive than unseen items, but unseen items could still be positive.

Rendle et al. (2012) built models based on the above principle by borrowing ideas from ranking. Recall from **??** that the above principle is similar to our goal when adapting classifiers for ranking: while positive (or relevant) items should appear near the top of a ranked list, we are not concerned with their actual scores.

Likewise, the principle behind their method, Bayesian Personalized Ranking, is that we should generate ranked lists of items such that positive items appear first. This is achieved by training a predictor $x_{u,i,j}$ that assigns a score

based on which of the two items (i or j) is preferred (i.e., ranked higher) by u:

$$x_{u,i,j} > 0 \rightarrow u \text{ prefers } i \tag{4.41}$$

$$x_{u,i,j} \leq 0 \rightarrow u \text{ prefers } j. \tag{4.42}$$

Now, if we know that i is a positive and j is a negative (or unseen) item for user u, then a good model should tend to output positive values of $x_{u,i,j}$.

$x_{u,i,j}$ could be any predictor, though the most straightforward option is to define it in terms of *difference* between pairwise predictors, e.g.:

$$x_{u,i,j} = \underbrace{x_{u,i}}_{u\text{'s preference toward } i} - \underbrace{x_{u,j}}_{u\text{'s preference toward } i}. \tag{4.43}$$

For instance, the pairwise compatibility could be defined via a latent factor model, similar to that of **??**:

$$x_{u,i,j} = x_{u,i} - x_{u,j} = \gamma_u \cdot \gamma_i - \gamma_u \cdot \gamma_j. \tag{4.44}$$

Again, note that our goal is not that $\gamma_u \cdot \gamma_i$ should be positive for the positive item, nor thot $\gamma_u \cdot \gamma_j$ should be negative for the unseen item, only that the *difference* is positive, i.e., that the positive item has a higher compatibility score.

We can now define our objective in terms of whether the model correctly outputs positive values $x_{u,i,j}$ given a positive item i and an unseen item j. Ideally, we would like to count how often the model is able to correctly rank positive items higher than unseen items. For a specific user u we have:

$$\text{AUC}(u) = \frac{1}{|I_u||\mathcal{I} \setminus I_u|} \underbrace{\sum_{i \in I_u}}_{\text{positive items for user } u} \underbrace{\sum_{j \in \mathcal{I} \setminus I_u}}_{\text{unseen items for user } u} \delta(x_{u,i,j} > 0). \tag{4.45}$$

The name 'AUC' stands for 'Area Under the ROC Curve;' this measure is equivalent to computing the area under ROC curve as we introduced in **??**. For an entire dataset we average the above across all users:

$$\text{AUC} = \frac{1}{|\mathcal{U}|} \text{AUC}(u) \tag{4.46}$$

Note that this quantity takes a value between 0 and 1, where an AUC of 1 means that the model always ranks positive items higher than unseen items; an AUC of 0.5 means that the model is no better than random.

Optimizing the above presents two issues. First, it is not feasible to consider all (u, i, j) triples; to address this one can randomly sample a fixed number of unseen items j per positive item i.[10]

[10] Furthermore, the sampled items could change during each iteration of training.

Second, the objective in Equation (4.45) is a step function, whose derivative is zero almost everywhere. This is much the same issue we encountered when developing logistic regression in Section 3.8.1; as such we can take the same approach by replacing the step function $\delta(x_{u,i,j})$ with a differentiable surrogate (see Figure 3.21). Using the sigmoid function allows us to interpret $\sigma(x_{u,i,j})$ as a probability:

$$p(u \text{ prefers } i \text{ over } j) = \sigma(x_{u,i,j}). \tag{4.47}$$

From this point, optimization proceeds in much the same way as we developed logistic regression: we use $\sigma(x_{u,i,j})$ to define a (log-)probability of a model given a training set, and subtract a regularizer:

$$\text{obj}^{(\text{BPR})} = \ell(\gamma; \mathcal{T}) - \lambda\Omega(\gamma) \tag{4.48}$$

$$= \sum_{(u,i,j)\in\mathcal{T}} \ln \sigma(\gamma_u \cdot \gamma_i - \gamma_u \cdot \gamma_j) - \lambda\Omega(\gamma). \tag{4.49}$$

Assuming an ℓ_2 regularizer $\Omega(\gamma) = \|\gamma\|_2^2$, we can compute the derivative, for example with respect to $\gamma_{u,k}$:

$$\frac{\partial \text{obj}}{\partial \gamma_{u,k}} = \sum_{(u,i,j)\in\mathcal{T}} \ln \sigma(\gamma_u \cdot \gamma_i - \gamma_u \cdot \gamma_j) - \lambda\|\gamma\|_2^2 \tag{4.50}$$

$$= \frac{\partial}{\partial \gamma_{u,k}} \sum_{(u,i,j)\in\mathcal{T}} -\ln(1 + e^{\gamma_u\cdot\gamma_j - \gamma_u\cdot\gamma_i}) - \lambda\|\gamma\|_2^2 \tag{4.51}$$

$$= \sum_{(i,j)\in I_u} (\gamma_{i,k} - \gamma_{j,k})(1 - \sigma(\gamma_u \cdot \gamma_i - \gamma_u \cdot \gamma_j)) - 2\lambda\gamma_{u,k} \tag{4.52}$$

(we abuse notation slightly so that $(i, j) \in I_u$ includes both positive and unseen items sampled from a user). Derivatives of other terms can be computed similarly. In practice, it is often preferable to use libraries that compute such derivatives automatically, as we explore in **??**.

4.7 Evaluating Recommender Systems

So far, when developing models to predict ratings as in **??**, we have done so by optimizing objectives based on a *sum of squared errors* (or equivalently, a Mean Squared Error). Recall that in Section 3.2.2, we discussed the motivation behind the Mean Squared Error, as well as some potential pitfalls when using it.

In the case of recommender systems, we must be aware of the same pitfalls, but also some different ones. Critically, since since the system is likely used

to provide ranked lists of items to the user, actual prediction of ratings may not be critical so long as desirable items appear near the top of the ranking. Consider, for example, some potential problems with the Mean Squared Error in a recommendation context:

- When using the MSE, mispredicting a 5-star rating as 4-stars incurs a smaller penalty than mispredicting a 3-star rating as 1-star. Arguably, the latter should have a smaller penalty, as it concerns an item which should never have been recommended anyway.
- Similarly, mispredicting ratings of 3 and 3.5 as 4 and 4.5 (respectively) would incur a larger penalty than mispredicting them as 3.5 and 3. However the former preserves the *ordering* of the two items, whereas the latter does not.
- As we saw in **??**, the MSE corresponds to an implicit assumption that errors are normally distributed; critically this assumes that outliers are extremely rare and should be penalized accordingly. In practice, outliers may be common, or alternately errors could be bimodal (or otherwise violate our model assumptions).
- Bellogin et al. (2011) noted the issue of 'popularity bias,' whereby strong performance on popular items can mask performance issues for less-popular ones.[11]

Ultimately, such problems raise the question of whether reductions in Mean Squared Error actually correspond to increased utility of a recommender system. Interestingly, it is reported in Koren (2009) that a carefully implemented model with temporally-evolving bias terms (which we describe in Section 6.2.1) outperformed Netflix's previous solution (*Cinematch*) in terms of the RMSE. Critically, a system without any interaction term can do little more than recommend popular items over time;

Some of the above issues suggest the use of alternative regression metrics, such as the Mean Absolute Error, which (for example) is less sensitive to outliers, as we argued in **??**. Others suggest that perhaps a recommender system should be evaluated less like a regression problem and more like a ranking problem. That is, so long as items matching a user's interests have the *highest* predicted scores, the precise accuracy of our predictions is unimportant.

Arguably, such problems with the Mean Squared Error are driving research toward settings that rely on implicit feedback (clicks, purchases, etc.) rather than ratings; alternately, these settings may be preferable simply because such

[11] Though this is not an issue with the MSE specifically, but rather a general problem of evaluation in imbalanced datasets.

data is more available than explicit feedback such as ratings. More crudely, optimizing clicks or purchases may simply correspond more closely to business goals compared to identifying highly-rated items.

We already saw in **??** one technique to train recommender systems to optimize a ranking loss based on implicit feedback, namely the AUC. Conceptually, the AUC reflects our ability to guess which of two items is 'relevant:' an AUC of 1 means that we always correctly select the correct item, whereas an AUC of 0.5 means our guesses are no better than random.

However, the AUC is but one choice of ranking loss, and was primarily chosen for its convenience when formulating the optimization problem in **??**. As in **??**, when considering cases where recommendations are surfaced to a user via an interface, we may be particularly interested in how the recommender system performs among the top K ranked items.

Below we present a few alternative evaluation functions to measure recommendation performance, most of which are focused on achieving high accuracy among the top-ranked items.

Recall @ K and Precision @ K When we evaluated classifiers in Section 3.9, we motivated the precision and recall @ K useful metrics in the context of evaluating user interfaces, where we have a fixed budget (K) of results that can be returned. Likewise, when recommending items, we might consider whether relevant items (e.g. those that a user eventually interacts with) are given a high ranking.

For convenience, when evaluating recommenders in this setting, it is useful to define a variable $rank_{u,i}$ that specifies in what position an item i was ranked for a particular user u. That is, given a compatibility function $f(u, i)$, and a set of N items that can potentially be recommended (potentially excluding e.g. interactions that already appeared in the training set), then $rank_{u,i} \in \{1 \dots N\}$ such that

$$rank_{u,i} < rank_{u,j} \Leftrightarrow f(u, i) > f(u, j) \tag{4.53}$$

$$rank_{u,i} = rank_{u,j} \Leftrightarrow i = j. \tag{4.54}$$

Now, given a test set of observed interactions I_u we could define the precision@K (for a particular user u) as

$$prec@K(u) = \frac{|\{i \in I_u | rank_{u,i} \leq K\}|}{K}. \tag{4.55}$$

As in **??**, the numerator is the number of relevant items that were retrieved, while the denominator is the number of retrieved items. Now to compute the

precision @ K we simply average over all users:

$$prec@K = \frac{1}{|U|} \sum_{u \in U} prec@K(u). \tag{4.56}$$

Likewise the recall @ K is defined similarly:

$$recall@K = \frac{1}{|U|} \sum_{u \in U} \frac{|\{i \in I_u | rank_{u,i} \leq K\}|}{|I_u|} \tag{4.57}$$

Mean Reciprocal Rank The *Mean Reciprocal Rank* (MRR) is another metric to assess whether a recommender system (or any classifier) ranks positive items highly; unlike the precision and recall @ K this expression does not depend on a particular size of the returned set of items, but rather rewards methods for ranking relevant items near the top of the list.

Traditionally, in search settings, the Mean Reciprocal Rank is defined in terms of the *first* relevant item among a ranked list of retrieved results, though in recommendation settings the metric is typically used by building a test set that consists of only a single relevant item per user, i_u. Then, the Mean Reciprocal Rank is defined in terms of the inverse (reciprocal) of the rank of the relevant item:

$$MRR = \frac{1}{|U|} \sum_{u \in U} \frac{1}{rank_{u,i_u}}. \tag{4.58}$$

This metric will typically rank between 1 (if the relevant item is always ranked in the first position) and 0.5 (if the relevant item appears near the middle, or at a random position in the ranked list).

Cumulative Gain and NDCG The *Cumulative Gain* (and its variants) aim to measure ranking performance in a setting that resembles a user browsing a page of search results: relevant results should be among the top K results, and ideally should be close to the top of the ranked list. The *Cumulative Gain* simply counts the number of relevant items among the top K results:

$$\text{Cumulative Gain}@K = \sum_{i \in \{i | rank_{u,i_u} \leq K\}} y_{u,i}, \tag{4.59}$$

where $y_{u,i}$ is either a binary label (e.g. whether an item was purchase) or a relevance score (such as a rating). That is, the Cumulative Gain will be high if there are many relevant items (or highly rated items) among the top K results.

Ideally, relevant results should appear closer to the top of the list; the *Discounted* Cumulative Gain (DCG) accomplishes this by discounting the reward

for items in lower ranks:

$$\text{DCG@}K = \sum_u \sum_{i \in \{i | rank_{u,i_u} \leq K\}} \frac{y_{u,i}}{\log_2(rank_{u,i} + 1)}. \tag{4.60}$$

This expression is often normalized by comparison against an idealized ranking function to obtain the *Normalized* Discounted Cumulative Gain (NDCG):

$$\text{NDCG@}K = \frac{DCG@K}{IDCG@K} \tag{4.61}$$

where $IDCG@K$ is the 'ideal' discounted cumulative gain, i.e., the discounted cumulative gain that would have been achieved via an optimal ranking function $rank^{\text{opt}}_{u,i}$ (i.e., where labels $y_{u,i}$ are sorted in decreasing order of relevance). This normalization, and the specific choice of logarithmic scaling in Equation (4.60) are theoretically justified in Wang et al. (2013).

Novelty, Diversity, Serendipity, etc. Serendipity: Herlocker et al. (2004)

Coverage: An algorithmic framework for performing collaborative filtering (maybe)?

Finally, it should be noted that there are several other qualities we might desire from a recommender system beyond its immediate utility to users (i.e., its ability to predict the next action). For example, we might be interested in exposing users to diverse viewpoints, ensuring that recommendations aren't biased against certain groups, that users don't get pushed toward 'extreme' content, etc. We'll revisit these issues in Chapter 9, when we consider the broader consequences and ethics of personalized machine learning.

4.7.1 Online Evaluation

See some refs in http://proceedings.mlr.press/v81/ekstrand18b/ekstrand18b.pdf

4.8 Other Aggregation Functions and Deep-learning Based Recommendation

4.8.1 Why the Inner Product?

Briefly, it is worth mentioning that the type of recommendation objective in **????** in which we measure compatibility via an inner product is only one potential choice of compatibility relationship.

While measuring compatibility between a user u and item i in terms of $\gamma_u \cdot \gamma_i$ arises naturally when deriving solutions in terms of matrix factorization

Neural Collaborative Filtering He et al. (2017b) argued that deep learning approaches can be used to learn

Discuss that recent paper by the BPR author Rendle et al. (2020)

Could discuss deep learning papers: e.g. the two youtube papers

4.9 Deep Learning for Recommendation

Some from Zhang et al. (2019)

Zhang et al. (2019) discuss various potential benefits and limitations of deep learning-based recommender systems. Arguably, the main benefit of deep learning-based is the ability to uncover complex, non-linear relationships between user and item representations. For example, previously (**??**) we discussed the relative benefits of the inner-product versus a Euclidean metric when comparing user and item representations. In principle, deep learning approaches could learn more flexible aggregation functions aggregation functions, reducing the need for manual engineering.

Zhang et al. (2019) also point other appealing properties of deep learning-based recommenders, including the effectiveness of deep learning when dealing with structured data such as sequences, images, or text, and the ubiquity of high-level libraries that facilitate straightforward implementation. We discuss such

4.9.1 Multilayer Perceptron-based Recommendation

Multilayer perceptrons (MLPs) are a staple of artificial neural networks, offering a straightforward way to learn non-linear transformations and interactions among features.

Roughly speaking, a 'layer' of a multilayer perceptron transforms a vector of input variables to a (possibly lower dimensional) vector of output variables; typically the output variables are related to the input variables via a linear transformation followed by a non-linear activation, e.g.:

$$f(x) = \sigma(Mx). \tag{4.62}$$

Here x is a vector of input variables, $f(x)$ is a vector of output variables, and M is a learned matrix, such that each term in Mx is a weighted combination of the original features in x. The sigmoid function is applied elementwise, in this case transforming the output variables to lie in the range $(0, 1)$.

While the above is just one *layer* of a multilayer perceptron, several such layers can be 'stacked' in order for the network to learn complex non-linear

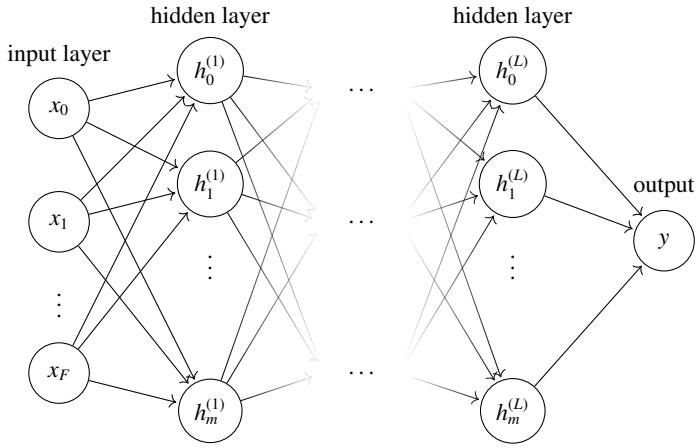


Figure 4.12 Representation of a multi-layer perceptron with L layers. Here for simplicity each layer has the same number (m) of units, and we have only a single output y (so that the inputs and outputs are similar to the regression and classification problems we studied in Chapter 3).

functions. Eventually, the final layer predicts some desired output, e.g. a regression or classification objective. For example, the final layer might simply take a weighted combination of features from the previous layer:

$$f(x) = \sigma(\theta \cdot x), \tag{4.63}$$

i.e., similar to the output of a logistic regressor (for a classification task).

We depict a multilayer perceptron in Figure 4.12. Note that a trivial linear model of the form $y = \theta x$ would be depicted by a similar figure in which the inputs were connected directly to the output.

Ultimately, multilayer perceptrons handle similar modalities of data and problems to those we saw in Chapter 3, i.e., feature vectors as inputs and regression or classification targets as outputs; the main difference compared to the models from Chapter 3 is simply in their ability to learn complex non-linear transformations and relations among features.

Neural Collaborative Filtering He et al. (2017b) attempted to apply the benefits of multilayer perceptrons to latent-factor recommender systems. The essential idea is fairly simple: rather than combining user and item latent factors via an inner product (as in **??**), γ_u and γ_i are combined via a multilayer perceptron to predict the model output (note that both the latent factors and the MLP parameters are learned simultaneously). As we discussed in **??** the inner

product function is only one possible choice when combining user and item preferences and other choices (such as a Euclidean distance) may be more appropriate in other settings; conceptually, the promise of a solution based on a multilayer perceptron is that one can be agnostic to these choices with the expectation that the model will learn the correct aggregation function automatically. While He et al. (2017b) showed this method to be effective in some settings, there has recently been some question as to the value of this type of technique: while MLPs can in principle learn quite general functions, in practice specific functions (like the inner product) are not easily recoverable by such models, meaning that simpler models may still often outperform these more complex approaches. We discuss this issue further in Section 4.9.4.

Deep Factorization Machines Likewise, Guo et al. (2017a) attempted to use multilayer perceptrons to improve the performance of factorization machines. The idea is similar to what we presented above: just as user and item embeddings were combined above, Guo et al. (2017a) combine the embeddings of several terms (users, items, previous items, etc.) via an MLP. In practice, this MLP is trained in parallel to a traditional factorization machine, and the outputs are combined in

He and Chua (2017)

4.9.2 Autoencoder-based Recommendation

Roughly speaking, the role of an *autoencoder* is to learn a low-dimensional representation of some input data that preserves the ability to reconstruct the original (high-dimensional) data from the low-dimensional representation. The basic principle of an autoencoder is depicted in Figure 4.13. Here an input vector x is projected into a lower-dimensional space via a function $g(x)$ (following an approach similar to that followed by a multi-layer perceptron above, which may include several layers). The low-dimensional representation is then mapped back into the original space via $f(g(x))$; the goal is that $f(g(x))$ should match the original data x as closely as possible. In this way $g(x)$ acts as a 'bottleneck,' forcing the model to learn a compressed representation that succinctly captures the meaningful information in x. Several variants of autoencoders exist, for instance *denoising autoencoders* partially corrupt the input in order to learn representations that are robust to noise; *sparse autoencoders* attempt to learn compressed representations that are sparse, etc.

AutoRec Sedhain et al. (2015) adapts the principle of autoencoders to recommendation problems. In their setting the data to be encoded is vector of item

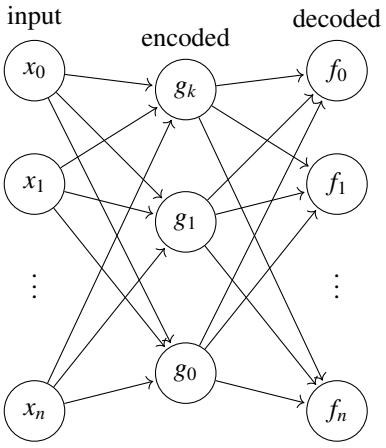


Figure 4.13: Autoencoder. g_i and f_i are shorthand for $g(x)_i$ and $f(g(x))_i$ respectively. Either of the encoding or decoding operations could include multiple layers.

ratings for an item i, or equivalently a column of an interaction matrix $R_{\cdot,i}$. Since $R_{\cdot,i}$ is only partially observed (rather than a dense vector as in a traditional autoencoer), the compressed representation is only responsible for (and gradient updates are only applied to) the observed entries $R_{u,i}$. At inference time the compressed representation can then be used to estimate the entries for unobserved pairs. Sedhain et al. (2015) term this version of the model *AutoRec-I*, since an autoencoder is used to learn compressed item representations; alternately, *AutoRec-U* consists of the same approach applied to user vectors $R_{u,\cdot}$.

Note that AutoRec-I lacks any user parameters (and likewile AutoRec-U lacks item parameters). As such it is a form of *user-free* personalization, that personalizes predictions using a model that aggregates data from the entire user history rather than 'memorizing' user preferences via an explicit parameter. We further explore such notions of user-free personalization when exploring methods based on sequences and text in **??**.

4.9.3 Convolutional and Recurrent Networks

Finally, Zhang et al. (2019) survey various recommender systems based on recurrent networks and convolutional neural networks. Recurrent networks are typically chosen as a way of exploring *sequential* dynamics in user activities; we explore this type of approach in Chapter 6, including deep learning based approaches in **??**. Convolutional neural networks are often used as a means of incorporating representations of rich content (such as images) into recommender systems; we explore this type of approach in Chapter 8.

Table 4.1 *Deep learning-based recommendation techniques.*

Reference	Technique	Description
He et al. (2017b)	neural collaborative filtering	Uses multi-layer perceptrons to learn complex interactions between user and item latent factors.
Guo et al. (2017a)	deep factorization machines	Uses multi-layer perceptrons to train factorization machines.
He and Chua (2017)	neural factorization machines	Trains factorization machines using the *Wide & Deep* approach from **?**.
Sedhain et al. (2015)	AutoRec	Learns compressed representations of item (or user) interaction vectors; the compressed representations can be used to estimate scores associated with unobserved interactions.

Zhang et al. (2019) also discuss potential limitations of deep learning-based approaches, including the challenges involved in interpreting the predictions of deep learning systems, and the difficulty of tuning hyperparameters in systems with many complex, interacting parts. They also highlight the potential lack of interpretability of deep learning-based models (though this is to some extent a challenge with any model based on latent representations), as well as the 'data hungriness' of deep learning approaches. The latter issue arises whenever fitting models with a large number of parameters, and indeed is also a problem when fitting traditional latent factor models as in **??**. On the one hand, this may narrow the conditions under which deep learning-based approaches are effective, e.g. they may underperform in cold-start situations where few interactions are available per user or per item. On the other hand, deep learning approaches may extend the modalities of data that can be incorporated into recommendation approaches (including in cold-start settings), for instance by leveraging text or image data; we explore such approaches in **??** and **??**.

4.9.4 How Effective are Deep Learning-Based Recommenders?

In spite of the proliferation of deep learning-based recommendation approaches, their benefit over simpler, more traditional forms of recommendation is perhaps questionable. Dacrema et al. (2019) conducted a thorough evaluation of several of the predominant deep learning-based recommeder systems (including several of the approaches discussed above), and found that deep learning

approaches were often outperformed by simpler baselines, so long as those baselines were carefully tuned. Most of the models for recommendation we've seen so far involve many tunable factors (e.g. number of factors, regularization schemes, and details of the specific training approaches), as well as choices in terms of dataset selection, pre-processing, etc. that can favor certain models over others. Although the evaluation in Dacrema et al. (2019) was limited to a few specific (but popular) approaches, it raised broader issues of evaluation and benchmarking in recommender systems. Some general points raised include the difficulty in reproducing reported results (while releasing research code is common practice, releasing exact hyperparameter settings or tuning strategies is not), and the proliferation of datasets, metrics, and evaluation protocols that make fair comparison difficult.

Rendle et al. (2020) explored the same issue, focusing on the specific comparison of inner product-based recommendation versus solutions based on multilayer perceptrons. They reiterate the main point from Dacrema et al. (2019) that simpler methods remain competitive so long as they're carefully tuned. They also argue that in spite of the hope that multilayer perceptrons can learn complex, non-linear relationships, that in practice even simple functions (like the inner product) are difficult for such models to reproduce.

Finally, both Dacrema et al. (2019) and Rendle et al. (2020) discuss issues of computational complexity, and whether the marginal benefits of deep learning-based approaches justify the substantial added complexity. Rendle et al. (2020) argue that simpler models may be preferable in production environments, especially when considering the efficiency of item retrieval (as we discuss below).

Note that the above criticisms are not an indictment of deep learning-based approaches in general, but only with respect to their ability to model specific types of interaction data (essentially the same settings discussed in this chapter). In later chapters we'll explore the use of deep learning-based approaches in a variety of other settings, in order to model sequence, text, and image data, with goals ranging from cold-start performance to interpretability.

4.10 Retrieval

Briefly, it is worth discussing one of the fundamental considerations when deploying a recommender system, namely, how can we efficiently retrieve items. Naïvely, having defined a compatibility function $f(u, i)$ between a user and an item, our goal might be to rank (unseen) items according to their compatibility,

i.e.,

$$rec(u) = \arg\max_{i \in \mathcal{I} \setminus I_u} f(u, i). \tag{4.64}$$

Presumably, recommendations must be made rapidly, for use in interactive settings. Given a large vocabulary of items, this procedure is likely to be prohibitively expensive if we were to attempt to enumerate scores for all items $i \in \mathcal{I}$; as such it is worth thinking about what types of relevance functions $f(u, i)$ admit efficient solutions to Equation (4.64).

Euclidean Distance Perhaps the most straightforward function for efficient retrieval is a Euclidean distance function, as in **??**, i.e.,

$$f(u, i) = \|\gamma_u - \gamma_i\|. \tag{4.65}$$

In this case, retrieval can be done efficiently (i.e., $O(\log(|\mathcal{I}|)$ on average) using traditional data structures such as a KD-tree. A KD-tree Bentley (1975) is a data structure that represents K-dimensional points (in this case γ_i for each item) in such a way as to allow efficient retrieval given a query (γ_u); such data structures predate recommender systems and have classical applications in nearest-neighbor retrieval for classification (for example).

Inner Product and Cosine Similarity Bachrach et al. (2014) showed that the same types of data structure can be adapted for other types of relevance function. Fundamentally, they showed that both inner product and cosine distance-based relevance functions can be related to nearest neighbor search (as above) via appropriate transformations:

$$\arg\max_i \|\gamma_u - \gamma_i\| \qquad \text{nearest neighbor (NN)} \tag{4.66}$$

$$\arg\max_i \gamma_u \cdot \gamma_i \qquad \text{maximum inner product (MIP)} \tag{4.67}$$

$$\arg\max_i \frac{\gamma_u \cdot \gamma_i}{\|\gamma_u\|\|\gamma_i\|} \quad \text{maximum cosine similarity (MCS).} \tag{4.68}$$

Doing so allows the same data structures that facilitate nearest-neighbor (Euclidean) search to be used for recommenders based on inner products (as in **??**) or cosine similarity (as in **??**).

Approximate Search and Jaccard Similarity In practice, approximation schemes may also be used for efficient retrieval, such as techniques based on locality-sensitive hashing (whereby 'similar' items are hashed to the same bucket). Versions of locality-sensitive hashing have been proposed to retrieve similar items

based on similarity functions including Euclidean distance Indyk and Motwani (1998), Jaccard Broder (1997), and Cosine similarity Charikar (2002). Bachrach et al. (2014) compares exact techniques for retrieval (as described above) to these types of hashing-based approximations, as well as other exact techniques for recommendation Koenigstein et al. (2012).

4.11 Online Updates

Online-Updating Regularized Kernel Matrix Factorization Models for Large-Scale Recommender Systems

4.12 Recommender Systems in Python with *Surprise* and *Implicit*

Although the types of recommender systems we've seen so far can (with some effort) be implemented 'from scratch' either by computing the gradient expressions as in **??** or by using high-level optimization libraries like *Tensorflow* (we'll explore a Tensorflow implementation in **??**), the recommendation techniques we've covered so far are reasonably well-supported by popular Python libraries.

Here we examine two specific libraries, *Surprise* and *Implicit* for latent-factor recommendation (as in **??**) and Bayesian Personalized Ranking (as in **??**. However these examples serve more to introduce the overall recommendation pipeline, rather than to dive deeply into the specifics of these libraries.

4.12.1 Latent-Factor Models (*Surprise*)

Surprise Hug (2020) is a library that implements various collaborative filtering methods and algorithms based on *explicit feedback* (e.g. ratings). Below we show how to use *Surprise*'s implementation of a latent factor model as in **??** ('SVD' as in **?**).

First we import the model ('SVD'), and utilities to read and split the dataset:

```
1 from surprise import SVD, Reader, Dataset
2 from surprise.model_selection import train_test_split
```

While the library has various routines to read in the dataset, the most straightforward is to read the data from a CSV/TSV file. Here we've processed the *Goodreads* 'fantasy' data to extract just the 'user_id,' 'book_id,' and 'rating'

fields, though this example could be applied to any similar dataset. After reading in the data, we split it into train and test fractions, with 25% of the data withheld for testing:

```
reader = Reader(line_format='user␣item␣rating', sep='\t')
data = Dataset.load_from_file("goodreads_fantasy.tsv", reader=
    reader)
dataTrain, dataTest = train_test_split(data, test_size=.25)
```

Next we fit the model, and collect its predictions on the test set:

```
model = SVD()
model.fit(dataTrain)
predictions = model.test(dataTest)
```

From 'predictions' we can then extract and compare the model's prediction (`p.est`) and the original value (`p.r_ui`), in this case to compute the mean squared error:

```
sse = 0
for p in predictions:
   sse += (p.r_ui - p.est)**2

MSE = sse / len(predictions)
```

4.12.2 Bayesian Personalized Ranking (*Implicit*)

Implicit[12] is a library for recommender systems that operate on implicit feedback datasets, as in **??**. Here we show how to use the library for Bayesian Personalized Ranking, as in **??**.

First, we read in the data. This time, the required data format is a *sparse* matrix describing all user/item interactions. Despite this matrix having hundreds of thousands of rows and columns, only observed interactions are stored:

```
from implicit import bpr

Xiu = scipy.sparse.lil_matrix((nItems, nUsers)) # Sparse matrix;
    initialized after extracting the number of users and items
for d in data:
   itemsToUsers[itemIDs[d['book_id']],userIDs[d['user_id']]] = 1
       # Only storing positive feedback instances

Xui = scipy.sparse.csr_matrix(Xiu.T)
```

Next, we initialize and fit the BPR model:

[12] https://github.com/benfred/implicit

```
model = bpr.BayesianPersonalizedRanking(factors = 5)
model.fit(Xiu)
```

Having fit the model, we can retrieve the user and item factors γ_u and γ_i, as well as recommendations (high $\gamma_u \cdot \gamma_i$) and similar items (high similarity to γ_i):

```
itemFactors = model.item_factors
userFactors = model.user_factors

recommended = model.recommend(0, Xui) # Recommendations for the
    first user
related = model.similar_items(0) # Highly similar to the first
    item (cosine similarity)
```

4.12.3 Implementing a Latent Factor Model in Tensorflow

Following our introduction to tensorflow in **??**, it is now fairly straightforward to implement the more complex models developed in this chapter. Here we fit a latent factor model following **??**.

We start by initializing our model, which takes as parameters the model dimensionality K and the regularization strength λ. Here we define our variables to be fit (α, β_u, β_i, γ_u, γ_i). In practice, appropriate initialization of such variables is important; here alpha is initialized to the mean rating μ while all other parameters are initialized following a normal distribution:

```
class LatentFactorModel(tf.keras.Model):
    def __init__(self, mu, K, lamb):
        super(LatentFactorModel, self).__init__()
        self.alpha = tf.Variable(mu)
        self.betaU = tf.Variable(tf.random.normal([len(userIDs)],
            stddev=0.001))
        self.betaI = tf.Variable(tf.random.normal([len(itemIDs)],
            stddev=0.001))
        self.gammaU = tf.Variable(tf.random.normal([len(userIDs),K
            ],stddev=0.001))
        self.gammaI = tf.Variable(tf.random.normal([len(itemIDs),K
            ],stddev=0.001))
        self.lamb = lamb
```

Next we define our function (a method in the same class) that makes a prediction for a given user/item pair, i.e., $f(u,i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i$ as in **??**:

```
    def predict(self, u, i):
        p = self.alpha + self.betaU[u] + self.betaI[i] +\
            tf.tensordot(self.gammaU[u], self.gammaI[i], 1)
        return p
```

Similarly we define our regularizer as in **??** (which could easily be adapted to include different coefficients for different terms, for example):

```
    def reg(self):
        return self.lamb * tf.reduce_sum(self.betaU**2) +\
                           tf.reduce_sum(self.betaI**2) +\
                           tf.reduce_sum(self.gammaU**2) +\
                           tf.reduce_sum(self.gammaI**2)
```

Finally we define the function to compute the squared error for a single sample, which will be called when computing gradients:

```
    def call(self, u, i, r):
        return (self.predict(u,i) - r)**2
```

4.12.4 Bayesian Personalized Ranking in Tensorflow

Bayesian Personalized Ranking (as in **??**) can be implemented similarly. Again we initialize our model variables (this time only β_i, γ_u, and γ_i) are included:

```
class BPR(tf.keras.Model):
    def __init__(self, K, lamb):
        super(BPR, self).__init__()
        self.betaI = tf.Variable(tf.random.normal([len(itemIDs)],
            stddev=0.001))
        self.gammaU = tf.Variable(tf.random.normal([len(userIDs),K
            ],stddev=0.001))
        self.gammaI = tf.Variable(tf.random.normal([len(itemIDs),K
            ],stddev=0.001))
        self.lamb = lamb
```

Our prediction function estimates the unnormalized score $x_{u,i} = \beta_i + \gamma_u \cdot \gamma_i$:

```
    def predict(self, u, i):
        p = self.betaI[i] + tf.tensordot(self.gammaU[u], self.
            gammaI[i], 1)
        return p
```

Finally we define our loss for a single sample, this time including a user u, and items i and j that they did and did not interact with:

```
    def call(self, u, i, j):
        return -tf.math.log(tf.math.sigmoid(self.predict(u,i) -
            self.predict(u,j)))
```

4.12.5 Efficient Batch-Based Optimization

Although the above implementations are straightforward, they are not particularly efficient and are memory hungry if we attempt to compute the complete

MSE (**??**) or likelihood (**??**) across the entire dataset. Instead, we compute gradients and update parameters in *batches* consisting of a random sample of our data.

First we generate our sample; for a BPR-like model this consists of three lists of user, positive item, negative item triples (u, i, j):

```
sampleU, sampleI, sampleJ = [], [], []
for _ in range(Nsamples):
    u,i,_ = random.choice(interactions) # positive sample
    j = random.choice(items) # negative sample
    while j in itemsPerUser[u]:
        j = random.choice(items)
    sampleU.append(userIDs[u])
    sampleI.append(itemIDs[i])
    sampleJ.append(itemIDs[j])
```

Next we must redefine our score function to operate over a sample rather than a single data point. Note that rather than merely iterating over all points, estimates for all samples in our batch are computed using efficient vector operations:

```
    def score(self, sampleU, sampleI):
        u = tf.convert_to_tensor(sampleU, dtype=tf.int32)
        i = tf.convert_to_tensor(sampleI, dtype=tf.int32)
        beta_i = tf.nn.embedding_lookup(self.betaI, i)
        gamma_u = tf.nn.embedding_lookup(self.gammaU, u)
        gamma_i = tf.nn.embedding_lookup(self.gammaI, i)
        x_ui = beta_i + tf.reduce_sum(tf.multiply(gamma_u, gamma_i)
            , 1)
        return x_ui
```

The 'call' function is similarly modified:

```
    def call(self, sampleU, sampleI, sampleJ):
        x_ui = self.score(sampleU, sampleI)
        x_uj = self.score(sampleU, sampleJ)
        return -tf.reduce_mean(tf.math.log(tf.math.sigmoid(x_ui -
            x_uj)))
```

4.13 Random-Walk Methods

Say something about this somewhere?

4.14 Beyond a 'Black-Box' View of Recommedation

Finally, we should mention that our view of recommendation through the lens of machine learning represents only part of the study of recommender systems. For the most part, we have taken a 'black box' view in which we regard the 'recommender system' as merely a model that predicts user-item interactions (clicks, purchases, ratings, etc.) as accurately as possible.

Although high-fidelity prediction is clearly necessary to build a successful recommender, it is only a small part of the picture. For example, we have not considered broader questions of what makes a recommender system 'usable' or would ultimately drive user satisfaction or engagement. For example, if a user watches *Harry Potter* should they be recommended its sequel, or another movie from the same genre? The former might maximize some naïve metric like click probability, whereas the latter is more likely to generate a suggestion that the user isn't already aware of. But either could be a legitimite goal of building a recommender system: helping a user quickly navigate a user interface by predicting their next interaction is just as important as recommending for novelty or discovery.

Such questions go beyond the black-box supervised learning view of recommendation: they are less questions about how to accurately predict the next action, and more about what we should do with that prediction. At the very least such questions require more nuanced evaluation metrics, if not user studies. While this book largely avoids discussion of user interface design, in Chapter 9 we'll revisit the consequences of how recommender systems are applied, and look at strategies to improve personalized recommendation beyond simply optimizing prediction accuracy.

4.15 History and Emerging Directions

So far we have attempted to construct something of a narrative behind the development of recommender systems: we started with simple 'memory-based' solutions (**??**), followed by 'model-based' approaches such as latent factor models (**??**); later we argued about the benefits of leveraging implicit feedback (clicks, purchases, etc.) rather than relying on ratings (**??**); finally, we began to discuss emerging trends in neural-network based recommendation (**??**). While this narrative reflects current thinking on the topic, the actual history of recommender systems is substantially more complicated: One survey paper Burke (2002) points out that even neural-network based recommender systems have been proposed since the early nineties Jennings and Higuchi (1993).

To a large extent, research on recommender systems has been driven by the release and adoption of large-scale benchmark datasets. High-profile competitions such as the *Netflix Prize* Bennett et al. (2007) have driven widespread interest in recommendation problems: the specific nature of the data (purely based on interactions with no side-information); the choice of metrics used (the mean-squared error); and the specific dynamics of the data itself (e.g. the critical role of temporal dynamics), all show their influence in the models we've explored throughout this chapter. Likewise other datasets and competitions, including industrial datasets (e.g. *Yelp*, *Criteo*) and (e.g. *MovieLens* Harper and Konstan (2015)) have inspired models based on alternate settings and evaluation metrics.

A constant theme in such research is the extent to which new models must be designed to adapt to the specific dynamics of new datasets. As we'll explore in upcoming chapters, recent research on recommender systems has sought to incorporate rich signals in the form of text, temporal and social signals, or images. Such factors serve not only to improve the predictive accuracy of recommendation models, but can also help to make recommendation models more interpretable, and to deal with modalities of data not supported by traditional recommendation approaches. We revisit such *content aware* approaches throughout the remainder of this book, as we begin to develop techniques that make use of social (Chapter 5), temporal (Chapter 6), textual (Chapter 7), and visual (Chapter 8) signals.

Methodologically, recent research on recommender systems has been dominated by deep learning-based approaches, as we discussed a little in **??**. Besides models based on multilayer perceptrons, convolutional neural networks, or autoencoders, a major trend has been to incorporate ideas from natural language processing. Roughly speaking, models of natural language are concerned with modeling the semantics of sequences of discrete tokens (i.e., words or characters), and thus translate naturally to recommendation problems involving sequences of interaction over a discrete set of items. Recommender systems based on natural language models (e.g. *Self-Attention*,*Transformer*,*BERT*, etc.) arguably represent the current state-of-the-art Kang and McAuley (2018); Sun et al. (2019). We'll explore this relationship in Chapters 6 and 7 when developing general-purpose models of sequences and text.

Images, sets, text

A more detailed history of recommender systems is provided in several survey papers Konstan et al. (1998)

Burke (2002)

Bobadilla et al. (2013)

4.16 Exercises

Exercises

4.1 Modify the Jaccard similarity implementation from **??** to implement the cosine similarity. Compare the results of the two implementations on the Amazon data.

4.2 The Jaccard similarity-based recommender we implemented in **??** proved an effective recommender, though our implementation was inefficient. The main source of inefficiency was due to iteration over *all* items. A more efficient implementation might first build smaller a *candidate set* of items, by noting that only those items with at least one user in common with this the query could potentially have non-zero Jaccard similarity. This candidate set can be built by taking all users who have purchased the query i, and taking the union over *other* items they have purchased (other than i), i.e., $\bigcup_{u \in U_i} I_u \setminus \{i\}$. After modifying our implementation from Section 4.3.2 to use this candidate set, confirm that it produces identical recommendations, and compare its running time to the naïve implementation.

4.3 Implement rating prediction models based on the formulas in Equations (4.20) to (4.22). Compare the three in terms of their Mean Squared Error (using either the entire dataset or a random sample).

– Show that the Pearson correlation is related to the R^2 coefficient.

– Some comparison of the various recommender systems error metrics, possibly following the rating prediction experiment above.

4.4 Before implementing the recommender system described in **??**, it is instructive to implement simpler variants in order to understand the gradient descent procedure. Try implementing a *bias-only* model, i.e., one that makes predictions according to $r(u, i) = \alpha + \beta_u + \beta_i$. This can be achieved either by computing derivatives for this simplified model (as we did in **??**), or more simply by discarding the latent-factor terms from the Tensorflow code from **??**. Implement this model and compare its performance (in terms of the MSE) to one which always predicts the average rating. When debugging gradient-descent models, it can be instructive to isolate individual terms (i.e., updating only a single parameter at a time) to determine that each update results in an improvement of the objective.

4.5 In practice the model above does little more than measure the overall quality of items (through β_i) and rating biases of users (β_u), i.e., its predictions are not personalized. Note however that the items with the largest values of β_i may not be those items with the highest average rat-

ings.[13] Using the model you trained above, find the items with the largest values of β_i, and compare those to the items with the highest average ratings.

4.6 Compute the derivatives $\frac{\partial \text{obj}}{\partial \beta_i}$ and $\frac{\partial \text{obj}}{\partial \gamma_{i,k}}$ for the objective from Section 4.5.3.

4.7 (Hard) In **??** we predicted star ratings using a model that optimized the Mean Squared Error. However the ratings we are predicting in many datasets are *integer* valued, e.g. $r_{u,i} \in \{1, 2, 3, 4, 5\}$. In light of this, it might be tempting to round the predictions of our model to the nearest integer. In this exercise, we'll show that this type of rounding is not effective, i.e., that the rounded predictions will tend to have higher MSEs. For this exercise, suppose that when your model outputs a prediction number a, the correct answer is $\lfloor a \rfloor$ with probability $\lceil a \rceil - a$, and $\lceil a \rceil$ with probability $a - \lfloor a \rfloor$ (e.g. if it outputs 4.2 then the correct answer is 4 80% of the time and 5 20% of the time).

4.8 aoeu

4.16.1 Project 2:

In this project we'll build recommender systems to make predictions related to book reviews from *Goodreads*. Goodreads data. Write introduction. Which dataset, how to split, etc.

(i) First, let's consider using the data to predict ratings. Start by implementing simple neighborhood-based approaches as in **??**, followed by a model-based approaches as in **??**. Consider at least the following baselines: (a) the approach of **??**; (b) the same approach, interchanging users and items; (c) a model-based approach using only an offset $f(u, i) = \alpha$; (d) using only an offset and biases $f(u, i) = \alpha + \beta_u + \beta_i$; (e) using latent factors $f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i$.

(ii) Try to thoroughly tune and regularize the latent factor model you developed above. Some factors you might consider include (a) the number of latent factors K; (b) the regularization appoach, for example you might be able to improve performance by using separate regularizers for the bias terms and latent factors, i.e., $\lambda_1 \Omega(\beta)$ and $\lambda_2 \Omega(\gamma)$; (c) other factors, such as learning rates, initialization schemes, etc.

(iii) Second, let's consider using the same data for an implicit feedback task, namely to predict whether a user will *read* a book or not. Before solving this using complex approaches specifically designed for implicit feedback,

[13] Consider why this should be the case. For example, a mediocre item which tends to be rated by 'generous' users (high β_u) could have a high average rating but a low value of β_i.

it's worth considering how much progress can be made using simpler approaches based on *classification*. Start by building a training set consisting of all pairs (u, i) of books i that user u has read; next build an (equally-sized) set of *negative* pairs (u, j) of books the user *hasn't* read (e.g. by sampling randomly). Now, we want to build a feature vector $\phi(u, i)$ that can be used to predict whether a user u has read book i or not. As we discussed in **??**, to build a useful recommender system, we must include features that describe *interactions* between the user and the book. Examples of features you might use could include:

- The popularity of the book (i.e., the number of times item i appears in the training set);
- The predicted rating score for the pair (u, i) that you found in Part (ii);
- The Jaccard similarity (or any other similarity measure) between i and the *most similar* book user u has read (i.e., $\max_{j \in I_u} \text{sim}(i, j)$);
- Likewise, the similarity between the *user* u and the most similar user who has read i;
- Any other similarity measures

5

Content and Structure in Recommender Systems

So far, the systems we've built for personalized recommendation have been based purely on interaction data. We argued in **??** as to why interactions are often sufficient to capture all of the critical signals that we need. However, our argument that

- Only a limited amount of interaction data may be available. Our argument that interaction data is sufficient to capture subtle preference signals applies only in the limit, i.e., when a large number of interactions are available for each user (or item). When few interactions are available (or none, in *cold-start* settings), one must instead rely on user or item *features* to estimate initial preference models.
- Beyond improving performance, incorporating features into recommender systems may be desirable for the sake of model interpretability. For example, we may wish to understand how a user will react to a change in price; doing so effectively may require that price features are appropriately 'baked in' to the model (we study this specific case in **??**).
- User preferences or item properties may not be *stationary*. Even the simple fact that Christmas movies are unlikely to be watched in July cannot be captured simply by adding more latent dimensions. Although not the topic of this chapter, we'll revisit models of such temporal and sequential dynamics in Chapter 6.
- Many settings simply do not follow the setup we developed in Chapter 4. For example, many recommendation scenarios have a social component (dating, bartering, etc.) that must be accounted for in addition to user-to-item compatibility.

In this chapter we'll develop models that help us to adapt to the situations above. First, we'll explore general-purpose strategies to incorporate *features* into recommender systems, starting with Factorization Machines in Sec-

tion 5.1. For the most part, our goal is to study strategies to incorporate simple numerical and categorical features; we develop strategies for the specific cases of temporal, textual, and visual features in **??????**. We are especially interested in how features can be incorporated for the sake of solving so-called *cold-start* problems (**??**), whereby we have little (or no) data associated with new users or items, as must infer an initial model of their preferences.

In this chapter we'll also explore various modaliteis of recommendation that don't exactly follow the basic setup from Chapter 4. We'll explore examples including online dating (**??**), bartering (**??**), social and group recommendation (**??**), among others. In exploring such settings, our goal is not only to explore a few specific applications of interest, but more importantly to understand the overall process of designing and adapting personalized machine learning techniques for situations that don't perfectly align with traditional recommendation settings.

5.1 The Factorization Machine

Factorization Machines Rendle (2010) are a general-purpose approach that seeks to incorporate general features into models that capture pairwise interactions.

In essence, the factorization machine extend the approach behind the latent factor model (**??**). The latent factor model embeds users and items into low dimensional space via γ_u and γ_i, and then models the interaction between them via an inner product; the factorization machine extends this approach to incorporate arbitrary pairwise interactions between users, items, and other features.

The input to the model is a feature matrix X and a target y. In the simplest case, X might simply encode the identity of the user and item via a one-hot encoding, though can be extended to incorporate any additional properties associated with the interaction:

$$\left[\begin{array}{c} 1000000\ldots 000100000\ldots 0001000\ldots 15.95 \\ 0001000\ldots 000000010\ldots 0001000\ldots 12.25 \\ 0100000\ldots 000100000\ldots 0000010\ldots 15.00 \\ 0000100\ldots 010000000\ldots 0010000\ldots 17.50 \\ 1000000\ldots 000000010\ldots 1000000\ldots 19.95 \\ \underbrace{0000100}_{\text{user}}\ldots\underbrace{000010000}_{\text{item}}\ldots\underbrace{0000010}_{\text{weekday}}\underbrace{\ldots 1}_{\text{price}} \end{array}\right] \tag{5.1}$$

The basic idea of the factorization machine is then to model *arbitrary* interactions between features. Each feature dimension is associated latent representation γ_i; the model equation is then defined in terms of all *pairs* of (non-zero)

features:

$$f(\mathbf{x}) = \underbrace{w_0 + \sum_{i=1}^{F} w_i x_i}_{\text{offset and bias terms}} + \underbrace{\sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \gamma_i, \gamma_j \rangle x_i x_j}_{\text{feature interactions}}. \tag{5.2}$$

It is instructive to consider the case where the interaction matrix in Equation (5.1) includes *only* a user and item encoding. In such a case, Equation (5.2) expands to be identical to the latent factor model from **??**, i.e., the only interaction term is $\gamma_u \cdot \gamma_i$ for a user u and item i.

As such, the Factorization Machine can be viewed as a generalization of a latent factor model, that allows for additional types of interactions to be considered. For example, if we include an additional one-hot feature in Equation (5.1) that encodes the *previous* item the user consumed, then the factorization machine will include an expression encoding the compatibility of the next item with the previous one, i.e., the model can learn how *contextually relevant* the previous item is compared to the next one. It is useful to compare this approach to the models we design specifically to handle sequential inputs in **??**. Rendle (2010) discusses such topics, describing the extent to which the general-purpose factorization machine formulation subsumes various approaches designed to handle specific types of features.

Rendle (2010) describes how the model equation of Equation (5.2) can be computed efficiently (and how parameter learning can be done efficiently), by showing that the interaction term can be rewritten as

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \gamma_i, \gamma_j \rangle x_i x_j = \frac{1}{2} \sum_{f=1}^{k} \left(\left(\sum_{i=1}^{n} \gamma_{i,f} x_i \right)^2 - \sum_{i=1}^{n} \gamma_{i,f}^2 x_i^2 \right), \tag{5.3}$$

which allows for computation that is $O(kn)$ (i.e., the dimension of the latent factors multiplied by the feature dimensionality).

5.1.1 Factorization Machines in Python with *FastFM*

As we saw above, the Factorization Machine is a highly flexible, general-purpose technique to incorporate numerical or categorical features into recommender systems. Later in this chapter, and in Chapter 6 we'll explore specific types of dynamics that can be captured via Factorization Machines, but for the moment we'll explore an implementation of a 'vanilla' Factorization Machine via the *FastFM* library Bayer (2016).

First, we read our dataset and construct a mapping from each user to a specific index (from 0 to $|U| - 1$); this index will be used to associate each user and item with a feature dimension in our one-hot encoding (as in **??**):

```
userIDs,itemIDs = {},{}
for d in data:
    u,i = d['user_id'],d['book_id']
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)

nUsers,nItems = len(userIDs),len(itemIDs)
```

Next, we build our matrix of features associated with each interactions. Each feature is simply the concatenation of a (one-hot encoding of) a user ID and an item ID. Note that we use a sparse data structure (`lil_matrix`) to represent the feature matrix. Although we only use user and item IDs here, these feature vectors could straightforwardly be extended to include other features, such as those we explore later in the chapter:

```
X = scipy.sparse.lil_matrix((len(data), nUsers + nItems))
for i in range(len(data)):
    user = userIDs[data[i]['user_id']]
    item = itemIDs[data[i]['book_id']]
    X[i,user] = 1
    X[i,nUsers + item] = 1

y = scipy.array([d['rating'] for d in data])
```

Finally, we split the data into training and test fractions, fit the model, and compute its predictions on the test set:

```
X_train,y_train = X[:2000000],y[:2000000]
X_test,y_test = X[2000000:],y[2000000:]

fm = fastFM.als.FMRegression(n_iter=1000, init_stdev=0.1, rank=2,
    l2_reg_w=0.1, l2_reg_V=0.5)

fm.fit(X_train, y_train)
y_pred = fm.predict(X_test)
```

The model has several tunable parameters, including

5.2 Cold-Start Recommendation

So far, all of the recommendation approaches we have developed have depended on having detailed *interaction histories* associated with users and items. Naturally, we cannot find similar users (as in Section 4.3) to a user who has no purchase history; likewise, we cannot estimate latent parameters γ_u (or even a bias β_u) for a user who has never rated or purchased any items (as in **??**).

As such, we need to develop recommendation approaches that can be use-

ful in so-called *cold-start* scenarios. Depending on the setting, either users or items may be 'cold' (i.e., have zero associated interactions).

We'll investigate two categories of approach to deal with cold-start problems. First, one may attempt to deal with cold-start situations via the use of *side-information* about users or items. Side information could range from product images, to text, or social interactions. In each case, side information gives clues as to the properties of an item, whether by learning user preferences toward observed item features (Section 5.2.1), harvesting weaker signals such as the preferences of a user's friends (**??**), or by using item features to estimate item latent factors (**??**). We explore some of the simpler methods below, but revisit the use of side-information throughout the book. Second, we'll explore methods that directly seek to elicit preferences from new users through surveys (**??**).

5.2.1 Addressing Cold-Start Problems with Side Information

In the absence of historical interaction data associated with users or items, one option is to resort to secondary signals. Park and Chu (2009) considers cold-start settings in the context of movie recommendation. For movies, associated features are available such as the release year, genre, (etc.), which can be encoded (for example) as a one-hot vector. For users, demographic features are used such as a user's age, gender, occupation, or location. These features are captured via user and item feature vectors x_u and z_i for each user u and item i.

Recall that at the beginning of Chapter 4, we discussed the differences between recommendation and regression and argued that recommendation was fundamentally different from simple linear regression on user and item features. Critically, we argued that recommender systems must model the *interaction* between users and items, in order to be able to meaningfully personalize predictions for each user.

In order to capture interactions, Park and Chu (2009) use what is termed a *bilinear* model, to estimate the compatibility between user and item features. The model parameters can then be described via a matrix W, and user-item compatibility can be written as

$$s_{u,i} = x_u W z_i^T = \sum_{a=1}^{|x_u|} \sum_{b=1}^{|z_i|} x_{u,a} z_{i,b} W_{a,b}, \tag{5.4}$$

Here, unlike the linear regression model from (e.g.) **??**, W now encodes how user features should *interact* with item features. That is, a parameter $W_{a,b}$ encodes the extent to which the a^{th} user feature is compatible with the b^{th} item

feature. So, the model can learn (for example) the extent to which users in the 35-50 demographic will respond positively to the teen romance genre.

Both x_u and z_i include a constant feature. These features (or rather the corresponding entries in W) roughly fill the role of bias terms (i.e., α, β_u and β_i in **??**), that is, they allow the model to learn the extent to which users in a certain demographic, or movies of a certain genre, tend to yield higher or lower ratings than others.

The model is trained so that the compatibility $s_{u,i}$ should align with observed interactions (e.g. ratings). Park and Chu (2009) achieves this using a specific type of pairwise loss (i.e., a loss that considers two items at a time, similar to the BPR loss of **??**), though this is an implementation detail that is not critical to the main idea of the method.

Ultimately, the method is evaluated on two movie datasets (*MovieLens* and *EachMovie*). Cold users and items in these datasets are simulated, simply by withholding interactions from a subset of users and items at training time, with interactions from those users being used to evaluate the system at test time. Experiments show that when considering cold users and/or items, the method outperforms alternatives that don't make use of features.

Other Types of Cold-Start Recommendation Obviously there is significant room to improve upon the above system, for example we might desire a model that simultaneously leverages features for cold users, and interactions for non-cold users, and learns to smoothly transition between the two groups.

We'll revisit the topic of cold-start recommendation regularly in later chapters, as well as 'cool-start' settings (where we have only a few interactions per user or item), as we develop systems that operate on features from sequences, text, and images. Whether explicitly designed for cold-start or not, such methods often seek to use side information to circumvent the paucity of available interaction data.

5.2.2 Addressing Cold-Start Problems with Surveys

An alternative to relying on side information in user cold-start settings is simply to directly solicit the preferences of new users once they first interact with the recommender system.

Rashid et al. (2002) investigated strategies for generating initial user 'surveys,' in order to most quickly learn the preference dynamics of new users

Functional Matrix Factorizations for Cold-Start Recommendation (interview based) Zhou et al. (2011)

5.3 Multisided Recommendation

So far, our view of recommendation and personalization has consisted of maximizing some predicted utility for each user, e.g. estimate their ratings or which items they'll interact with. Furthermore, every user has predictions made independently of each other.

Such a setting seems natural when considering contexts such as movie recommendation, but there are a variety of cases where such models are inappropriate. For example, recommendation on an online dating platform would require quite different assumptions. For instance, the problem has symmetries in the sense that the users being recommended are also receiving recommendations—as such, users must be interested in their matches, but the matches must have a reasonable chance of reciprocating. Likewise we must ensure not only that everyone receives recommendations, but that everyone *is recommended to somebody*.

These types of problems are referred to as *multisided* or *multistakeholder* recommendations Abdollahpouri et al. (2017). Such constraints appear in many settings, many of which we will visit throughout the book. In **??** we will look at *group* recommendation, where recommendations must simultaneously satisfy the interests of multiple users in a group. And in **??**, we will consider advertising settings, where we must consider not only user preferences but also the budgets of individual advertisers (which prevent us from recommending the most compatible ads to everyone). Finally we'll revisit the topic in depth in **??**, where we consider issues of fairness, calibration, balance, etc. For example, when recommending movies we might want reasonable coverage of different genres Steck (2018), or when recommending authors, we might want our recommendations not to be too narrow in terms of gender or nationality Ekstrand et al. (2018b).

For now, we'll consider two specific examples of multistakeholder recommendation, namely online dating and bartering (i.e., recommending trading partners).

5.3.1 Online Dating

Pizzato et al. (2010) studied recommendation in the context of online dating. Online dating has several constraints not present in the types of recommendation problems we've seen so far, in particular due to the fact that the users being recommended are the same ones receiving recommendations.

Pizzato et al. (2010) considers the specific objective of *reciprocal communication*, which is partly motivated by a specific mechanism in the data they

study. That is, a recommendation of a user v to a user u should be considered successful only if u messages v *and* v responds to their message.

Actual compatibility scores $f(u, v)$ in Pizzato et al. (2010) are estimated using a fairly simple feature-based strategy that looks for a match between u's preferences and v's attributes (some of which may be matched strictly, e.g. if a user has historically expressed interest in only a certain gender). Following this the *reciprocal compatibility* is simply the harmonic mean of the two compatibility scores:

$$\text{reciprocal compatbility}(u, v) = \frac{2}{f(u, v)^{-1} + f(v, u)^{-1}}. \tag{5.5}$$

The harmonic mean here is preferable to (e.g.) the arithmetic mean as it does not allow either user's preference to 'dominate' the compatibility estimate, i.e., two users are only compatible if *both* have high compatibility scores for each other.

Beyond the notion of reciprocity considered in Pizzato et al. (2010), online dating has certain 'balance' or 'diversity' constraints not yet seen in other problems, for example we cannot identify a user with 'ideal' characteristics and recommend them to everyone (which might be perfectly reasonable for, say, movies); instead, the system only has utility if users both receive, and appear in, recommendations.

5.3.2 Bartering Platforms

Rappaz et al. (2017) considered the problem of generating recommendation systems for *bartering* platforms, i.e., settings in which users exchange goods.

They study several settings in which products are exchanged, including CDs and DVDs, though most of their analysis centers around three datasets, of books (from *bookmooch.com*), beers (from *ratebeer.com*), and video games (from *reddit.com/r/gameswap*).

On each of these websites, users have both a 'wishlist' W_u and a 'give-away' list G_u, i.e., sets of items they wish to give or receive. Given this constraint on the problem, one might think that recommending compatible trades is as simple as identifying compatible pairs. However surprisingly the data reveals that 'eligible' swapping partners are incredibly rare, and the vast majority of logged trades occur between items that were not expressly included in a user's wishlist; thus there is a need to build a system that can model likely trading partners via latent preferences. They also note that users repeatedly trade with the same partners, indicating that there is a social component to trading.

Given the above two factors, the basic model combines a standard latent fac-

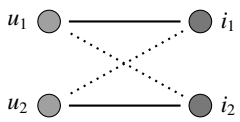


Figure 5.1 Basic idea behind reciprocal interest, from Rappaz et al. (2017); in bartering settings a strong preference from one user compensates for a potentially weaker one from the other.

tor representation with a social term. Given a user u, an item i, and a potential trading partner v, their compatibility is modeled as

$$f(u, v, i) = \gamma_u \cdot \gamma_i + S_{u,v}. \tag{5.6}$$

Here γ_u and γ_i are low-rank factors as in **??**, whereas $S_{u,v}$ is a (potentially full-rank) matrix $S \in \mathbb{R}^{|U|\times|U|}$; although S potentially encodes a large number of parameters, in practice it is very sparse as the number of observed trading partners is limited.

Note that the above model captures only the interest of one user toward another's item; to model reciprocal interest, Rappaz et al. (2017) simply captures the average of interest in both directions (**??**):

$$f(u, i, v, j) = \frac{1}{2}(f(u, v, i) + f(v, u, j)). \tag{5.7}$$

Other aggregation functions besides the arithmetic mean (such as the harmonic mean) can be used, though the arithmetic mean proved the most effective, indicating that a weak preference from one user can be made up for by a strong preference from another. The model also includes a temporal term encoding timepoints when certain users are particularly active and certain items are particularly popular, though we leave discussion of temporal models to **??**.

Ultimately, the method is evaluated in terms of its ability to assign higher scores to observed interactions compared to non-observed ones (i.e., using a BPR-like training and evaluation scheme, as in **??**). The main conclusion of the experiments is that several components are important in bartering settings: reciprocal interest, a social history of trades, as well as temporal 'trends' across users and items.

See other examples from multisided fairness paper

5.4 Music

Thesis: Music recommendation and discovery in the long tail

Features derived from music
Interaction patterns such as skips and completed listens

5.5 Group- and Socially-Aware Recommendation

5.5.1 Socially-Aware Recommendation

Several approaches have sought to incorporate signals from social networks to improve recommendation. The basic idea behind doing so is that social connections will help us to circumvent *sparsity* issues in interaction data. That is, even if a user has only a small number of observed interactions, we can (to some degree) leverage the interactions of their friends, whose opinions they are likely to trust.

Conceptually, the typical approach behind socially-aware recommendation is to use social connections as a form of *regularizer*, which states that a user's preferences should be *similar to* those of their connections in a social network. For example, given a user with few interactions, we might assume that their preferences align with the (average of) their friends; this is a possibly better assumption than the regularizer of **??**, which in practice will essentially discard user latent factors (γ_u) for users with few interactions.

An early attempt to incorporate social networks into recommender systems extended the basic framework of a latent factor model Ma et al. (2008). They looked at data from *Epinions*, which in addition to interaction data in the form of ratings (much like **??**), includes a network of 'trust' and 'distrust' relationships. Unlike a typical social network, these are *signed* relationships, whereby a user explicitly indicates that they 'trust' (1) or distrust (-1) another user. That is, in addition to our interaction matrix we have a (directed) adjacency matrix:

$$A = \underbrace{\left.\left[\begin{array}{ccccc} 1 & \cdot & \cdot & \text{-1} & 1 \\ \cdot & 1 & \text{-1} & \cdot & \cdot \\ \cdot & 1 & 1 & \text{-1} & \cdot \\ 1 & \cdot & 1 & \cdot & 1 \\ \text{-1} & \text{-1} & \cdot & 1 & 1 \end{array}\right]\right\}\text{users.}}_{\text{users}} \tag{5.8}$$

Ultimately though, the distrust relationships are not used in Ma et al. (2008), as it is argued that the semantics of 'distrust' are somewhat more complex than (e.g.) users having different preference dimensions.

Thus, give a rating matrix R, and an adjacency matrix A, we want to predict ratings in R in such a way that A informs us about each user's likely latent

preferences. The basic idea is to make use of a *shared parameter* γ_u for each user. For rating data, γ_u is no different from a user latent factor in a matrix factorization model, i.e., it is combined with an item latent factor and used to predict the rating, in this case via a sigmoid function:[1]

$$r_{u,i} = \sigma(\gamma_u \cdot \gamma_i). \tag{5.9}$$

Next, the parameter γ_u is *re-used* to predict trust relationships in A:

$$a_{u,v} = \sigma(\gamma_u \cdot \gamma'_v). \tag{5.10}$$

The original paper allows A to be a weighted matrix, indicating varying degrees of social trust, but for simplicity we assume here that A contains only trust (1) and not-trust (0) values.

Note that while γ_u is a shared parameter, $\gamma' v$ is not; since the matrix A is directed, γ_u can be thought of as explaining why u trusts others, whereas γ'_v explains why *v is trusted by* others.

In practice we are usually not interested in predicting entries $a_{u,v}$; rather, the trust relationships are additional data that should help us to calibrate γ_u more efficiently. For example, if a user has few observed ratings, γ_u can still be estimated via their trust relationships, giving us an initial estimate of their rating behavior in cold- (or cool-)start settings.

The overall objective then takes the form

$$\underbrace{\sum_{(u,i)\in R} (r_{u,i} - \sigma(\gamma_u \cdot \gamma_i)^2}_{\text{rating prediction error}} + \lambda^{(\text{trust})} \underbrace{\sum_{(u,v)\in A} (a_{u,v} - \sigma(\gamma_u \cdot \gamma'_v)^2}_{\text{trust prediction error}} + \lambda ||\gamma||_2^2, \tag{5.11}$$

where $\lambda^{(\text{trust})}$ trades-off the importance of predicting the trust network. Ultimately, experiments in Ma et al. (2008) show that the trust network helps to predict ratings more accurately than matrix factorization alone. Of course, it should be noted that trust relationships on *epinions* are very closely tied to opinion dimensions, presumably moreso than in other social networks.

Note that the above is essentially a more complex form of cold-start (or 'cool-start') recommendation (as we saw in **??**), in the sense that we are leveraging a form of side-information (social connections) to make up for a paucity of interaction data. In the case of a user who has never rated an item (but has social connections), the system can still reasonably estimate γ_u from the preference dimensions of u's friends.

[1] The specific choice of the sigmoid function is an implementation detail, and ratings are scaled to be in the range $[0, 1]$ to accommodate this choice.

5.5.2 Social Bayesian Personalized Ranking

Above we showed how matrix factorization frameworks can be extended to incorporate signals from social networks. The intuition behind the idea simply stated that users are likely to have preference dimensions aligned with other users who they 'trust.'

Next, we'll see how this idea can be adapted to predict interactions (rather than ratings), by incorporating social connections into the Bayesian Personalized Ranking framework from **??**.

Conceptually, using social links to predict interactions relies on a possibly different assumption than we made above. Whereas our previous intuition above was based on some notion of trust, here we are simply assuming that a user is more likely to interact with items (e.g. to watch movies or read books) if their friends have previously interacted with them.

Zhao et al. (2014) attempted to adapt the assumptions made by Bayesian Personalized Ranking (BPR) to datasets involving social connections. Recall that BPR makes the assumption that a user's compatibility with items they've interacted with ($x_{u,i}$) should be higher than their compatibility with items they haven't interacted with ($x_{u,j}$), which is captured via a sigmoid function:

$$x_{u,i} \geq x_{u,j} \quad \rightarrow \quad \sigma(x_{u,i} - x_{u,j}) \text{ should be maximized} \tag{5.12}$$

(see **??**). To adapt this to settings involving a social network, Zhao et al. (2014) assumes a third type of feedback: for a user u, in adition to *positive* feedback i, and *negative* feedback j (as with BPR), we also have *social* feedback k, which consists of items consumed by u's connections in a social network.

Zhao et al. (2014) tests two assumptions about how this social feedback should be incorporated. The first states that social interactions are weaker than positive interactions, but still stronger than negative interactions, essentially stating that users are somewhat more likely to interact with items their friends have interacted with:

$$\underbrace{x_{u,i}}_{\text{positive}} \geq \underbrace{x_{u,k}}_{\text{social}} ; \quad \underbrace{x_{u,k}}_{\text{social}} \geq \underbrace{x_{u,j}}_{\text{negative}} . \tag{5.13}$$

An alternate hypothesis states the opposite: if our friends have interacted with an item but we haven't, this might instead be a signal that we know about the item, but deliberately chose not to interact wit it; in this instance we drop the second assumption from Equation (5.13) and replace it with a weaker assumption:

$$\underbrace{x_{u,i}}_{\text{positive}} \geq \underbrace{x_{u,k}}_{\text{social}} ; \quad \underbrace{x_{u,i}}_{\text{positive}} \geq \underbrace{x_{u,j}}_{\text{negative}} . \tag{5.14}$$

Table 5.1 *Comparison of socially-aware recommendation techniques.*

Reference	Dataset	Description
Ma et al. (2008)	Social trust data from *epinions*	Trust links help to regularize γ_u, which must simultaneously explain rating and trust factors.
Zhao et al. (2014)	*epinions*, *Ciao*, *Delicious*, *LibraryThing*	Friends' interactions act as additional *implicit signals* for recommendation.
O'connor et al. (2001)	*MovieLens*	Studies the desirable characteristics of good interfaces for group recoommendation.
Amer-Yahia et al. (2009)	*MovieLens*	Designs measures of *group consensus* and proposes recommendation algorithms to maximize them.
Pan and Chen (2013)	*MovieLens*, *Netflix*	Treats group preferences as weak signals to design pairwise sampling strategies for BPR.

Note that neither of these assumptions is 'better' than the other; rather they are simply hypotheses that must be tested by determining which best fits real datasets.

To train the model, a BPR-like objective (**??**) is used, but which now involves two terms. E.g. using the assumption from Equation (5.13):

$$\sum_{(u,i,k)\in\mathcal{T}} \ln \sigma(x_{u,i} - x_{u,k}) + \sum_{(u,k,j)\in\mathcal{T}} \ln \sigma(x_{u,k} - x_{u,j}) + \|\gamma\|_2^2, \tag{5.15}$$

where $\mathcal{T}$ is a training set consisting of positive, negative, and social feedback (i, j, and k) for each user u.

Ultimately, Zhao et al. (2014) shows that both models outperform alternatives that don't leverage social connections. Several datasets besides *epinions* are used, including data from *Ciao*, *Delicious*, and *LibraryThing* (product reviews, social bookmarks, and books). Overall, the assumption from Equation (5.14) slightly outperforms that of Equation (5.13) on all datasets.

5.5.3 Group-Aware Recommendation

Somewhat related to the topic of social recommendation is the idea of *group* recommendation, where recommendations should be made collectively to a

group of users, rather than to an individual. However, unlike social recommendation, where the goal is normally to use social signals to improve the accuracy of recommendations each individual receives, the goal of group-aware recommendation is generally to produce recommendations that mutually satisfy a group of individuals.

Early work on this topic includes *PolyLens* O'connor et al. (2001), though this work was mostly concerned with designing user interfaces for the purpose of group recommendation, rather than using group data to improve recommendation performance. Although focused on interface-building, the work does show the utility of group-based recommenders, which could help users to find items that are mutually compatible with the interests of all group members.

Later, Amer-Yahia et al. (2009) attempted to formalize the notion of group recommendation, by defining useful objectives that define how compatible a set of items is with a group. Given a pre-defined compatibility function $f(u, i)$ (e.g. the output of a latent factor model), a simple attempt to define group compatibilty between a group of users $\mathcal{G}$ and an item i might consist of computing

$$\text{Average compatibilty of } i\text{:} \quad rel(\mathcal{G}, i) = \frac{1}{|\mathcal{G}|} \sum_{u \in \mathcal{G}} f(u, i). \tag{5.16}$$

Alternately, we could define the compatibility as that of the *least* compatible user in the group, known as *least misery*:

$$\text{Least misery:} \quad rel(\mathcal{G}, i) = \min_{u \in \mathcal{G}} f(u, i). \tag{5.17}$$

The latter is preferable in settings where users have constraints, e.g. to avoid recommending a steakhouse to a group of users, some of whom are vegetarian Amer-Yahia et al. (2009).

Amer-Yahia et al. (2009) argues that in addition to maximizing relevance (or minimizing misery), it is also important that the group has some degree of *consensus* about the quality of an item. That is, users in a group should not drastically disagree about $f(u, i)$, separately from their actual scores. Two disagreement functions proposed are average pairwise disagreement:

$$dis(\mathcal{G}, i) = \frac{2}{|\mathcal{G}|(|\mathcal{G}| - 1)} \sum_{(u,v) \in \mathcal{G}, u \neq v} (|f(u, i) - f(v, i)|), \tag{5.18}$$

and disagreement variance:

$$dis(\mathcal{G}, i) = \frac{1}{|\mathcal{G}|} \sum_{u \in \mathcal{G}} \Big(f(u, i) - \underbrace{\frac{1}{|G|} \sum_{v \in \mathcal{G}} f(v, i)}_{\text{average compatibility in } \mathcal{G}} \Big)^2. \tag{5.19}$$

Finally, Amer-Yahia et al. (2009) argues that a group *consensus function* should be a combination of both of these factors, i.e., a relevance function (Equation (5.16) or Equation (5.17)):

$$\mathcal{F}(\mathcal{G}, i) = w_1 \times rel(\mathcal{G}, i) + w_2 \times (1 - dis(\mathcal{G}, i)), \tag{5.20}$$

where w_1 and w_2 trade-off the relative importance of the two terms. *Group recommendation* then consists of finding suitable items (e.g. movies that no user in the group has seen) that maximize a (tuned) group consensus function.

Beyond defining these notions of group consensus, Amer-Yahia et al. (2009) show how to efficiently select items that maximize the above criteria (note the large number of comparisons involved when performing optimization naïvely). They also demonstrate experimentally (via a *Mechanical Turk*-based user study of movie recommendations) that both *relevance* and *disagreement* are simultaneously important to achieve satisfaction within a group.

5.5.4 Group Bayesian Personalized Ranking

Much like Zhao et al. (2014) incorporated social links into Bayesian Personalized Ranking by treating friends' interactions as additional implicit signals, 'Group BPR' (Pan and Chen, 2013) seeks to treat group preferences as a form of implicit signal that can be used within a BPR framework.

While BPR assumes that a user u's compatibility with an observed interaction i is greater than their compatibility with an unobserved interaction j (i.e., $x_{u,i} > x_{u,j}$, as in **??**), Pan and Chen (2013) assumes that *group* preference acts as a similar form of implicit feedback. As with Social BPR (**??**), the goal Group BPR is essentially to leverage weak signals from related users as a way of harvesting implicit pairwise preference feedback.

Specifically, if a group of users $\mathcal{G}$ has interacted with some item i, their *mutual preference* toward the item, defined as

$$x_{\mathcal{G},i} = \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} x_{g,i} \tag{5.21}$$

is assumed to be greater than a user u's preference toward an unseen item j (i.e., $x_{u,j}$). This notion of group preference is combined with the pairwise preference model from **??**, resulting in a preference model of the form:

$$\rho x_{\mathcal{G},i} + (1 - \rho) x_{u,i} > x_{u,j}, \tag{5.22}$$

where ρ is a hyperparameter controlling the relative importance of individual versus group preference.

Interestingly, the training data used for evaluation by Pan and Chen (2013)

Figure 5.2 When is side-information useful for recommendation?

- Most of the settings we've considered in this chapter have essentially been forms of *cold-start*. In other words, features features compensate for a lack of historical interaction data (from either the user or item side).
- Features are *unlikely* to be particularly useful in 'warmer' settings: even if a feature (price, brand, genre) explains variance, high-variance dimensions will already be captured by latent terms (i.e., γ_u and γ_i).
- An exception to the above is features which are not *static*. We study price variability in this chapter, and temporal dynamics more broadly in Chapter 6. Latent terms cannot will struggle to capture this type of variability unless it is explicitly modeled.
- Another important use of features is for model *interpretability*: even features that yield a modest improvement in predictive performance may help us to understand the underlying dynamics of a particular problem better than we can from latent representations. We discuss such notions of interpretability in Chapter 7.

consists of 'standard' interaction datasets that do not contain explicit groups; rather, groups are sampled randomly among users who have consumed a particular item. As such, Group BPR is perhaps best thought of as a different means of leveraging explicit and implicit signals in implicit feedback settings, rather than as a group method as such.

Experiments in **?** show that sampling pairwise preferences as in Equation (5.22) can improve performance over standard BPR on various benchmark datasets.

5.6 Price Dynamics in Recommender Systems

In spite of the obvious impact price has on user decisions, there is surprisingly little work that seeks to incorporate price features into personalized predictive models. Partly this owes to the lack of suitable datasets: the vast majority of the datasets we've study so far (concerned with movies, books, restaurants, etc.) include few useful features from which to build a model of price.

Even in datasets that do include a price variable, it is not obvious how this variable should be incorporated in to the types of algorithms we've seen so far. Naïvely one might think that price might be incorporated into (e.g.) a factorization machine (**??**) much like any other feature. While such a feature might help in cold-start settings, it is unlikely to improve predictive performance in general: to the extent that price explains significant variability in user preferences or item properties, it will already be captured by user or item latent factors. This form of ineffectiveness often comes as a surprise when implementing content-aware model: the features that explain the most variance (price, brand, genre, etc.) are precisely those that latent-factor models already capture, and add little predictive capacity (see Figure 5.2). A notable exception to this is

features that are not *static*: while a simple feature like the price of an item may already be 'baked in' to a latent factor representations, what our current models cannot tell us is how a user would react to a *change* in price. As such, much of the research we'll explore below is concerned questions of price variability, and modeling the impact that a change in price will have on user preference.

Early attempts at modeling price within the context of recommender systems explored the overall difficulty of successfully incorporating price dynamics Umberto (2015).

5.6.1 Disentangling Prices and Preferences

Ge et al. (2011) considers price from the perspective of a user who wants recommendations that satisfy a *budget constraint*. They consider this problem in the context of recommending 'travel tours,' where a user has constraints in terms of time (the length of the vacation), and the amount they are able to spend. They note however, that while the length and price of an actual travel package is observed, a user's constraints may not be, and as such must be modeled (or estimated) based on their historical activities.

To achieve this type of price-aware recommendation, Ge et al. (2011) consider a modification of a latent factor model that includes both a preference compatibility, and a price-compatibility term:

$$f(u,i) = \underbrace{S(C_u, C_i)}_{\text{price compatibility}} \cdot \overbrace{\gamma_u \cdot \gamma_i}^{\text{user preference}}. \tag{5.23}$$

γ_u and γ_i are user and item (travel tour)-related latent factors, as in **??**. C_u and C_i are cost-related factors; the cost for the tour (C_i) is assumed to be an observed, two-dimensional vector, encoding both time and price; C_u is assumed to be a corresponding latent compatibility encoding the user's price constraints; S is then a compatibility function, such as the (negative) euclidean distance:

$$S(C_u, C_i) = 1 - \|C_u - C_i\|_2^2. \tag{5.24}$$

A few improvements to this basic model are proposed (including in a follow-up paper Ge et al. (2014)); for example, different training strategies are proposed based on different types of explicit and implicit feedback; and the user factor C_u is carefully regularized (since e.g. a trivial ℓ_2 regularizer would center user price constraints around zero).

Experiments (on a proprietary dataset of historical travel tour interactions) show that the model outperforms variants that fail to consider price information.

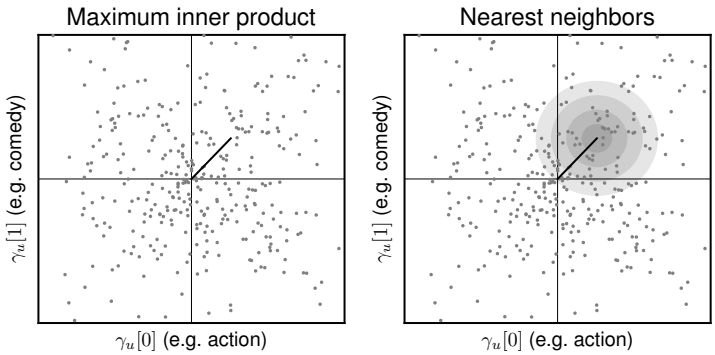


Figure 5.3 User vectors γ_u in latent space, and candidate items to be recommended. Highly compatible items appear in the highlighted region under inner product (left) and Euclidean (right) compatibility models.

Guo et al. (2017b) also builds a model to separately capture price and preference dynamics. Although the specific method is somewhat different from those we've discussed (a form of Poisson Factorization, see e.g. Gopalan et al. (2013)), this approach has a common goal with Ge et al. (2011) of separating price and preference concerns. In essence, latent item properties γ_i are responsible for estimating ratings via $\gamma_u^{(\text{rating})} \cdot \gamma_i$ and also price compatibility via $\gamma^{(\text{price})} \cdot \gamma_i$; compatible items are then those that satisfy both of these concerns.

5.6.2 Estimating Willing-to-Pay Prices within Sessions

Hu et al. (2018) also considered the effect that price has on users' purchasing decisions, but did so at the level of individual browsing sessions. That is, the sequence of products a user browses might provide some clue as to their purchase intent or their 'willing-to-pay' amount, e.g. if they are comparison shopping among items within a certain price range.

Like Ge et al. (2011), the basic model of Hu et al. (2018) extends a latent-factor model to incorporate a price-compatibility term:

$$f(u, i) = \gamma_u \cdot \gamma_i + \alpha_u C(u, p_i). \tag{5.25}$$

Here $C(u, p_i)$ encodes the compatibility between the user u and the price p_i of the item i, and α_u is a personalized measure of user u's sensitivity toward this term.

A trivial price-compatibility term might take the form

$$C(u, p_i) = \exp\left(-\omega(b_u - p_i)^2\right) \tag{5.26}$$

Table 5.2 *Comparison of price-aware recommendation techniques.*

Reference	Dataset	Description
Ge et al. (2011)	(Proprietary) data of travel-tour purchases	Incorporates price and time constraints into travel-tour recommendations
Guo et al. (2017b)	Various *Amazon* categories	Disentangles interactions in terms of preferences versus price compatibility.
Wan et al. (2017)	Purchases from Seattle grocery stores	Estimates how purchase decisions (item choice, quantity, etc.) are affected by price fluctuations.
Hu et al. (2018)	Purchase and browse data from *Etsy*	Forecasts a user's target purchase price from a sequence of browsed items.

where p_i is the price of the item and b_u is a latent estimate of the user's budget; this function is essentially a variant of Equation (5.24).

To incorporate session dynamics, Hu et al. (2018) considers a price-compatibility term based on a feature vector ρ_i, which is a one-hot encoding representing the price *quantile* of the price p_i compared to the prices of previously-viewed items in the session;[2] then the price-compatibility is merely $C(u, p_i) = \theta \cdot \rho_i$. This is further extended by using a *mixture model*, which essentially says that there could be different parameters θ_g for different (latent) users groups:

$$C(u, p_i) = \sum_g \underbrace{\frac{e^{\psi_{u,g}}}{\sum_{g'} e^{\psi_{u,g'}}}}_{\text{extent to which } u \text{ belongs to group } g} \overbrace{\theta_g \cdot \rho_i}^{\text{price-compatibility model for group } g}. \tag{5.27}$$

What Hu et al. (2018) ultimately find is that there are several different classes of user (based on latent group membership $\psi_{u,g}$): some tend to gradually browse toward more expensive items, others gradually browse toward cheaper items,

5.6.3 Price Sensitivity and Price Elasticity

Notions such as *price sensitivity*, and *price elasticity* (defined as the change in purchase quantity given a change in price) are well understood in economics and marketing Case and Fair (2007). Understanding such factors can help to guide custom marketing and promotion strategies Zhang and Krishnamurthi (2004); Zhang and Wedel (2009).

[2] For example, $\rho_i = [0, 0, 0, 1]$ would indicate that the price p_i was among the top 25% of browsed prices.

However they are less well understood in terms of their effectiveness in a *predictive* setting, i.e., in terms of how price should be used to understand and forecast user actions (or to make recommendations).

Part of the reason that price has received relatively little attention (at least in academic literature) is presumably the lack of useful available data; even when price is observed, it is confounded by numerous other factors, such as brand or manifest aspects of a product; moreover even when price data is available, on rarely has *historical* data on price that allows for measurement of the impact of a *change* in price.

Wan et al. (2017) studied price in the context of grocery recommendation. Their research was mostly based on real transaction data from a physical grocery store (in Seattle), though was also validated based public data from *dunnhumby*. Both datasets contain price measurements, and critically measurements of price *variation* over time. As such the main questions center around the extent to which purchase decisions are affected by changes in price.

In the context of grocery shopping, potential questions include:

- Will a reduction in price cause a user to buy a *category* of product they otherwise wouldn't have (e.g. would they buy milk at a discount if it wasn't on their shopping list)?
- Will a reduction in price cause users to buy a *specific item* that they otherwise wouldn't have (e.g. would they buy a different brand of milk because of a discount)?
- Will a reduction in price cause users to buy a *larger quantity* of an item than they otherwise would have?

To study these questions, prediction is broken down into three subsequent choices:

$$\begin{aligned} p_u(\text{buy } q \text{ units of an item } i \text{ from category } c) = \\ & p_u^{(\text{category})}(\text{buy a product from category } c) \\ \times & p_u^{(\text{item})}(\text{buy product } i \mid \text{buying from category } c) \\ \times & p_u^{(\text{quantity})}(\text{buy } q \text{ units} \mid \text{buying item } i). \end{aligned} \tag{5.28}$$

Each of these three prediction tasks (category, item, and quantity prediction) is based on a predictor $f(u, c, t)$, $f(u, i, t)$, and $f(u, q, t|i)$; the underlying method behind each is a latent-factor model, as in **??**, including additional features associated with the time (e.g. what day of the week the trip occurs on). Each is passed through a different activation function, for example quantity prediction

is modeled via a *Poisson* function:

$$p(\text{quantity} = q|\text{buying item } i) = \frac{f(u,q,t|i)^{q-1}\exp(-f(u,q,t|i))}{(q-1)!}. \tag{5.29}$$

Next, the change in purchase probability due to price is captured (for each of the three models) using a simple feature encoding the price at a particular point in time. Specifically (e.g. for quantity)

$$f'(u,q,t|i) = f(u,q,t|i) + \beta_{u,q}\log P_i(t), \tag{5.30}$$

where $P_i(t)$ is the price of the item at time t. $\beta_{u,q}$ is then a bias term which encodes the price sensitivity, i.e., extent to which a particular user u, when purchasing q units of an item, will react to changes in price; a negative value of $\beta_{u,q}$ would indicate that a user is less likely to purchase a particular quantity given a price increase. All parameters are learned by training on purchase data, using a BPR-like training scheme.

Price-elasticity now reflects how much preference changes given a change in price, e.g. for a particular item i:

$$e_{i,u}(t) = \frac{df'(u,i,t)}{f(u,i,t)}\Big/\frac{dP_i(t)}{P_i(t)} \approx (1 - f'(u,i,t))\beta_{u,i}(t). \tag{5.31}$$

A related notion, *cross-elasticity*, measures the extent to which a change in i's price will change a user's compatibility toward (e.g. probability of purchasing) a *different* product j:

$$e_{i,j,u}(t) = \frac{df'(u,j,t)}{f(u,j,t)}\Big/\frac{dP_i(t)}{P_i(t)} \approx -f(u,i,t)\beta_{u,i}(t). \tag{5.32}$$

Note that the above equations (price-elasticity and cross-elasticity) are *measurements* after the model has been trained. The main finding of the model is that price-elasticity applies mostly to product choice, but not to category choice or quantity.

Ruiz et al. (2020) develops a somewhat similar model of consumer choice, also in a setting of grocery purchases. Like the above model, Ruiz et al. (2020) attempts to disentangle the various effects of item popularity, user preference, and price dynamics, though also includes additional terms involving seasonal effects. The main goal of the paper is to answer 'counterfactual' queries about price (i.e., what would the user have done if the price had been different?). By modeling how users will react to changes in price, they argue that the model is also able to detect interactions between products, namely in terms of which items are likely to be substitutable and complementary.

Finally, we mention attempts to use similar ideas within the context of dynamic pricing. Jiang et al. (2015) seek to combine ideas from ideas from from

pricing and recommendation in order to design optimal (i.e., profit-maximizing) promotion strategies. Conversely, Chen et al. (2016) analyzes the characteristics of sellers on *Amazon*, in order to automatically detect the presence of algorithmic pricing.

5.7 Other Contextual Features in Recommendation

5.7.1 Music and Audio

Improving Content-based and Hybrid Music Recommendation using Deep Learning Wang and Wang (2014)

Wang and Wang (2014) notes the difficulty of directly using high-dimensional audio features (e.g. based on spectrogram-based representations of audio) within a traditional feature-based recommender system. The solution they propose is to use a neural network-based representation (essentially a multi-layer perceptron) to learn embeddings of songs that are useful for recommendation, i.e.,:

$$\gamma_i = MLP(x_i) \tag{5.33}$$

Deep content-based music recommendation Van Den Oord et al. (2013)
Million song dataset McFee et al. (2012)

5.7.2 Recommendation in Location-Based Networks

Several attempts have been made to incorporate geographical features into recommender systems. Actions are often guided by geographical constraints, whether due to a user operating within a certain geographical region, or due to sequential actions being highly localized. Bao et al. (2015)

5.7.3 Temporal, Textual, and Visual Features

So far we have covered content in recommender sytems, including features ranging from price, geography, social signals, and audio. In the following chapters we'll revisit content-aware recommendation techniques using features based on temporal and sequential dynamics (Chapter 6, text (Chapter 7), and images (Chapter 8). These modalities require special attention, primarily to deal with the complex semantics of the data and high-dimensional signals involved. Exploring how to build personalized models based on these complex modalities of data is one of the main themes explored throughout the remainder of this book.

5.8 Exercises

Exercises

5.1 In **??** we introduced the Factorization Machine as a general-purpose technique for incorporating features into recommender systems. In this exercise we'll incorporate a few features into a factorization machine to measure the extent to which they improve recommendation performance. You could use any dataset for this exercise, so long as it includes a few features. For example, using our beer dataset (as in **??**) we might include features such as:

- Simple numerical features such as the ABV of the beer or the age of the user.
- The category of the item (one-hot encoded).
- The timestamp; this will require some care to encode, e.g. you might use an encoding of the season or day of week.

Using a few of these features (or similar features on another dataset), measure the extent to which performance is improved by adding features to the recommender.

5.2 In **??** we discussed the potential value of incorporating features (side information) into recommender systems, and argued that features may be most informative in *cold-start* scenarios. To assess this, conduct the following experiment (using your model from **??** 5.1):

- For each user (or item) in the test set, count how many times that user appeared in the training set.
- Plot the testing performance (y) of your model as a function of how many times that user appeared in the training set (x).
- Generate the same plot both with and without additional features in your factorization machine.

Naturally, our expectation is that performance will improve as users (or items) are less 'cold' (i.e., more training interactions). However, we expect the performance degredation to me more mild when features can compensate for the lack of interactions.

5.3 Before implementing a socially-aware recommender, here we'll test the hypothesis that friends actually interact with more (or less) similar items compared to randomly chosen sets of users. This could be measured in various ways, e.g.:

- Compute the average Jaccard (or cosine, etc.) similarity between randomly chosen pairs of users, versus randomly chosen pairs of friends.

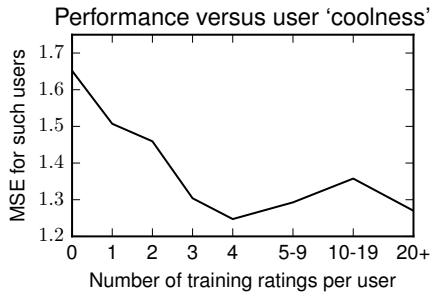


Figure 5.4: Performance (MSE) as a function of user coolness (defined as the number of times that user was seen during training). Goodrdeas 'Graphic Novels' data.

5.4 In **??** we explored a few ways to incorporate social signals into one-class recommender systems (**??**). For the most part, these techniques amounted to ways of sampling negative feedback instances, e.g. we might be more (**??**) or less (**??**) to interact with negative instances that our friends have already interacted with. Using a dataset that contains social interactions (e.g. **??**, as in **??**) experiment with these sampling strategies and determine which (if any) lead to improved performance over a traditional BPR implementation.

5.5 Incorporate price into some dataset (amazon?) in a few ways

5.8.1 Project 3: Cold-Start Item Recommendations on *Amazon*

As we argued in **??**, one of the main reasons to incorporate features into personalized recommendation approaches is to improve their performance in *cold-start* settings. Here, we'll look at cold-start recommendation problems using data from *Amazon*, following data from (e.g.) **?**. We select this dataset as it includes various types of item metadata (prices, brands, categories, etc.) that can potentially be useful in cold-start settings, though this project could be completed using any dataset that includes user or item metadata.

We'll build our system for cold-start recommendation via the following steps:

(i) Start by implementing a 'vanilla' factorization machine to solve the prediction problem, i.e., without incorporating any side-information. Note that this problem could be cast either as one of rating prediction (as in **??**) or as purchase prediction (as in **??**).

(ii) To build models for cold-start recommendation, it is useful to develop some evaluation metrics specifically for the purpose of evaluating cold-start (and 'cool-start') performance. To do so, try plotting the performance on the test set *as a function of the number of times the item appears in the training set*.

Our hypothesis when incorporating side-information into recommendations is that performance improvements will be largest for the coldest items, and less useful for items with longer interaction histories. We'll use this same type of plot to compare models in the following.

(iii) Several features could potentially be useful in cold-start scenarios. Consider how to encode the following: (a) the brand of the item; (b) the category (or categories) the item belongs to; and (c) the price of the item. You might also consider more complex features based on (e.g.) the text of the description or temporal information, though we'll revisit these topics in Chapters 6 and 7.

For each cold-start feature you include, compare

6

Temporal and Sequential Models

Throughout Chapter 4 and **??** we developed gradually more refined arguments about the role that features (or 'side-information') play when modeling user interaction data. Our initial argument (started in **??**) was that latent user and item representations are sufficient to capture complex preference dynamics, and whatever features are most predictive (i.e., explain the most variability) of interactions will automatically emerge via our latent representations.

Later, (in **??**) we refined this argument, arguing that side-information *can* be useful, especially in settings where there is a paucity of interaction data from which to learn high-dimensional latent representations (i.e., the *cold-start* setting).

However in both of the above cases we still assumed that our model of users and items was *stationary*. In practice, preferences and interactions can be non-stationary for a variety of reasons. For instance, no matter how many interactions we observe, and how many latent factors we fit, a model such as the one we developed in **??** cannot tell us that a user might only buy swimsuits in summer, or that they are unlikely to watch the third film in a series until they have already seen the second.

In this chapter we explore techniques to build personalized models around user behavior that has *temporal and sequential dependencies*. Most straightforwardly, we might simply treat temporal and sequential information as additional features that can be modeled, e.g. via the framework of a Factorization Machine as we developed in **??** (and indeed we will explore temporal models based on this type of approach). However as we will see in this chapter, a variety of complex and subtle temporal dynamics could be at play, for example:

- Dynamics could apply to users (e.g. users may grow out of certain movies), items (e.g. a movie's special effects may become dated over time), or the *zeitgeist* of the entire community may shift over time.

- Temporal dynamics can exist at multiple scales, e.g. when modeling heart-rate data (Section 6.8), dynamics can be short-term (e.g. a user running), medium-term (e.g. a user getting tired) or long-term (e.g. a user becoming more fit).
- In addition to short- and long-term dynamics, dynamics can also be periodic (e.g. weekly or seasonal trends), or prone to bursts and outliers (e.g. purchases around a major holiday).

Our main focus in this chapter is to explore a wide variety of (personalized) models for temporally-evolving data. As we'll see, dynamics such as those above require carefully-designed models (rather than simply including a temporal feature in a general-purpose model). As such, we focus on understanding the overall process and design considerations when building personalized models with temporal dynamics.

6.1 Introduction to Regression with Time Series

Before investigating more sophisticated models of temporal and sequential data, it is instructive to consider how much progress can be made with techniques we have already developed.

To do so, we'll consider developing predictors that estimate the next value in a sequence (or the next several values). In the simplest case, we are given a sequence of observations $y = \{y_1 \dots y_n\}$, from which one would like to predict the next value (y_{n+1}). For example, one might want to estimate website traffic on the basis of historical traffic patterns.

To solve such a problem using a regression approach (or a classification approach for a binary outcome), we can imagine constructing features based on the previous observations ($\{y_1 \dots y_n\}$) in order to predict the next one. Presumably, once we observe the true value of y_{n+1}, we would like to predict y_{n+2}, and so forth. Note that this blurs the line somewhat between 'features' and 'labels,' since the label for one prediction becomes a feature for the next.

This procedure is known as *auto-regression* (referring to the fact that we are regressing on the same data that was used for prediction). As usual, we are typically interested in defining our regressor (or classifier) such that it minimizes some error between the predictions and the true values. That is, we want to define a predictor $f(y_1 \dots y_n)$ that estimates the next value (y_{n+1}) in the sequence so as to minimize (e.g.) a Mean Squared Error:

$$\frac{1}{n}\sum_{i=1}^{n}(f(y_1 \dots y_i) - y_{i+1})^2. \tag{6.1}$$

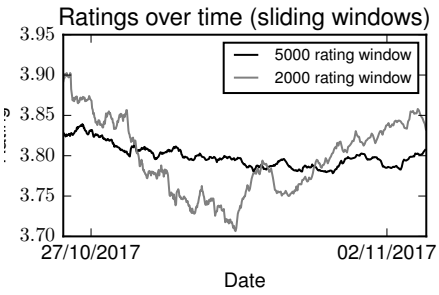


Figure 6.1: Moving-average plots of ≃1 week of *Goodreads* Fantasy novel ratings. Although not particularly effective as predictors, moving averages can be used to plot data so as to summarize overall trends.

Trivially, we might imagine several naïve techniques for estimating the next value in a sequence, e.g. we could predict the next value as a weighted sum of previous values:

$$\text{moving average:} \quad f(y_1 \dots y_n) = \frac{1}{K} \sum_{k=0}^{K-1} y_{n-k} \tag{6.2}$$

$$\text{weighted moving average:} \quad f(y_1 \dots y_n) = \frac{\sum_{k=0}^{K-1} y_{n-k}}{\sum_{k=1}^{K} k}. \tag{6.3}$$

Although simple, one can imagine how these averages could potentially be better predictors than always predicting the next value to be equal to the previous (for example, see **??**). These types of trivial predictors can also be used to plot trends in noisy data (Figure 6.1); averages over larger intervals (i.e., larger values of K) will produce smoother summaries of the data. Such averages can be efficiently computed for successive values via a dynamic programming solution; code to generate the plots in Figure 6.1 is presented below:

```
1  xSum = sum(x[:wSize]) # Given data x and y to plot, and a window
       size wSize
2  ySum = sum(y[:wSize]) # Sum of first wSize values
3  xSliding = []
4  ySliding = []
5
6  for i in range(wSize,len(x)-1):
7      xSum += x[i] - x[i-wSize] # Strip off oldest value and add
           newest one
8      ySum += y[i] - y[i-wSize]
9      xSliding.append(xSum / wSize)
10     ySliding.append(ySum / wSize)
```

The two trivial strategies above are heuristics for predicting the next value, capturing the intuition that it should be similar to recently observed values (Section 6.1), and possibly that more recent values should be more predictive than less recent ones (Section 6.1). However, a better strategy might be to *learn*

which of the recent values are the most predictive, i.e.,

$$f(y_i \dots y_n) = \sum_{k=0}^{K-1} \theta_k y_{n-k}. \tag{6.4}$$

The values θ_k now determine which of the previous k values are predictive. For example, in periodic data (e.g. network traffic, seasonal purchases), the most predictive values may include recent observations, observations from 7 days ago, observations from the same day in the previous month, etc.

For example, training a simple autoregressive model on a dataset of hourly measurements of bike rentals in the Bay Area[1] yields the following model:

$$\begin{aligned} y_n = \; & 0.471 \times y_{n-1} \\ & - 0.284 \times y_{n-2} \\ & + 0.106 \times y_{n-3} \\ & + 0.014 \times y_{n-4} \\ & - 0.021 \times y_{n-5} \\ & + 0.175 \times y_{n-24} \\ & + 0.540 \times y_{n-24\times 7} \end{aligned} \tag{6.5}$$

Here we see that the two most predictive observations are those from the previous hour (y_{n-1}) and from exactly one week ago ($y_{n-24\times 7}$). Observe that we didn't include *every* previous observation up to 24×7 hours ago as a feature, rather we only included those previous observations that we expected to be predictive.

Note that although the solution in Equation (6.4) uses *only* previous values in the sequence, one can of course include other features associated with the current timepoint or previous values, just like in a normal regression model.

Although a simple approach to regression and classification of time series data, the basic idea behind auto-regression (i.e., to use previous observations to predict future values in a sequence) will reappear in many of the more complex models we develop, especially as we develop sequential models in **??**.

6.2 Temporal Dynamics in Recommender Systems

Several attempts have been made to improve recommendations by incorporating temporal dynamics. There are countless reasons why preferences, purchases, or interactions may change over time, or more simply why knowing the current timestamp may help us to more accurately predict the next interaction.

[1] Each observation y_h is a measurement of how many trips were taken during hour h.

Consider for example the following scenarios which could cause changes in movie ratings or interactions over time:

(i) Users may give higher ratings to older movies, e.g. due to feelings of 'nostalgia'.
(ii) Alternately, ratings of older movies may represent a biased sample of items that users had explicitly searched for (versus newer items which a user selected due to their being surfaced on a landing page).
(iii) A mundane change to a user interface, such as modifying the tool-tip text associated with a certain rating, may alter the rating distribution.
(iv) Users who favor special effects may give lower ratings as a movie's special effects become dated.
(v) A family member may borrow a user's account, and temporarily consume movies quite different from the account's typical preferences.
(vi) Users may binge-watch a series, dominating their interaction patterns for a short period.
(vii) Action blockbusters may be more favored during summer (or Christmas movies during Christmas).
(viii) Users may want to consume (or avoid) content very similar to what they have previously interacted with.
(ix) Users may gradually develop an appreciation for certain characteristics of a movie as they consume more content from that genre.
(x) Users may be anchored by external forces, i.e., the *zeitgeist* of what is currently popular in their community.

The above dynamics are quite varied in their sources and scale: Effects (i) and (iv) are gradual and long-term; (i) and (iii) owe to vageries of a changing user interface; (v) and (vi) are 'bursty' or short-term; (vii) is seasonal; (viii) is sequential; (ix) owes to user growth; and (x) due to a changing community. One can imagine many other sources, especially in different settings subject to social dynamics, price variability, fashion, etc. As we will see, understanding such dynamics is often key to making successful recommendations. Some may scarcely seem like 'temporal dynamics' at all: e.g. a change to the user interface has little to do with user or item evolution. Nevertheless, we'll argue that modeling even these trivial or mundane dynamics proves critical in order to disentangle them from the 'real' real personalization dynamics in the data.

Methods for Temporal Recommendation Methods for temporal recommendation fall broadly into two classes. The first make use of the actual *timestamps* of events. Each interaction (u, i) is augmented with a timestamp (u, i, t), and the

goal is to understand how ratings $r_{u,i,t}$ vary over time. That is, our goal is to extend models such as the one in **??** so that parameters vary as a function of time, e.g.

$$r_{u,i,t} = \alpha(t) + \beta_u(t) + \beta_i(t) + \gamma_u(t) \cdot \gamma_i(t). \tag{6.6}$$

Modeling temporal dynamics in this way is effective at capturing long-term shifts of community preferences over time; short-term or 'bursty' dynamics, such as purchase patterns being affected by external events, or periodic events, such as purchases being higher at a particular time of day, day of week, or season.

A second class of methods discards the specific timestamps, but preserves only the *sequence* (or order) of events. Thus the goal is generally to predict the next action as a function of the previous one, i.e.,

$$p(\text{user } u \text{ interacts with item } i | \text{they previously interacted with item } j). \tag{6.7}$$

This type of model makes the assumption that the important temporal information is captured by the *context* provided by the most recent event. This is useful in highly-contextual settings, such as predicting the next song a user will listen to, or other items they will place in their basket, etc. In such settings knowing the most recent actions (or most recent few actions) is more informative than knowing the specific timestamp.

These two classes of model are quite orthogonal, both in terms of the settings where they are effective, as well as the techniques involved. Below we'll explore both settings via several case-studies. First we'll explore parametric temporal models (as in Equation (6.6)) through the example of the Netflix prize, where temporal dynamics are carefully modeled to capture a wide variety of application-specific dynamics. We explore sequential models (as in Equation (6.7) in Section 6.5 via several examples, starting with online shopping scenarios; these settings tend to be highly contextual, where the context of recent activities is often more informative than long-term historical activities.

Later, we'll introduce *recurrent networks* as general-purpose models of sequence data (**??**), and as a potential approach to capture complex dynamics in evolving interaction sequences. Such models have the potential to overcome the limitations of traditional sequential models by learning complex semantics that persist over many steps. Such models arguably represent the current state-of-the-art for general purpose recommendation.

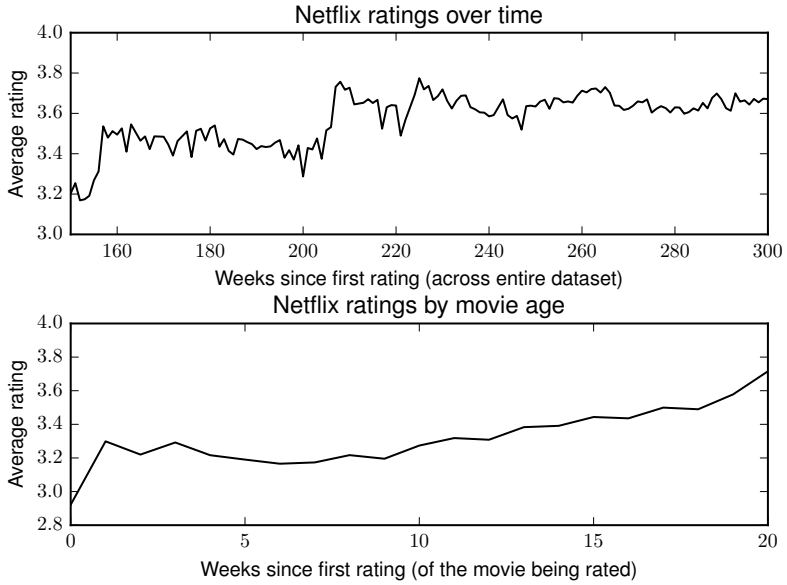


Figure 6.2 Temporal dynamics on Netflix. The top plot shows ratings averaged across each week over the lifetime of the dataset; the bottom plot shows how ratings change for newly-introduced movies, showing that ratings gradually increase during the first few weeks the movie is on Netflix. These plots reveal a combination of sudden and gradual trends in movie ratings over time.

6.2.1 Case Study: Temporal Recommendation and the Netflix Prize Koren (2009)

Careful modeling of temporal dynamics was one of the key features characterizing the strongest solutions to the Netflix Prize. Several of the ideas that proved effective for modeling temporal dynamics on Netflix specifically are covered in Koren (2009) ("Collaborative Filtering with Temporal Dynamics"), which we summarize here.

As a motivating example, consider the two plots shown in Figure 6.2.[2] At top, we see ratings over time averaged across weekly bins. We see, several points where ratings appear to increase suddenly, followed by plateaus of relatively stable ratings; for example at around week 210, average ratings appear to jump from around 3.5 stars to 3.6 stars.

Such long-term, population-level changes could owe to several explanations.

[2] These two plots are based on similar figures from Koren (2009); the purported trends are somewhat more pronounced in the original paper, possible due to differences in sampling schemes.

Changes in rating patterns could be due to changing user base, or from certain movies being added to Netflix; such changes could even be due to world events exogenous to Netflix. Or, the change could owe to a factor as simple as a change in Netflix's User Interface (UI), causing users to rate movies differently.

The bottom plot in Figure 6.2 shows another temporal trend, demonstrating that individual movies recieve higher ratings the longer they have been available on Netflix. Again, such trends could be due to a variety of factors: users may be favorably biased toward older movies (e.g. by nostalgia); or again it could be a function of the UI: a user who specifically sought out an older movie may view it more favorably than a user who had been recommended the same movie from the front page.

Whatever the underlying cause of these trends, they account for significant variability in the observed ratings. And as such, we should model these dynamics to predict ratings more accurately. Naïvely, one might simply discard (e.g.) older data that does not correspond to the predominant rating trends. But, a more effective model would attempt to account for the differences between newer and older data.

To model the kinds of long term trends captured in Figure 6.2, Koren (2009) first focuses only on temporally evolving bias terms, i.e.,

$$b_{u,i}(t) = \alpha + \beta_u(t) + \beta_i(t). \tag{6.8}$$

Starting with item biases, one can capture long-term, gradual variation simply by having different bias terms for different periods, i.e.,

$$\beta_i(t) = \beta_i + \beta_{i,\text{Bin}(t)}. \tag{6.9}$$

In Koren (2009) the authors suggest $\sim$ 30 bins corresponding to about 10 weeks each in the case of the Netflix data.

This basic idea of separating bias terms into bins could likewise be applied to capture periodic trends, much like we encoded periodic terms in Section 3.3.4. Here the bias term would again take the form of several bins:

$$\beta_i(t) = \beta_i + \beta_{i,\text{Bin}(t)} + \beta_{i,\text{period}(t)}, \tag{6.10}$$

where period(t) could represent periodic effects at the level of different days of the week, or months of the year (etc.).

The above ideas are effective for modeling long-term and periodic dynamics, but are quite expensive: e.g. the model in Equation (6.9) requires an additional 30 parameters *per item*; this is affordable for items on Netflix since the average item has over 5,000 ratings (100 million ratings of 17,770 titles). However this would likely not be possible for users, as the average user has only around 200

ratings. Thus one needs a way of parameterizing user temporal dynamics that is considerably cheaper (i.e., involves fewer parameters).

A solution suggested in Koren (2009) is to use an 'expressive deviation' term for each user:

$$\text{dev}_u(t) = \underbrace{\text{sign}(t - t_u)}_{\text{before } (-1) \text{ or after } (1) \text{ the mean date}} \cdot |t - t_u|^x. \tag{6.11}$$

The term t_u represents the mean date amongst a particular user u's ratings, so that the term $(t - t_u)$ represents whether a particular point in time t is before or after the midpoint of the user's rating lifetime. The expressive deviation term is depicted in Figure 6.3; the exponent of 0.4 was found to work well on Netflix data. The deviation term augments the user bias term via a user-specific scaling term α_u, essentially controlling how strongly the deviation term applies to a specific user:

$$\beta_u(t) = \beta_u + \alpha_u \cdot \text{dev}_u(t), \tag{6.12}$$

e.g. a negative value of α_u would mean the user's ratings trend down over time, whereas a value $\alpha_u \simeq 0$ would mean that the user's overall bias is not subject to temporal variation. A similar strategy can also be used to capture variation at the level of individual latent dimensions, e.g.

$$\gamma_{u,k}(t) = \gamma_{u,k} + \alpha_{u,k} \cdot \text{dev}_u(t) + \gamma_{u,k,t}. \tag{6.13}$$

Note that the deviation terms in Equations (6.12) and (6.13) add only a single term per user (Equation (6.12)) or a single term per factor (Equation (6.13)). The final term $\gamma_{u,k,t}$ in Equation (6.13) (a temporally-evolving term applied to a specific factor for a specific user) models highly-local preference dynamics and can be used to model (e.g.) day-specific variability; this term is however highly expensive (in terms of the number of parameters introduced) so may only be available for some users.

When combined with the carefully-engineered deviation term of Equation (6.11) this allows the model to capture quite complex dynamics while adding only a modest number of parameters; this basic design philosophy (a carefully-designed parametric model that costs only a small number of parameters) will prove a common theme when designing temporal models (**??**).

While the expressive deviation term is effective for users who have few interactions, for users who have more it can be useful to fit a more complex model. To do so, a spline function can be used to model gradual shifts in user biases. A spline function smoothly interpolates between a series of control points via

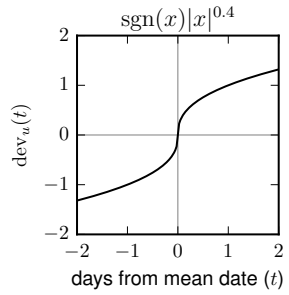


Figure 6.3: Expressive deviation term from Koren (2009).

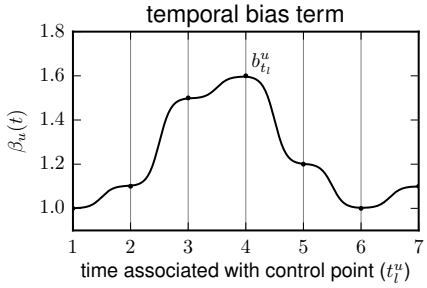


Figure 6.4: Spline interpolation of temporally evolving user bias.

the following function:

$$\beta_u(t) = \beta_u + \frac{\sum_{l=1}^{k_u} e^{\gamma|t-t^u_l|} b^u_{t_l}}{\sum_{l=1}^{k_u} e^{-\gamma|t-t^u_l|}}. \tag{6.14}$$

Here k_u is the number of control points for user u (which grows with the number of ratings that user has entered); t^u is a series of uniformly spaced time-points for each user; and $b^u_{t_l}$ is the bias associated with each control point. This type of interpolation is depicted in Figure 6.4.

The above term is a reasonably flexible way to capture gradual drift in user preferences, though still can't handle sudden changes. **?** addresses this with a 'per day' user bias $\beta_{u,t}$, which can be useful for particular days in which users have a lot of activity. Note that while such a bias is unlikely to be helpful when predicting future events, by modeling outliers in this way, the model can essentially learn to 'ignore' events that are not useful for prediction. This too is a common design principle when building temporal models: the goal is not so much to 'forecast' *future* trends but rather to adjust and account for *past* events appropriately.

The above ideas captures the essential components described in Koren (2009), though their work does include an exploratory study of several alternative mod-

eling approaches, including incorporating temporal dynamics into neighborhood models like that of **??**.

6.2.2 What can Netflix Teach us about Temporal Models?

The model developed in the above case study involved several decisions that are quite specific to Netflix, and indeed the model was designed with the explicit goal of achieving strong performance on a single dataset. As such, many of the specific choices (such as the specific parametric form of the expressive deviation term) may not generalize to other settings. Nevertheless, there are several important lessons in the above study that apply general when developing temporal models of user behavior:

- First, developing bespoke models around specific types of dynamics is somewhat the norm when designing temporal models. As we will see in the following sections, different temporal models abound to capture dynamics in different settings.
- One reason for the proliferation of many different hand-crafted temporal models (rather than more 'general purpose' models as we developed in **????**) is that temporal models are *expensive*, in terms of the number of parameters required. As such, models are designed to capture the required dynamics while maintaining model parsimony (i.e., using as few parameters as possible).
- Modeling temporal dynamics is not about *forecasting* so much as it is about determining how to properly adjust, re-weight, or discount past events.
- Some specific lessons from Netflix generalize quite well, especially the importance placed on temporally evolving bias terms: the vast majority of temporal variation often owes to shifts in item popularity, user activity (etc.), and can be captured via evolving β_i or β_u; temporally evolving latent factors (γ_i or γ_u) play less of a role, or otherwise are simply too expensive to model.

6.3 Other Approaches to Temporal Dynamics

6.3.1 Long-Term Dynamics of Opinions

Early works that deal with temporal dynamics explore the notion of *concept drift* Widmer and Kubat (1996); Tsymbal (2004); early works on this topic are concerned with systems for classification in settings with temporally-evolving

Figure 6.5 The models we discussed when exploring temporal dynamics on *Netflix* teach us several general lessons about building temporal models in other settings.

- Successful solutions to the *Netflix Prize* highlighted the critical importance of temporal dynamics in recommendation settings. While the approaches we explored in **??** are quite Netflix-specific, they highlight a general philosophy followed when building temporal models: temporal models tend to be carefully designed around the dynamics of specific datasets and applications.
- A main focus when building temporal models is that of model *parsimony*: to prevent the parameter space from exploding, one must carefully choose models that capture the desired dynamics with as few parameters as possible.
- Temporal dynamics may range from 'lofty' concepts such as users becoming nostalgic toward older movies, to more mundane sources of variation, such an overall shift in average ratings across the community. Both are important to model as both explain variance in the data: untangling even mundane dynamics is a critical step toward extracting meaningful personalization signals.
- The goal of temporal models is typically not to forecast long-term trends. E.g. our model of opinion dynamics on *Netflix* tells us nothing about what will be popular next year. Rather the goal is typically to account for discrepencies across different time periods so that interactions across time can be meaningfully compared. The resulting model thus gives a more accurate sense of *current* rating dynamics, if not future trends.

data. Simple approaches consist of (e.g.) taking only a *window* of recent examples during training (much like we saw in Figure 6.1). More sophisticated approaches allow the context window size to adapt based on how 'stable' a particular concept is; or to reuse concepts that recur periodically; or to distinguish drifting concepts from noise. Models based on these ideas, along with theoretical results, are discussed in Widmer and Kubat (1996), among others.

Among temporal techniques for recommendation, early approaches incorporate temporal factors into heuristic techniques, such as the model from Equation (4.20). For example in Ding and Li (2005), the basic idea is to weight related items in Equation (4.20) so that recent interactions are weighted more highly:

$$r(u,i) = \frac{\sum_{j \in I_u} R_{u,j} \cdot \text{Sim}(i,j) \cdot f(t_{u,j})}{\sum_{j \in I_u} \text{Sim}(i,j) \cdot f(t_{u,j})}. \tag{6.15}$$

Here $t_{u,j}$ is the timestamp associated with rating $R_{u,j}$, and $f(t_{u,j})$ is a monotone function of the timestamp. For example, relevance can decay exponentially for older items:

$$f(t) = e^{-\lambda \cdot t} \tag{6.16}$$

Other than the methods we discussed in Section 6.2.1 for temporal recommendation on *Netflix*, other papers studied the notion of gradually evolv-

ing concepts within the context of preferences and opinions. Godes and Silva (2012)

6.3.2 Short-Term Dynamics

Temporal recommendation on graphs via long- and short-term preference fusion

6.3.3 User-Level Temporal Evolution

Most of the sources of temporal dynamics we've explored so far owe to shifting properties of items (or how items are perceived over time, due to e.g. nostalgia)

McAuley and Leskovec (2013b) Modeling the evolution of user expertise through online reviews

6.4 Personalized Markov Chains

The temporal models we saw in Section 6.2 directly modeled (or extracted features from) the *timestamps* associated with each interaction. We showed how features extracted from timestamps could include factors like seasonality, the day of the week, or nostalgia effects (etc.) on the *Netflix* dataset, and later we'll revisit such models in the context of the temporal dynamics of fashion (**??**).

However in many settings, the best predictor of what a user will do next is simply what they did *last*. For example, if you click on a winter coat, then I should recommend you other winter coats, regardless of whether those items are currently in-season.

Even trivial models such as the item-to-item recommenders which we saw in Section 4.3 implicitly made this assumption. For example, the recommendations made in Figure 4.2 are made based purely on the context of what is currently being viewed; the user's historical interactions—or their preferences—are not considered.

Markov Chains The assumption described above—that the next action is conditionally independent[3] of the interaction history given the previous action—describes exactly the setting of a *Markov Chain*. Formally, given a sequence of

[3] Two variables a and b are said to be *conditionally independent* given a third variable c if $p(a, b|c) = p(a|c)p(b|c)$. Essentially this c 'explains' any dependence a and b. In the case of Markov Chains, the most recent event is sufficient to explain the next action's dependence on the history.

interactions (among a discrete set of items $i \in \mathcal{I}$) $I^{(1)} \dots I^{(t)}$, a Markov Chain assumes that the probability of the next interaction given the history can be written purely in terms of the previous interaction:

$$P(I^{(t+1)} = i | I^{(t)} \dots I^{(1)}) = P(I^{(t+1)} = i | I^{(t)}) \tag{6.17}$$

Personalized Markov Chains generalize Equation (6.17) by allowing the probability of the next item to depend on both the previous item and the identity of the user u. That is, for a given user we have:

$$P(I_u(t+1) = i | I_u(t) \dots I_u^1) = P(I_u^{(t+1)} = i | I^{(t)}). \tag{6.18}$$

What this means in practice is just that when predicting a user's next action, our prediction should be a function of their previous action as well as their preference dimensions. Most of the time we can ignore the formalism of Markov Chains and more simply state that we are trying to fit a function of the form

$$f(\overbrace{u, i}^{\text{score associated with the next interaction}} \mid \underbrace{I_u^{(t-1)}}_{\text{given the user's previous interaction}}). \tag{6.19}$$

That is, whereas our previous models took as inputs a user and an item (i.e., $f(u, i)$), or a user, item, and timestamp ($f(u, i, t)$), we now wish to model a user, an item, and the user's previous interaction. We might fit this function via a rating estimation framework (as in Section 4.5), or a personalized ranking framework (as in **??**), etc.

The key challenge in fitting models of the form in Equation (6.19) is that techniques like matrix factorization can no longer be straightforwardly applied. Whereas we previously modeled user/item interactions by factorizing a $U \times I$ matrix we must now factorize a $U \times I \times I$ *tensor*.

We'll explore this idea by investigating specific implementations from case studies in the following section.

6.5 Case Studies: Markov-Chain Models for Personalized Recommendation

Below we describe various attempts to extend latent factor recommendation approaches to incorporate signals from the previous item (though some incorporate additional information such as social signals). The main challenges involved include

summarized in Table 6.1.

Table 6.1 *Markov-Chain models for personalized recommendation.*

Method	Reference	Description
FPMC	Rendle et al. (2010)	The next item should be compatible with the user, as well as the previous item.
SPMC	Cai et al. (2017)	Extends FPMC, incorporating a social term which states that the next item should be similar to our *friends'* previous items.
PRME	Feng et al. (2015)	Similar to FPMC, but measures compatibility via similarity in a metric space.
Translation-based Recommendation	He et al. (2017a)	Replaces the fixed user embedding γ_u with a translation operation in latent item space.

6.5.1 Factorized Personalized Markov Chains

An early paper to use Markov Chains for personalized recommendation was *Factorizing Personalized Markov Chains for Next-Basket Recommendation* Rendle et al. (2010). The paper was specifically concerned with predicting what Factorized Personalized Markov Chains (FPMC for short) predict what items a user will purchase next, based on the items in their previous basket; customer basket data from *Rossmann* (a German drugstore) was used to train and evaluate the model.

The basic premise of FPMC is that the contents of the previous basket should help to predict the contents of the next one, but also that the basket contents should be personalized to the user. This is achieved by fitting a function of the form

$$f(u, i|j) \tag{6.20}$$

where u is a user, i is a potential item to be recommended, and j is an item from the user's previous basket.[4].

The paper discusses the difficulty of modeling sparse interactions of the form in Equation (6.20), and explains how this might be addressed with tensor de-

[4] Although the original paper uses basket data, baskets are mostly a complication needed to handle their specific dataset. It is more straightforward to present the work by simply considering item sequences (which is how the method is often adopted by other papers). Where we write a term for a previous item, in Rendle et al. (2010) that expression would usually be replaced by a sum over items in the previous basket

composition. The decomposition used is essentially a generalization of the matrix factorization schemes we saw in Section 4.5, where $f(u, i|j)$ decomposes into a series of pairwise factors:

$$f(u, i|j) = \gamma_u^{(ui)} \cdot \gamma_i^{(iu)} + \gamma_i^{(ij)} \cdot \gamma_j^{(ji)} + \gamma_u^{(uj)} \cdot \gamma_j^{(ju)}. \tag{6.21}$$

The three terms above denote the user's compatibility with the next item ($\gamma_u^{(ui)} \cdot \gamma_i^{(iu)}$), the next item's compatibility with the previous item ($\gamma_i^{(ij)} \cdot \gamma_j$), and the user's compatibility with the previous item ($\gamma_u^{(uj)} \cdot \gamma_j^{(ju)}$). In practice, the latter expression cancels out when optimizing the model (which is perhaps intuitive as the expression doesn't include the candidate item i), so that the factorization can be rewritten

$$f(u, i|j) = \overbrace{\gamma_u^{(ui)} \cdot \gamma_i^{(iu)}}^{\text{user's compatibility with next item}} + \underbrace{\gamma_i^{(ij)} \cdot \gamma_j^{(ji)}}_{\text{next item's compatibility with previous item}}. \tag{6.22}$$

Intuitively, this factorization simply states that the next item should be compatible with both the user and the previous item consumed. Note that (as indicated by superscripts), item parameters are not shared between the terms $\gamma^{(iu)}$, $\gamma^{(ij)}$, $\gamma^{(ji)}$, i.e., we use separate sets of factors when modeling how an item interacts with a user vs. another item.

Ultimately, the model is optimized using a BPR-like framework (as in **??**), i.e., using a contrastive loss of the form

$$\sigma(f(u, i|j) - f(u, i'|j)), \tag{6.23}$$

where i' is a sampled negative item that the user did not consume.

Experiments compare two variants of FPMC. Regular Matrix Factorization (MF), includes only the first term in Equation (6.22); the other, Factorized Markov Chains, includes only the second. That is, they model $f(u, i) = \gamma_u \cdot \gamma_i$ and $f(i|j) = \gamma_i \cdot \gamma_j$. These experiments thus measure (albeit on a specific dataset) the extent to which future actions can be explained by the previous action, versus overall historital preferences.

Ultimately FPMC outperforms both variants, though interestingly FMC and MF outperform each other under different conditions. Importantly, FMC is particularly effective in *sparse* settings (i.e., few interactions per user/item) whereas MF works better on dense data.

6.5.2 Socially-Aware Sequential Recommendation

Just as we saw how Bayesian Personalied Ranking can be augmented by sampling signals from social interactions in **??**, sequential models like FPMC can also be improved by leveraging social information.

Socially-Aware Personalized Markov Chains (SPMC) Cai et al. (2017) extend FPMC (and Social BPR) by merging both temporal and social signals. The basic idea is to extend Equation (6.22) from FPMC based on the reasoning that the user's next item should be similar to those that their friends have recently consumed:

$$f(u,i|j) = \gamma_u^{(ui)} \cdot \gamma_i^{(iu)} + \gamma_i^{(ij)} \cdot \gamma_j^{(ji)} + |\mathcal{S}|^{-\alpha} \sum_{(v,k)\in\mathcal{S}} \overbrace{\sigma(\gamma_u^{(uv)} \cdot \gamma_v^{(uv)})}^{\text{similarity between } u \text{ and } v} \underbrace{(\gamma_i^{(ik)} \cdot \gamma_k^{(ik)})}_{\text{next item's compatibility with friend's previous item}}. \quad (6.24)$$

Here the set $\mathcal{S}$ consists of only the most recent interactions k by each of u's friends v. The first time inside the summation $\sigma(\gamma_u^{(uv)} \cdot \gamma_v^{(uv)})$ measures the similarity between u and v, so that we only consider the effect of social influence if u and v are sufficiently similar. The term $\mathcal{S}^{-\alpha}$ normalizes the expression so that social influence does not saturate the other terms for users with a large number of friends. Note that there is no term $\gamma^{(vu)}$ or $\gamma^{(ki)}$ in Equation (6.24) (i.e., only one set of representations is learned, rather than asymmetric representations as in Equation (6.22)); this is done simply to reduce the number of model parameters. Cai et al. (2017) show that including both temporal and social terms improves predictive performance over FPMC and Social BPR.

6.5.3 Locality-Based Sequential Recommendation

In **??**, we discussed how different aggregation functions (besides the inner product) could be useful in various contexts. This proves to be the case in various sequential recommendation settings, where sequential actions follow some notion of *locality*.

In Feng et al. (2015), sequential recommendation schemes were studied in the setting of Point-of-Interest recommendation. In such a setting, the context of the previous action is particularly informative, since the following action is likely to be (geographically) close.

If the actual semantics of the problem demand some notion of locality, then arguably similarity in the latent space should also be based on locality (rather than, say, an inner product).

The framework of Feng et al. (2015), *Personalized Ranking Metric Embedding* (PRME) models transition probabilities using an expression of the form:

$$f(i|j) = -d(\gamma_i - \gamma_j)^2 = -||\gamma_i - \gamma_j||_2^2. \tag{6.25}$$

Note two differences between this model (PRME) and FPMC (Section 6.5.1):

- The main difference is the use a distance (actually a squared distance) function, so that sequential activities exhibit locality in the latent space.
- Unlike FPMC, which used separate latent spaces $\gamma^{(ij)}$ and $\gamma^{(ji)}$ for the next and previous item, PRME uses only a single latent space (which saves parameters).

Like FPMC, PRME also includes an expression encoding the compatibility between the user and the item (again using a distance function), and also trains the model using a BPR-like framework (i.e., including a negative item i' as in **??**). Other specific details include an explicit feature encoding geographical distance (based on latitude and longitude), and a temporal feature which down-weights the influence of the sequential term if sequential events are temporally far apart.

Though the authors of PRME argue that Euclidean distance is a more natural way of comparing sequential items (and show that PRME outperforms FPMC for POI recommendation), it should be noted that whether one similarity function is 'better' than the other largely depends on the semantics of the specific problem and dataset.

Another paper makes use of a similar model for personalized playlist generation Chen et al. (2012). Like PRME, their model (Factorized Markov Embeddings, or FME) notes that sequential songs in playlists tend to be highly localized, such that a metric embedding is possibly well-motivated. The compatibility function given a user u, song i, and previous song j takes the form

$$f(i|u, j) = -d(\gamma_i^{(\text{start})} - \gamma_j^{(\text{end})})^2 + \gamma_u \cdot \gamma_i'. \tag{6.26}$$

Note a few differences between FME and PRME:

- FME uses a separate embedding for the next song ($\gamma^{(\text{start})}$) and the previous song ($\gamma^{(\text{end})}$). The basic idea being that songs in playlists should not just be highly local, but should gradually 'transition' from one song to the next, so that the 'start point' of the next song in latent space should be similar to the 'end point' of the previous song (Figure 6.6).
- FME uses a combination of both a distance function (for compatibility with the previous item) and an inner product (for compatibility with the user) in Equation (6.26). Again, this demonstrates that the correct choice of compatibility function is highly dependent on problem semantics.

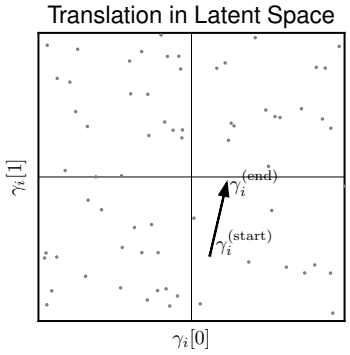


Figure 6.6: Some sequential models use the principle of translation to model sequential transitions between items.

6.5.4 Translation-Based Recommendation

A third class of models for sequential recommendation are also based on the principle of translation. He et al. (2017a) built recommender systems by borrowing ideas from knowledge-base completion.

Several techniques for knowledge base completion are based on the principle of learning low-dimensional embeddings that describe relationships among entities Bordes et al. (2013); Wang et al. (2014); Lin et al. (2015). The basic idea is to represent both entities and relationships as vectors in a low-dimensional space, such that a relation vector encodes how to 'translate' between entities. For example, we might seek to learn vectors describing entities such as *'Alan Turing'* and *'England,'* such that given a vector describing the relation *'born in'* we should have

$$d(\overrightarrow{\text{Alan Turing}} + \overrightarrow{\text{born in}}, \overrightarrow{\text{England}}) \simeq 0, \tag{6.27}$$

where d is a Euclidean distance.

Translation-based recommendation He et al. (2017a) adapts this type of approach to personalized recommendation. Whereas for knowledge graph completion relations tell us how to traverse the space of entities, in a recommendation setting, *items* fulfil the role of entities, and *users* traverse the space of items. Then, given two items j and j that are consumed in sequence, we should have

$$d(\gamma_i + \gamma_u, \gamma_j) \simeq 0. \tag{6.28}$$

Training such a model is quite similar to how we trained FME and PRME in Section 6.5.3: that is, we fit a compatibility function between a user, an item, and a previous item (much like Equation (6.26)):

$$f(j|u, i) = \beta_j - ||\gamma_i + \gamma_u - \gamma_j||_2, \tag{6.29}$$

where β_j is incorporated so that the method is capable of capturing overall item popularity as well as preference. He et al. (2017a) further constrains item representations to live on a unit ball (i.e., $\|\gamma_i\|_2^2 = 1$), which was found to be effective in the knowledge graph completion settings above.

Conceptually, the above model corresponds to users following a 'trajectory' through their interactions over time (Figure 6.6). In principle, this ought to mean that related items (e.g. sequential songs on in a playlist) should be aligned to form a chain of equally-spaced items in the latent space. In practice, such complex dynamics are unlikely to emerge from the model; rather, like other temporal modeling approaches, the model benefits from its parsimony (i.e., it has much fewer parameters than other sequential models, due to the use of only a single latent space) but is still able to capture common sequential patterns even in sparse datasets.

6.5.5 FPMC in *Tensorflow*

Although several of the above models can be implemented via an appropriately designed Factorization Machine (**??**), it is worth briefly describing how a sequential model might be implemented 'from scratch;' this will be useful when implementing variants that do not straightforwardly map to existing architectures (such as the Factorization Machine of **??**) or libraries.

Here we implement the Factorized Personalized Markov Chain (FPMC) method from Section 6.5.1, though the code can straightforwardly be adapted to implement any of the sequential methods discussed in **??**.

We build our solution on top of our Bayesian Personalized Ranking implementation from **??**. First, when parsing the data, we must be careful to process the timestamp:

```
for d in parse("goodreads_reviews_comics_graphic.json.gz"):
    u = d['user_id']
    i = d['book_id']
    t = d['date_added'] # Raw timestamp string
    r = d['rating']
    dt = dateutil.parser.parse(t) # Structured timestamp
    t = int(dt.timestamp()) # Integer timestamp
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)
    interactions.append((t,u,i,r))
    interactionsPerUser[u].append((t,i,r))
```

Note the use of the `dateutil` library to process the timestamp. The original timestamp in this dataset consists of raw strings (e.g. 'Wed Apr 03 10:10:41 -0700 2013'); the operation `dt = dateutil.parser.parse(t)` converts this

to a structured format; this can be used to extract features associated with the timestamp, e.g. `dt.weekday()` reveals that this date is a Wednesday, which might be useful for extracting features for temporal models such as those in **??**.[5] To build a sequential recommender, we are mainly interested in determining the *sequence order* of the interactions, for which we call `dt.timestamp()`; for the date above this returns 1365009041, which represents the number of seconds since January 1, 1970 ('unix time'). Such a time representation, while seemingly fairly arbitrary, is useful when our goal is simply to sort datapoints in order of time, as we do when building sequential recommenders.

Next we sort each user's history by time, and augment our interaction data such that each interaction (u, i) includes the previous item j. We also add a 'dummy' item which acts as the previous item for the first observation:

```
itemIDs['dummy'] = len(itemIDs)
interactionsWithPrevious = []

for u in interactionsPerUser:
    interactionsPerUser[u].sort()
    lastItem = 'dummy'
    for (t,i,r) in interactionsPerUser[u]:
        interactionsWithPrevious.append((t,u,i,lastItem,r))
        lastItem = i
```

Given these augmented interactions, we can modify the model from **??** to include the additional terms from **??**. Here we train in a BPR-like setting (i.e., including a sampled negative item k), though we could similarly adapt the model for rating prediction following code from **??**. Omitting a few boilerplate elements, the model equation (**??**) becomes:

```
gamma_ui = tf.nn.embedding_lookup(self.gammaUI, u)
gamma_iu = tf.nn.embedding_lookup(self.gammaIU, i)
gamma_ij = tf.nn.embedding_lookup(self.gammaIJ, i)
gamma_ji = tf.nn.embedding_lookup(self.gammaJI, j)
# (etc.)
x_uij = beta_i + tf.reduce_sum(tf.multiply(gamma_ui, gamma_iu), 1)
     +\
                tf.reduce_sum(tf.multiply(gamma_ij, gamma_ji), 1)
x_ukj = beta_k + tf.reduce_sum(tf.multiply(gamma_uk, gamma_ku), 1)
     +\
                tf.reduce_sum(tf.multiply(gamma_kj, gamma_jk), 1)
return -tf.reduce_mean(tf.math.log(tf.math.sigmoid(x_uij - x_ukj))
    )
```

The code above could be straightforwardly adapted to implement other se-

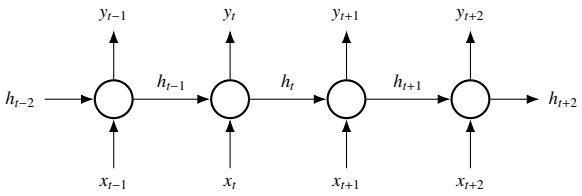
[5] While fairly obvious in this instance, determining even such simple properties is difficult for certain date formats.

quential models, such as PRME (Section 6.5.3) or translation-based recommendation (Section 6.5.4).

6.6 Recurrent Networks

A fundamental limitation of the Markov-Chain-based models we saw in Section 6.4 is that they have a very limited notion of 'memory,' due to the assumption that the next event is conditionally independent of all historical events, given the most recent observation. This assumption may be sufficient in certain scenarios, such as recommendation settings that are highly dependent on the context of the previously clicked item (for example). However, as we begin to model text data (Chapter 7, or sequence data such as heart-rate logs (Section 6.8), which have longer-term semantics (such as grammatical structures in a sentence, or even an individual's level of 'fatigue' in a heart-rate trace).

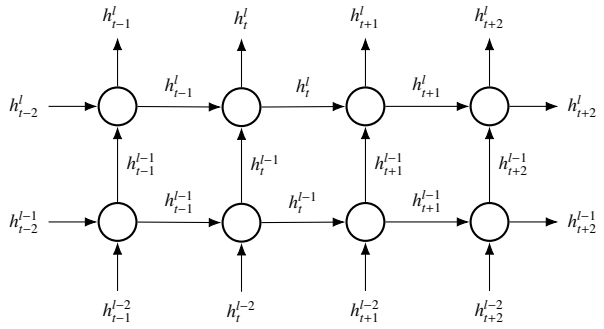
Recurrent Neural Networks (RNNs) seek to achieve this notion of 'memory' by maintaining a 'hidden state' during each step.[6] The hidden state is a vector of latent variables that somehow capture the 'context' that the model needs to know to capture the long-term semantics of the problem. Formally, we can visualize the RNN as taking a sequence of inputs $(x_1 \dots x_N)$, and producing a sequence of outputs $(y_1 \dots y_N)$, and maintaining hidden states that update at each step $(h_1 \dots h_N)$. We might visualize this model as follows:



The RNN cell, depicted here as a box, is now responsible for determining how the hidden state should change at each step, and what outputs should be generated.

More complex RNN models repeat this idea across multiple *layers*, so that RNN cells can be stacked, e.g.:

[6] Note that 'simpler' models attempt to accomplish the same goal within the framework of Markov-Chain-based models; see e.g. Hidden Markov Models. However Recurrent Neural Networks are more typical of current practice and form the basis of models we develop later.



In this depiction (and in many treatments of the topic), the model is described only in terms of hidden states: the model receives an input, updates its hidden state, and passes this state to the next timestep and the next layer. The first layer may receive inputs (i.e., observed values x) while the last layer generates outputs (i.e.,y).

When designing this cell, one might consider the types of dynamics required to model state transitions between successive steps:

- Based on the current hidden state, what output should be generated?
- How should the hidden state change as a function of the input that was just seen?
- What part of the hidden state should be preserved, and what part can be discarded?
- How can hidden state be preserved across long interaction sequences?

Below we describe one particular implementation of an RNN cell. Although there is nothing particularly sacred about the particular model we present, it is representative of the overall design approach, in terms of capturing the features described above.

6.6.1 The Long Short Term Memory Model

The Long Short-Term Memory Model **?** is a specific implementation of an RNN that has been popularized especially for use in text generation tasks (as we'll discuss in Chapter 7).

A challenge in designing RNN cells as described above is how to encourage them to 'remember' state across long sequences. To achieve this, the LSTM cell (Figure 6.7) preserves the cell state (c in the equation below), mostly unmodified across steps. Other components are responsible for 'forgetting' part of the state (f), as a function of the current input and previous hidden state;

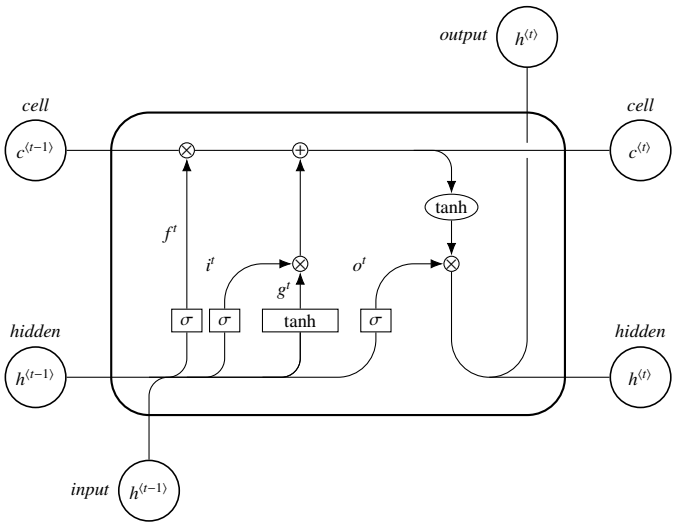


Figure 6.7 Caption

updating the cell state (i and g); updating the hidden state (h); and determining what to output (o). Although not particularly relevant for the current discussion—the models discussed here could easily be replaced by different architectures—the specific form of these components in an LSTM cell is as follows:

$$f_l^t = \sigma(W_l^{(f)} \times [h_{l-1}^t; h_l^{t-1}]) \tag{6.30}$$
$$i_l^t = \sigma(W_l^{(i)} \times [h_{l-1}^t; h_l^{t-1}]) \tag{6.31}$$
$$o_l^t = \sigma(W_l^{(o)} \times [h_{l-1}^t; h_l^{t-1}]) \tag{6.32}$$
$$g_l^t = \tanh(W_l^{(g)} \times [h_{l-1}^t; h_l^{t-1}]) \tag{6.33}$$
$$c_l^t = g_l^t \odot i_l^t + c_l^{t-1} \odot f_l^t \tag{6.34}$$
$$h_l^t = \tanh(c_l^t) \odot o_l^t. \tag{6.35}$$

Several variants of the above have been proposed to incorporate additional components, mostly consisting of specific differences to how the state is preserved and transformed among the equations above. It should be remarked though that the overall view of a 'cell,' in terms of input, outputs, and hidden state, is largely interchangeable across models in spite of differences in specific details.

Table 6.2 *Summary of user-free recommendation models.*

Method	Reference	Description
Sparse Linear Methods (SLIM)	Ning and Karypis (2011)	Each user is associated with a linear model weighting their interactions over past items; sparsity-inducing regularizers are used to deal with the large number of model parameters.
Factored Item Similarity Models (FISM)	Kabbur et al. (2013)	Replaces the user term in a latent factor model with a second term that represents the user by averaging over item representations from their history.
item2vec	**?**	

6.7 'User-Free' Sequential Models

When introducing sequential models based on Markov Chains in **??**, we argued that it was valuable to model *both* compatibility between the user and the item, as well as the compatibility between adjacent items in a sequence. In short, both sources provide useful and complementary signals that each explain variation in users' activities.

However it is unclear whether this argument applies in the limit. Clearly, a single previous interaction does not provide the full context of a user's preferences; however, a model that includes *enough* historical interactions can possibly capture all of the necessary context without explicitly including any user term.

Below we describe several attempts to model users' historical interaction sequences without explicitly including a user term. Doing so is conceptually appealing as an alternative to the cold-start models we discussed in **??**: by eliminating the need for an explicit user term, we can quickly make effective recommendations by observing a few user actions, without needing to retrain the model (to fit user terms) or rely on side-information.

As we see below (**??**), this type of 'user-free' sequential model is a growing trend in deep learning-based recommendation. Borrowing ideas from natural language processing, recent sequential models are able to model dynamics across long user histories without needing an explicit user term.

Table 6.3 *Summary of deep-learning based sequential models.*

Method	Reference	Description
?	Hidasi et al. (2015)	Item sequences are passed to a recurrent network; the network's hidden state is used to predict the next interaction.
Neural Attentive Recommendation (NARM)	Li et al. (2017)	Combines an RNN Hidasi et al. (2015), but with the addition of an attention mechanism that operates on the sequence of network latent states in order to 'focus' on relevant interactions.
Self-Attentive Sequential Recommendation	Kang and McAuley (2018)	Similar to NARM, but uses the principle of *self-attention* (i.e., a Transformer model) rather than a recurrent network.
BERT4Rec	Sun et al. (2019)	Also uses the principle of self-attention, though with a different architecture (based on BERT Devlin et al. (2018)).

6.7.1 Sparse Linear Methods (SLIM)

A direct way to avoid including an explicit user term (i.e., $gamma_u$) is to describe all of a user's interactions in terms of a binary feature vector enumerating which items they have interacted with (i.e., a vector of length $|\mathcal{I}|$). To predict the score associated with an item i, we can then train a linear model (again with $|\mathcal{I}|$ parameters), much as we did in Chapter 3:

$$f(u, i) = R_u \cdot W_i \tag{6.36}$$

Here R_u is a (sparse) vector describing all of a user's interactions, i.e., equivalent to a row of the interaction matrix R from **??**.

Fitting such a model naïvely is not straightforward, given the high dimensional feature and parameter vectors. Ning and Karypis (2011) attempts to fit this type of model by exploiting the specific sparsity structure of the vector R_u, noting that Equation (6.36) can be rewritten in terms of just the items I_u that the user has interacted with:

$$f(u, i) = \sum_{j \in I_u} R_{u,i} W_{i,j}. \tag{6.37}$$

Here W is an $|I| \times |I|$ parameter matrix which essentially measures item-to-item compatibilities (or similarities).

Conceptually, Equation (6.37) is similar to the simple heuristic we developed in **??**, in which we predicted a rating using a weighted average of previous ratings, where the weighting function was determined by an item-to-item similarity measure (such as the Cosine similarity). Essentially, SLIM follows the same reasoning, but replaces the heuristic item-to-item similarity of **??** with a learned matrix W.

The main challenge in fitting W is that it has dimension $|\mathcal{I}| \times |\mathcal{I}|$. Were W a dense matrix (i.e., every item interacts with every item) training and inference would be expensive; this is circumvented by using a regularization approach which ensures that W is sparse.[7] Sparsity is achieved via a regularization strategy which includes both an ℓ_2 and ℓ_1 regularizer:

$$\arg\min_{W} \|R - RW^T\|_2^2 + \lambda\Omega^{(2)}(W) + \lambda'\Omega^{(1)}(W) \tag{6.38}$$

$$\text{s.t.} \qquad W_{i,j} \geq 0; \quad W_{i,i} = 0. \tag{6.39}$$

Note that $\|R - RW\|_2^2$ is merely a matrix shorthand for the predictions made for all interactions (u, i) following Equation (6.37). The first constraint in Section 6.7.1 ensures that all terms in the weighting function are positive; the second constraint ($W_{i,i} = 0$) ensures that each item i's rating is predicted only based on interactions with *other* items j. $\Omega^{(1)}$ is an ℓ_1 regularizer (i.e., $\Omega^{(1)} = \sum_{i,j} |W_{i,j}|$); as we discussed in **??**, ℓ_1 regularization leads to sparsity of the matrix W.

Ning and Karypis (2011) discuss various merits of the above approach. Notable is the rapid inference time (i.e., the rate at which recommendations can be made) compared to standard recommendation approaches, and also the *long-tail* performance of the approach. For the latter, compared to (e.g.) latent-factor approaches—whose representations tend to favor whichever types of items predominate in the data, and fail to capture the dynamics of rarer items—SLIM maintains (relatively) good performance even for tail items.

6.7.2 Factored Item Similarity Models (FISM)

Factored Item Similarity Models Kabbur et al. (2013) attempt to directly apply the above reasoning, by replacing the user term γ_u in a latent factor model (**??**) with a term that aggregates item representations from a user's history.

[7] Though in practice experiments are conducted with moderate item vocabularies (e.g. $|\mathcal{I}| \simeq 50000$), and various computational tricks are used to allow for parallelization and efficient inference.

Specifically, the user term in **??** is replaced with an average over *item* terms for all items consumed by that user:

$$f(u,i) = \alpha + \beta_u + \beta_i + \frac{1}{|I_u|}\sum_{j \in I_u} \gamma'_j \cdot \gamma_i \tag{6.40}$$

(recall that I_u is the set of items consumed by user u). Note that the item term γ_i is separate from the term used to average user actions γ_j, i.e., the model learns *two* sets of latent factors per item.

Spiritually, the average $\frac{1}{|I_u|}\sum_j \gamma'_j$ fulfils the same role as γ_u, by summarizing the dimensions that are compatible with a particular user.

Kabbur et al. (2013) considers variants of Equation (6.40) for both rating prediction and ranking problems (i.e., to optimize the MSE as in **??** or the AUC as in **??**).

Kabbur et al. (2013) argues that the above approach is particularly useful in *sparse* datasets (presumably, datasets where *users* have few associated interactions, while items have several). That is, a traditional latent factor model as in **??** would struggle to meaningfully fit γ_u for a user who has only a few interactions; whereas if item histories are denser, a reasonable estimate of user preferences can be estimated by averaging over item terms. Indeed, experiments in Kabbur et al. (2013) show that the settings in which FISM are effective are closely related to dataset sparsity.

6.7.3 Neural Network-Based Approaches

A variety of modern approaches to modeling temporal and sequential dynamics of user behavior are built on top of neural network models. Although we won't present a full treatment of such approaches here (since they depend on techniques and models quite different from those covered in this book), here we outline the general directions explored by a few representative examples.

Neural network-based approaches have emerged as the state of the art for a variety of problems in natural language processing (as well as in computer vision, as we discuss in Chapter 8, and elsewhere). Techniques range from recurrent neural networks (similar to those we introduced in **??**), to recent approaches based on self-attention and transformer architectures.

Conceptually, the appeal of neural network approaches to sequential modeling is that they allow models to capture complex *syntax* or *semantics* in sequential data. So far, when developing temporal and sequential models we have mostly focused on either of (a) developing simple parametric functions to capture long-term dynamics, as we did in Section 6.2.1; or (b) modeling sequential

dynamics using the context of previous interactions, following a Markov chain-like setup, as in **??**. Both approaches have limitations: the former tends to require carefully hand-crafting models around specific dynamics, while the latter potentially captures only limited context (e.g. a single previous interaction). In principle, the techniques below aim to address these issues, by automatically uncovering both long- and short-term dynamics simultaneously, or potentially capturing complex interrelationships between items in a sequence.

Below we describe a few specific instances of models that use neural networks to model user interaction sequences. However ultimately the specific models and architectures are not particularly import: our intention here is to highlight the general approach by which state-of-the-art models, from sequence mining or natural language processing, can be adapted to model user interaction sequences.

Recurrent Networks An early paper to explore the use of recurrent networks for recommendation did so using the types of model we developed in **??**.[8]. Hidasi et al. (2015) explored the problem of *session-based* recommendation, where user interactions are divided into distinct 'sessions' (typically using some heuristic based on interaction timestamps). This setting is typical for recommenders based on recurrent networks (or more broadly, for recommenders that borrow techniques from natural language), since it allows sessions to be treated as analogous to *sentences,* i.e., short sequences of discrete tokens (items).

Conceptually, the method is fairly similar to *item2vec* (**??**). Recall that item2vec learns representations of items such that adjacent items in a sequence (e.g. items that were both clicked on during a short period) have compatible representations. Likewise, the method from Hidasi et al. (2015) seeks to pass sequences of items into a recurrent network (similar to that of **??**) such that the hidden state of the network is capable of predicting the next interaction.[9]

Note that this method (along with most of the related methods below) does not learn a user representation: rather the user 'context' is captured via the hidden state of the recurrent network. As such, these methods are a form of 'contextual personalization' in the sense that there is no user model as such. Hidasi et al. (2015) argue that the benefits of this type of approach are that they will work well in situations where long user histories are unavailable: in

[8] Specifically they used a Gated Recurrent Unit, or GRU, though spiritually the approach is similar to the LSTM-based techniques we explored in **??**

[9] The specific architecture used to achieve this passes the final hidden state into a feed-forward network (similar to those we studied in **??**), whose final layer estimates scores associated with all items; while computationally expensive, this is feasible for datasets with relatively small item vocabularies, e.g. in the low tens-of-thousands for data in Hidasi et al. (2015).

such cases, techniques like those we developed for the Netflix prize (**??**) are unreliable as effective user representations (γ_u) cannot be learned from only a few interactions; Hidasi et al. (2015) suggests the scenario of recommendation on a niche e-Commerce platform. This possible benefit of 'user-free' sequence modeling approaches is put into more context in our broader discussion of the benefits of deep learning-based recommendation in **??**.

Attention Mechanisms The main argument in favor of recurrent network-based approaches is that they can potentially capture longer-term sequential semantics compared to (e.g.) the Markov chain-based approaches we studied in **??**. On the other and, the fact that simple Markov chain-based models are so effective in the first place suggests that even the context of a single recent item might be enough to capture a user's 'context' in many cases.

The basic intuition behind incorporating *attention mechanisms* into sequential recommender systems is to help the model to 'focus' on a small set of interactions among a relatively longer interaction sequence. Intuitively, this ought to allow the model to capture context from long interaction sequences, while still leveraging the fact that the *relevant* part of the context may only consist of a few interactions Kang and McAuley (2018). Attention mechanisms have been used in other settings such as image captioning Xu et al. (2015) and machine translation Bahdanau et al. (2014) (among others), where a small component of the input is 'attended on' when generating part of the output.

Neural Attentive Recommendation Neural Attentive Recommendation (NARM) Li et al. (2017) essentially seeks to extend and combine ideas recurrent recommendation with the use of an attention mechanism. Their suggested model has two main components: a *global* encoder, which models sequential data from sessions in much the same way as a traditional RNN-based model (as described above); here the final hidden state of the RNN is responsible for (generating features to) predict the user's next action. A second component, the *local* encoder, uses an attention mechanism over *all* of the hidden states in the sequence to build a feature. Roughly speaking, the local encoder's representation captures those specific actions in the user's recent history that best capture the user's intent.

Self-Attentive Sequential Recommendation The approaches above essentially treat attention mechanisms as an *additional* component to extend existing recurrent models (i.e., NARM uses an attention mechanism to learn features from the latent representations from an RNN). A recent trend in natural language processing has been to rely more heavily on attention modules to capture

complex structures in sentences, again with the attention mechanism retrieving relevant words in a source sentence to generate words in a target sentence (e.g. for machine translation). Recently, attention-based sequence-to-sequence methods, especially the *Transfomer* model Vaswani et al. (2017), have emerged as the state-of-the-art for various general-purpose language modeling tasks. The overall goal of such models is much the same as we saw when developing recurrent networks in **??**: to estimate an output sequence on the basis of an input sequence (or in the case of modeling interaction sequences, estimating the next item in the sequence based on the previous items). Whereas the recurrent networks we developed in **??** did so by successively passing and modifying a latent state across across network cells, the Transformer architecture eschews this, relying purely on attention modules, whereby the estimation of the next item in a sequence is a function of all previous items,[10] with attention determining which of the previous items is *relevant* to the next prediction.[11]

Kang and McAuley (2018) sought to apply the principle of self-attention to sequential recommendation problems; the method is a fairly straightforward adaptation of the transformer architecture, whereby the next item in a sequence of user interactions is predicted on the basis of the previous items, with an attention module being responsible for determining which of the previous interactions are relevant to predicting the next one.

Other than the performance benefits of Transformer-based models (compared to recurrent networks and Markov chain-based models), Kang and McAuley (2018) attempt to visualize the attention weights of their model in order to understand which previous interactions the model 'attends on' when predicting the next item in a sequence. Not surprisingly they find that the most relevant item tends to be the previous item in the sequence, though interestingly, significant weight is given to more distant interactions. On some datasets these weights decay rapidly, while on others (such as *MovieLens*), relevance weights are more widely distributed across several historical interactions; collectively this suggests that while recommendations are subject to sequential context, this 'context' requires more than a single previous observation to capture.

BERT4Rec While Kang and McAuley (2018) is based on a 'standard' Transformer implementation, various extensions have been made to this architecture. For example, BERT4Rec Sun et al. (2019) adapts BERT Devlin et al. (2018), another Transformer-based model architecture. Without going into detail, this

[10] Up to some fixed maximum length.

[11] This description of the Transformer architecture is admittedly very rough, though a full description is quite involved; see the original paper Vaswani et al. (2017) or one of many excellent tutorials that attempt to distill the core idea.

model is largely an architectural modification to the approach above; again their superior performance in language modeling tasks appears to translate well to sequential recommendation problems. Overall this (and other related approaches) further highlight the general trend of applying state-of-the-art language models to build sequential recommenders.

Attentional Factorization Machines We also note that attention mechanisms have been applied in the context of traditional recommender systems. Attentional Factorization Machines Xiao et al. (2017) apply attention mechanisms to factorization machines, as introduced in Chapter 5 (recall that such models are not necessarily sequential, though can encode sequential features). The application of attention mechanisms is fairly straightforward. Recall from Chapter 5 that factorization machines aggregate pairwise interactions across (latent representations of) all features, as in **??**. Xiao et al. (2017) uses attention to determine which pairwise interactions are most important in a particular context (and arguably helps the model not to focus on irrelevant interactions). Experiments show improvements over traditional factorization machines, and some of the deep learning-based recommendation approaches studied in **??**.

Summary of Neural Network-Based Sequential Recommenders Above we have only given a limited sample of recommendation methods based on recurrent networks and attention mechanisms. Although we have provided little detail about the specifics of each approach, we have hopefully highlighted the general trend of borrowing models from natural language and repurposing them for recommendation. The use of general-purpose language and sequence models for recommendation is increasingly representative of the current state-of-the-art; we'll further visit this idea as we explore additional language modeling approaches in Chapter 7.

Note also that the above models in general have no explicit user representation: the input to the model is a sequence of items, based on which the next item is predicted. As such they can be thought of as contextual or memory-based recommendation (though they do have *item* representations which are analogous to γ_i in traditional models). Extending the analogy to natural language, a sequence of items is essentially a 'sentence' made up of discrete tokens. In principle, so long as long enough sequences are modeled, we can still capture the broad historical interests of the user, without a need for an explicit user term. In many cases the lack of a user term is a virtue of this type of model: when dealing with new users, we can make reasonable predictions based on a few sequential actions, without needing to resort to complex cold-start approaches, or retraining the model (though of course, attempts have been made

to to explicitly incorporate user terms into such sequential models, see e.g. Wu et al. (2020)).

We will further revisit neural network based approaches and various points throughout the book. In Chapter 7 we consider neural network approaches to modeling text (including for text generation), following the same types of approaches we developed in **??**. We also consider text representation approaches in **??**, which can in turn be used to develop *item* representations for use in sequential recommendation (**??**). In Chapter 8 we explore convolutional neural network approaches for image representation and generation

6.8 Case Study: Personalized Heart-Rate Modeling

Other than building personalized models of events as in **??**, personalized sequence models can also be used to model various other types of sequence data. In **??** we'll explore using sequence models for personalized text generation; here we briefly examine how such models can be used to model heart-rate sequences.

Ni et al. (2019b) explored using personalized sequence models to estimate users' heart-rate profiles as they exercised using data from *EndoMondo*. That is, given the GPS route (latitude, longitude, and altitude) that a user intends to run (or walk, cycle, etc.), and a historical record of the user's previous activities, the goal of the model is to forecast the user's heart-rate profile as accurately as possible.

Modeling the dynamics of heart-rate sequences is particularly challenging for a few reasons, including:

- The dynamics are noisy, complex, and are highly dependent on external factors (e.g. the user's running speed, the altitude, etc.).
- These factors are a combination of both long- and short-term dynamics, e.g. the user gradually becoming fatigued, versus the user encountering a hill.
- There is significant variation among individuals, to the point that a *non*-personalized model would be ineffective for the task.
- A large amount of training data is likely required to capture the complex factors involved in heart-rate dynamics. On the other hand, very few observations are likely to be available for each individual.

The model used by Ni et al. (2019b) is fairly similar to the personalized language models we studied in **??**, where a recurrent neural network (an LSTM

in Ni et al. (2019b)) is augmented with low-dimensional embeddings capturing user characteristics, and contextual features describing the current activity: these embeddings are essentially analogous to user/item embeddings γ_u and γ_i. The argued benefit of low-dimensional user representations in this context is that one can learn a 'global' model to capture complex heart-rate dynamics from a large amount of data, along with a small number of user-specific parameters that allow the model to adapt to specific user characteristics from a limited number of observations. Again this is similar to personalized language models, where a high-dimensional language model is adapted via low-dimensional user and item terms.

In addition to user and contextual features, the recurrent network in Ni et al. (2019b) takes as input the GPS trace (latitude, longitude, altitude) of the route the user intends to complete. This sequence of variables is passed as an additional input to the model during each network step, so that the goal is to forecast heart-rate profiles as a function of the user, contextual variables (weather, time, etc.) and the intended route.

Ni et al. (2019b) show the benefit of such a model, over non-personalized alternatives and various alternative model architectures. Ultimately they argue that a system that can accurately make such forecasts can be used in various ways, e.g.

- To recommend routes that will help a user to achieve a desired heart-rate profile.
- To recommend alternative routes that are semantically similar (in terms of heart-rate dynamics) to those they normally take, e.g. if they want to maintain their routine while visiting a new city.
- To detect anomalies in the event that a user's heart-rate should significantly exceed the projected rate at any point.

6.9 History of Personalized Temporal Models

Although the *Netflix Prize* was one of the early drivers for the use of temporal models in recommendation scenarios, the basic idea of *concept drift*, in which the distribution of labels changes over time, dates back significantly further. For example, the *Dynamic Weighted Majority* algorithm Kolter and Maloof (2007) considers using an ensemble of classifiers ('experts'); the weighting of these classifiers' decisions changes over time as a function of the empirical performance of the classifiers. Kolter and Maloof (2007) also gives a historical

perspective on the problem of concept drift, dating back to early papers on the topic Schlimmer and Granger (1986).

While Koren (2009) showed the effectiveness of temporal factors on the Netflix data, a few earlier efforts to use temporal dynamics in recommendation scenarios are noted. For example, Sugiyama et al. (2004) considered the problem of *personalized search*, though their solution to this problem is actually a form of recommender system. Theirs is an interesting case-study in the use of simple similarity-based methods (like those of Equation (4.19) and Equation (4.22)) for a problem (personalized search) that doesn't immediately appear to be a recommender systems problem. In terms of temporal dynamics, the main idea is to distinguish between 'persistent preferences' and 'ephemeral prefernces' when building user profiles (from interaction histories), though this is mostly achieved by considering interactions within differently sized recency windows.

Following the success of modeling temporal dynamics on Netflix, there has been a proliferation of models that attempt to capture temporal dynamics in a variety of settings. As we described in **??**, temporal dynamics my occur for a variety of reasons, ranging from long- and short-term dynamics Xiang et al. (2010), community effects community Godes and Silva (2012), or

users McAuley and Leskovec (2013b)

6.10 Exercises

Exercises

6.1 Just as the autoregressive and sliding window techniques we introduced in **??** can be used for prediction, they can also be used for *anomaly detection*, that is, to determine which events in a sequence substantially deviate from our expectations

6.2 Our approaches to similarity-based recommendation (write something that uses a similarity-based recommender but considers both user-item and item-to-item similarity)

6.3 In **??**, we developed a model that included user and item latent factors γ_u and γ_i. Adapt the model so that instead of modeling the user, it models the previous item (γ_j)

6.4 In **??** we introduced Factorization Machines as a technique to incorporate various types of features into recommendation approaches (and discussed libraries to do so in **??**). When introducing Factorized Personalized Markov Chains in Section 6.5.1 we argued that this type of model

can be represented via a factorization machine. Essentially, this can be done by concatenating representations for the item, the user (as in **??**, and the previous item. Following this approach, using any interaction dataset, compare models that use (a) only the sequential term $\gamma_i \cdot \gamma_j$; (b) only the preference term $\gamma_u \cdot \gamma_i$; and (c) both (i.e., FPMC).

6.5 Likewise, FISM (**??**) can be implemented via a factorization machine. Here, the user term is replaced by a feature representing the items the user has consumed (divided by the number of interactions). When extracting a feature based on the interaction history, be sure to exclude the current interaction.

6.6 *Herding* effects occur when users' decisions (e.g. their ratings) are biased by those they've already seen: e.g. a user may enter a high rating for an item simply because they see that ratings are already high.

6.10.1 Project 4: Temporal and Sequential Dynamics in Business Reviews

In this project we'll explore the various notions of temporal dynamics we covered in this chapter, and conduct a comparative study to determine which notions work well in a particular setting. Specifically, we'll consider consumption patterns of business reviews on *Google Local*, which has been used in various studies on temporal dynamics for recommendation (see e.g. He et al. (2017a)). Business reviews are useful for the study of temporal dynamics, as activities have a combination of short-term, long-term, and sequential dynamics, though in principle this project could be conducted using any dataset that includes temporal information.

For most of the components in this project, you can base your implementation on the *factorization machine* framework we covered in **??** (e.g. using the fastFM library as in **??**). Others which don't follow this framework, such as models based on metric embeddings, are significantly more challenging to implement, and may require solutions based on gradient descent.

We'll explore this problem via the following steps:

(i) Start by training a (non-temporal) latent factor model for the task. Note that this problem could be cast either as one of rating prediction (as in **??**) or as visit prediction (as in **??**). Use this initial model to set up your learning pipeline and to find good initial values in terms of the number of latent factors, regularizers, etc. Also consider how you might pre-process the dataset, e.g. is it better to consier all businesses, or to consider only businesses within a specific category (such as restaurants)? If your setting

requires negative samples (e.g. you are modeling visit prediction), consider how you will sample timestamps for the negative items.[12]

(ii) Next, we'll incorporate temporal dynamics into our model via simple features. Consider how temporal dynamics might be at play in each of these settings, e.g. ratings might vary on certain days of the week, or be more positive for new businesses, etc. Process the timestamps associated with each activity to extract the day of the week, the month of the year, and the 'absolute' timestamp (which you may want to scale e.g. for the range [0, 1] to represent the lifetime of the dataset). Consider any other temporal features that might be useful in this dataset, e.g. following ideas from Section 6.2.1.

(iii) Alternately, try including features based on sequential dynamics, e.g. following the techniques from Exercises **??**. Again you may use simple features (e.g. a one-hot encoding of the previous item, as in **??**), or more complex ones (such as including several recent items, weighting in terms of recency or geographical distance, etc.).

(iv) (Hard) Try to implement an alternate sequential model, such as one based on recurrent neural networks from **??**. Although challenging to implement 'from scratch,' several of the cited papers have public code repositories on which you may base your implementation.

Although the main goal of the project is to build familiarity with and compare various temporal modeling techniques, consider also whether the temporal models you develop give you *insight* into the underlying dynamics of the data. For example, under what circumstances are users guided by temporal dynamics? Or what types of businesses are most subject to seasonal trends?

[12] For instance, you could duplicate a timestamp from a positive instance from the same user.

EMERGING DIRECTIONS IN PERSONALIZED MACHINE LEARNING

7

Personalized Models of Text

Throughout this book we have already worked with a wide variety of datasets involving text. Although we have made little use of this data so far, text can be useful both as a feature to improve the predictive performance of the types of method we've seen so far, but can also be used to explore a variety of new applications.

As a feature to improve prediction, effectively making use of text is not straightforward. Text is noisy, varies in length, has complex syntax, and only a small fraction of words may be important to prediction. As such, we start this chapter with a primer on the types of feature engineering techniques that can be used to extract meaningful information from text (Section 7.1).

Following this, we explore how text can be used to improve the types of personalized models we've developed in previous chapters. In the case of recommender systems, text ought to be helpful, as there is abundant text (e.g. product reviews) that can help us to 'explain' the underlying dimensions of users' preferences and items' properties; however effectively extracting these signals is not straightforward (**??**).

As a form of sequential data, models from natural language processing can also guide us to better ways to model sequences, following the material we developed in Chapter 6. After introducing a selection of models of natural language, we explore techniques that borrow the same ideas for sequential recommendation (Section 7.2.1).

Having explored methods that use text as a feature to improve prediction, we also explore recent trends in natural language *generation*. The proliferation of language models based on recurrent networks, along with a slew of recent architectures for general-purpose language modeling and generation, have opened up a range of applications ranging from goal-oriented dialog Bordes et al. (2016) to story Roemmele (2016) or poetry Zhang and Lapata (2014) generation. Naturally, such methods benefit from personalization, to better cap-

ture the context, preferences, or characteristics of individual users Joshi et al. (2017); Majumder et al. (2020). We'll explore such settings in the context of recommendation approaches that generate text to 'explain' recommendations to users (**??**), and systems that generate recommendation via natural language conversations with users (**??**).

Outside of recommendation, we also examine the use of text in other personalized or contextual settings, from simple forms of personalized retrieval (**??**), to complex systems such as Google's *Smart Reply* (**??**).

7.1 Basics of Text Modeling: The Bag-of-Words Model

Many of the datasets we've already been working with contain natural language data. Several tasks can potentially benefit from from this data when making predictions, and as we'll see later, text can drive a variety of unique applications.

7.1.1 Sentiment Analysis

To understand why modeling textual data is difficult, let's consider the seemingly simple task of predicting a rating based on the text of a review:

$$\text{rating} = X^{(\text{review})} \cdot \theta \tag{7.1}$$

This is the same as the type of problem we set up in Chapter 3, except that our features X are derived from the reviews. Intuitively it makes sense that reviews should help us predict ratings, as they are specifically intended to explain a user's rating.

The task described in Equation (7.1) captures the basic setting of *sentiment analysis*, i.e., learning what types of features are associated with 'positive' (i.e., high ratings) and 'negative' language. The main challenge is how to appropriately extract meaningful features from text.

The first problem that we must deal with is that the features in Equation (7.1) are *fixed length* (i.e., X is a matrix), whereas text is sequence data. Later, we'll see how to model sequences more directly in **??**, but for now let's see how much progress we can make just by extracting a pre-defined set of features from each review.

Bag-of-words models attempt to solve this by composing X out of features that encode the presence or absence of certain words in a model. Thus it ignores key details such as the order in which words appear (see **??**).

A key component in the bag-of-words model is the *dictionary* that is used to

Figure 7.1 What's the point of sentiment analysis?
Sentiment analysis, viewed superficially, may seem like an odd (or a 'toy') task: why would we ever want to predict ratings from reviews, given that in practice we never observe a review *without* a rating? Nevertheless, sentiment analysis is one of the core topics in natural language processing whose importance transcends the immediate task of predicting ratings. Sentiment analysis research is generally focused on:

- Understanding the sociolinguistic dimensions of sentiment, rather than the immediate task of predicting the rating.
- Building sentiment models that are *general purpose*, i.e., models that can be trained on one corpus (e.g. of reviews), but can be used to predict sentiment in settings where ratings aren't available.
- As a benchmarking task to test scalability, and the ability of NLP systems to understand detailed nuances in language.

Figure 7.2 Bag-of-Words models

Loved every minute. So sad there isn't another! I thought JK really made Harry an even stronger archetypal hero - almost in a Paul Maud'Dib from Dune kind of way. He's fighting the ultimate evil, he's brave and takes risks, and believes in himself and doesn't give up despite many hardships.	risks, Paul and hardships. believes the almost sad ultimate kind up every - an there in brave hero I fighting Dune another! way. himself made really he's despite He's Loved from archetypal minute. and a Maud'Dib isn't even evil, of in many stronger So takes JK thought Harry give and doesn't

Figure 7.3 The two reviews above have identical *bag-of-words* representations (the second randomly shuffles the words of the first). The review at right misses details that depend on syntax. Is there still enough information in the review at right to tell whether the overall sentiment is positive?

build our feature vector. Our first attempt to build this dictionary might merely consist of compiling *every* word in a given corpus, e.g.:

```
1 wordCount = defaultdict(int)
2 for d in data:
3   for w in d['review/text'].split():
4     wordCount[w] += 1
5
6 print(len(wordCount))
```

Doing so on just the first 5,000 of our beer reviews reveals a total of *36,225* unique words. In other words, each review (on average) contains around seven previously unseen words. This number sounds large, but we might think that by the time we have seen 5,000 reviews that we have 'saturated' the vocabulary of English words, and will not see too many more. However, if we repeat the same experiment for (e.g.) 10,000 reviews, we obtain 55,699 unique words—not

quite double, but still a substantial increase, revealing that we do not quickly saturate the vocabulary.

Looking at a few of the actual words in our dictionary, we see words like:

...the; 09:26-T04.; Hopsicle; beery; #42; $10.65; (maybe; (etc.)

That is we see words including proper nouns, unusual spelling, prices, punctuation, etc. From this, we quickly see that we will not quickly run out of unique 'words,' given any reasonable number of reviews.

To use a bag-of-words representation, we will need to reduce this dictionary to a manageable size. Some potential steps to do so include:

Removing capitalization and punctuation Removing punctuation will reduce our dictionary size substantially, as words like '(maybe' (i.e., a word following a parenthesis) will be resolved to a common word. Likewise, we might ignore different capitilization patterns simply by converting all documents to lower-case.

Stemming Stemming, i.e., resolving similar words to their common word stems. Words like 'drink,' 'drinking,' and 'drinks' would all map to 'drink'.[1] Such techniques are mostly motivated by search and retrieval settings (i.e., to make sure results are retrieved even if a query uses a different word inflection than the result), though they could also be used to reduce our dictionary size. See e.g. Lovins (1968); Porter *et al.* (1980) for examples of stemming algorithms.

Stopwords Stopwords are common English words that likely carry (relatively) little predictive power compared to their frequency in a document. Standard stopword lists[2] include words such as 'am,' 'is,' 'the,' 'them,' etc. Removing them can reduce our dictionary size a little, or otherwise prevent our feature representations from being overwhelmed by common words (though we will see other ways to address this in Section 7.1.3).

Decisions like whether to remove punctuation, whether to stem, or whether to remove stopwords, are largely dataset and application dependent. Characters like exclamation points could be predictive of sentiment;[3] or different word inflections (such as 'drinks' or 'drinker' in a corpus of beer reviews) may have

[1] Or words like 'argue,' 'arguing,' and 'argus' would map to 'argu'—the stem need not be an actual word.

[2] See e.g. `nltk.corpus.stopwords` in Python.

[3] Often, important punctuation characters are preserved by treating them like separate words, e.g. the string 'great!' would be replaced by 'great !'.

different meanings; or stopwords like 'i' and 'her' could change the meaning of a sentence.

As such, the procedures above essentially amount to feature engineering choices: we should ultimately accept or reject the above procedures based on whether they improve performance for our given application (again, these are the kind of choices we'd make with our validation set as in **??**).

For the moment, let's consider removing punctuation and capitalization, e.g.:

```
for d in data:
  r = ''.join([c for c in d['review/text'].lower() if not c in
      string.punctuation])
  for w in r.split():
    wordCount[w] += 1
```

After removing them, we are left with 19,426 unique words. This is a reduction of nearly half compared to what we had before removing them, but is still a fairly large dictionary size.

A more straightforward way to reduce our dictionary to a manageable size is simply to include only the most commonly occurring words:

```
counts = [(wordCount[w], w) for w in wordCount]
counts.sort()
counts.reverse()

words = [x[1] for x in counts[:1000]]
```

Although perhaps not a completely satisfactory solution, this representation at least allows us to build a feature representation. A simple bag-of-words-based sentiment analysis model would now consist of predicting:

$$\text{rating} = \theta_0 + \sum_{w \in \mathcal{D}} \text{count}(w) \cdot \theta_w. \tag{7.2}$$

Here $\mathcal{D}$ is our *dictionary* (i.e., our set of words); θ is indexed by a word w, but in practice we would replace each word by an index (here from 1 to 1000) for the sake of building a feature matrix.

Fitting such a model on our beer review dataset[4] yields a training MSE of 0.27 and a testing MSE of 0.51; this attests to both the high accuracy of such models, as well as their ability to overfit.

Examining the coefficients θ, we find the five words associated with the most

[4] Here with an ℓ_2 regularizer with coefficient $\lambda = 1$, 4,000 data points for training and 1,000 for testing.

negative coeficients are:

$$\begin{aligned}\theta_{\text{skunk}} &= -0.364;\\ \theta_{\text{oh}} &= -0.312;\\ \theta_{\text{skunky}} &= -0.292;\\ \theta_{\text{bland}} &= -0.284;\\ \theta_{\text{recommend}} &= -0.267,\end{aligned}$$

and the five most positive words are:

$$\begin{aligned}\theta_{\text{raisins}} &= 0.204;\\ \theta_{\text{impressed}} &= 0.224;\\ \theta_{\text{keeps}} &= 0.234;\\ \theta_{\text{always}} &= 0.256;\\ \theta_{\text{exceptional}} &= 0.320\end{aligned}$$

For example, every instance of the word 'skunk' (a negative association with beer) decreases our prediction of the rating by around one third of a star; likewise every instance of the word 'exceptional' increases our prediction by around the same amount. A few more observations:

- Words like 'skunk' and 'skunky' presumably convey the same information, indicating some redundancy in our representation; possibly this could be addressed by stemming.
- The word 'oh' is extremely negative, despite conveying little meaning by itself. Presumably, it occurs in phrases like 'oh no' (whereas 'no' itself appears in a wide variety of phrases so is not as negative); possibly we could account for such confounds by using n-grams (Section 7.1.2).
- Likewise words like 'always' or 'keeps' are highly positive despite conveying little sentiment in isolation; the word 'raisins' possibly appears in the context of a specific popular item.
- The word 'recommend' is highly *negative*, in spite of seeming to convey positive sentiment. Presumably, this word frequently appears in negative phrases ('would not recommend,' etc.); we could account for such confounds by better handling of negation.

Finally, it is notable that the words 'skunk' and 'exceptional' are the 962$^{\text{nd}}$ and 991$^{\text{st}}$ most popular words in our corpus—that is, they are words we nearly discarded when we selected a 1,000 word dictionary. On the one hand, this might be an argument that our dictionary size was too small, since we nearly missed the most important words. On the other hand, it is almost true by definition that

the most predictive words will be rare ones: it is, after all, quite exceptional for an item to be described as 'exceptional'!

7.1.2 N-grams

Some of the oddities with our sentiment model from the previous section possibly arise due to the simplifying assumptions made by the bag-of-words model. Critically, a bag-of-words model has no notion of *syntax* in a document, and as such cannot handle even simple concepts such as negation (e.g. 'not bad') or compound expressions that carry different meaning together than alone (e.g. 'oh no').

N-gram models attempt to address some of these issues by considering words that frequently co-occur in sequence. That is, *bigrams* consist of pairs of words that are adjacent in a document; *trigrams* consist of groups of three words that appear consecutively in a document (etc.). not great writing

For example, the N-grams associated with the following sentence would be:

Sentence:	'Dark red color, light beige foam'
Unigrams:	['Dark', 'red', 'color,', 'light', 'beige', 'foam']
Bigrams:	['Dark red', 'red color,', 'color, light', 'light beige', 'beige foam']
Trigrams:	['Dark red color,', 'red color, light', 'color, light beige', ...]
(etc.)	

From the above example we can already see some potential benefits of using an N-gram representation, e.g. several of the words in the above sentence are adjectives that modify the words 'color' and 'foam;' without an N-gram representation we might fail to correctly understand those nouns in context.[5] Similarly, negated terms (e.g. 'not bad' etc.) are readily handled by an N-gram representation.

N-grams are straightforward to extract in Python, e.g.:

```
1 sentence = "Dark␣red␣color,␣light␣beige␣foam"
2 unigrams = sentence.split()
3 bigrams = list(zip(unigrams[:-1], unigrams[1:]))
4 trigrams = list(zip(unigrams[:-2], unigrams[1:-1], unigrams[2:]))
```

Having extracted N-grams, our approach to modeling the data is much the same as that of our bag-of-words model, i.e., we extract counts associated with each N-gram, and use those counts as features to predict some outcome.

Note that generally we would not use (e.g.) bigrams exclusively, but rather

[5] Though one could argue the opposite: if an adjective like 'beige' rarely occurs in any other context, then a regular bag-of-words representation may be sufficient to capture the relevant information.

would use a representation that included both unigrams and bigrams simultanously. Much like our previous approach from Section 7.1.1, which simply found the most popular words (i.e., unigrams), we might simply count the popularity of unigrams and bigrams simultaneously.

On the same review corpus as in Section 7.1.1, the majority of popular terms are still unigrams—N-grams comprise 452 of the 1,000 most popular terms.

A few of the most popular N-grams include terms like 'with a,' 'in the,' 'of the,' etc. At first glance, these don't look particularly useful, and are mostly combinations of stopwords. Likewise, few of the top N-grams appear to include negation—for example among the top 1,000 terms, only eleven include the word 'not', including e.g. 'but not', 'not a', 'is not', 'not much', 'not too', etc. Again these seem unlikely to be informative given that they do not modify meaningful terms.

The above example highlights that N-grams are not quite a simple panacea for the issues mentioned above about handling adjectives and negation (etc.). Indeed, one must carefully trade off the fact that N-grams introduce substantial redundancy into our feature vector with the possibility that they introduce some useful compound terms. Note that it is not straightforward to get the 'best of both worlds' in this scenario: assuming a dictionary of fixed size, we are potentially discarding informative unigrams (like 'exceptional' from Section 7.1.1) in favor of less-informative N-grams. Again, these issues are model- and dataset-specific, and likely could be addressed by better accounting for stopwords (or possibly by using a larger dictionary). Mostly this example simply highlights that we must make extra considerations when incorporating N-grams into a model, and that they will not confer benefits unless carefully handled.

Some of these potential advantages and disadvantages of N-gram representations are summarized in Figure 7.4. Many of the disadvantages center around the redundancy of the representation, which can partly be mitigated by carefully selecting *important* features. We will revisit the topic of word important in Section 7.1.3

Finally, let's evaluate an N-gram representation on the same task from Section 7.1.1. Again we'll build a 'bag-of-ngrams' model by taking the 1,000 most popular N-grams (for any value of N). Once extracted our model is again the same as that from Equation (7.2), the only difference being that our dictionary consists of a combination of N-grams of different lengths.

After fitting the model, performance in fact slightly degrades compared to the model from Section 7.1.1 (testing MSE of 0.54 compared to 0.51 in Section 7.1.1). Examining some of the most predictive N-grams (i.e., largest and

Figure 7.4 Arguments For and Against N-grams.
To summarize our discussion in Section 7.1.2, N-grams are not always beneficial in language modeling tasks, partly because one has to trade of the predictive value of N-grams against the redundancy in encoding them. Some positive and negative points are summarized below:

- N-grams can straightforwardly allow us to handle negation, and various forms of compound expressions,
- Our dictionary size quickly multiplies when using N-grams. Assuming that we can handle a fixed dictionary size, in practice this sometimes means some informative unigrams will be replaced by uninformative N-grams.
- Stopwords, which made up a small fraction of our unigram dictionary, quickly multiply when building N-gram representations; thus there is additional redundancy in the N-gram representation.
- An N-gram representation may add substantial redundancy (or co-linearity) between features, e.g. as informative unigrams are now duplicated among several N-gram features.

smallest values of θ_w) we find terms such as

$$\theta_{\text{pitch black}} = -0.397; \quad \theta_{\text{ptich}} = 0.354.$$

Upon further inspection, the word 'pitch' *always* appears in the expression 'pitch black', to the point that the two terms will mostly 'cancel out'; again this highlights an issue of redundancy (and co-linearity, as in **??**) of our representation.

Possibly we can address this simply by further regularizing our model. Increasing the regularization coefficient to $\lambda = 10$ improves the performance somewhat (test MSE of 0.506), and results in more reasonable coefficients, such as:

$$\theta_{\text{wonderful}} = 0.177; \quad \theta_{\text{not bad}} = 0.174; \quad \theta_{\text{low carbonation}} = 0.137, \tag{7.3}$$

which appear to correctly handle negation ('not bad') and compound words ('low carbonation').

7.1.3 Word Relevance and Document Similarity

Suppose we wanted to build a system that recommends articles that are similar in content to ones a user had recently interacted with. As in Section 4.3, we might do so by defining an appropriate similarity metric between articles, and recommend those articles that are most similar:

$$f(i) = \arg\max_j \text{Sim}(i, j). \tag{7.4}$$

I read **this** after hearing from **a** few people that **it was** among their all-time favorites. **I was** almost put off when **I** saw **it was a story about** rabbits, originally written as **a** tale by **a** father **to** his children—but I'm glad **I** wasn't. **I** found **the** folk tales **about** El-ahrairah **to** be very impressive. **The** author clearly had **a** vivid imagination **to** create so much of **the** rabbits culture **and** history. But **I** think **this** book **was** worth reading as it's really **a story about** survival, leadership, **and** human nature. Oh **and** Fiver rocks. **And** BigWig is **the** man.

I was delighted by this book... the only fault is that it was too short! What a fantastic idea; a refuge for the children who have had adventures & now cannot fit back into the identity assigned to them. How many of us are not comfortable in the families we were born to? I loved the way the different doorways were sorted; one would think that adventures shared would be a bonding moment. Rivalries will be ever present; guess that is human nature. I don't want to describe too much & ruin the magic[...]

Figure 7.5 Term frequency and *tf-idf* comparison. At left, the top 10 words by term frequency are bolded (i.e., the most common words in the review), and top 10 tf-idf words (based on a sample of 50,000 reviews) are underlined. A highly-similar review (based on cosine similarity of *tf-idf* vectors is shown at right.

Given that our goal is to define similarity based on article *content* (rather than interaction histories as in Section 4.3), we might consider defining similarity based on the feature representations we developed in **??**. For example, we could compare the cosine similarity of two bag-of-words representations:

$$\text{Sim}(i, j) = \text{Cosine Similarity}(x_i, x_j). \tag{7.5}$$

However as we discussed in **??**, the vectors x_i and x_j will be dominated by the most common words in the corpus (i.e., the largest magnitude words will likely be stopwords).

In practice a user would probably not regard two documents as 'similar' just because they used words like 'the,' 'of,' or 'and' in similar proportions. As such, we presumably want a feature representation x_i that focuses on *relevant* terms.

In Figure 7.5 we see an example of a book review (of *Watership Down*) with the most frequently occurring words in boldface. Naturally, we would not say that the topic of this review was 'i' or 'a,' even though those words are the most frequently occurring.

Rather, we might argue that the words like 'nature' or 'children' are more characteristic of the document, presumably because they do *not* occur in most documents. As such, we might consider words characteristic of a particular document to be those that occur frequently in that document but not in others.

To capture such a definition we should separately consider word frequency in a particular document, and frequency across a corpus at large. To do so we define two terms. First, the *term frequency* of a word w in a document d is

simply the number of times that word appears in the document:

$$\text{Term Frequency}(w, d) = tf(w, d) = |\langle t \in d | t = w \rangle|. \tag{7.6}$$

Note that this is essentially the same as the Bag-of-Words representation we developed in **??** (although the latter includes a fixed dictionary size).

Next, we define the *document frequency* of a word across a set of documents in terms of a word w and a corpus $\mathcal{D}$:

$$\text{Document Frequency}(w, \mathcal{D}) = df(w, \mathcal{D}) = |\langle d \in D | w \in D|. \tag{7.7}$$

Now, for a word to be 'relevant' in a particular document, we want the term frequency $tf(w, d)$ to be high, and the frequency of the word across the entire corpus $dF(w, \mathcal{D})$ to be relatively low. The *tf-idf* measure (term frequency-inverse document frequency) is a heuristic which achieves this goal via the following function:

$$tf\text{-}idf(w, d, \mathcal{D}) = tf(w, d) \times \log_2 \left(\frac{|\mathcal{D}|}{df(w, \mathcal{D})} \right). \tag{7.8}$$

While the expression above captures our intuition that the term frequency should be high while the document frequency is relatively low, the specific expression may seem somewhat arbitrary (e.g. the inclusion of the $\log_2$ term). Indeed, this expression is merely a heuristic, as described in the original implementation Jones (1972). Later work has attempted to justify this choice, e.g. by interpreting $\log_2 \frac{|\mathcal{D}|}{df(w,\mathcal{D})}$ as a log-probability of a word appearing in a document Robertson (2004), though these are post-hoc justifications for what was ultimately a heuristic choice.

Likewise, the term frequency is also a heuristic and is often modified for use in specific contexts. For instance, two alternate definitions of the term frequency include:

$$tf'(w, d) \quad = \delta(w \in d) \tag{7.9}$$

$$tf''(w, d) = \frac{tf(w,d)}{\max_{w' \in d} tf(w',d)}. \tag{7.10}$$

Both of the above are essentially normalization schemes, intended to prevent *tf-idf* scores being higher for longer documents.

7.1.4 Using TF-IDF for Search and Retrieval

Cover BM-25?

7.2 Distributed Word, Item, and Document Representations

Word2Vec is a popular approach to developing *semantic representations* of words Mikolov et al. (2013). Such representations are somewhat analogous to the user and item representations (γ_u and γ_i) we have been studying throughout this book. That is, just as our latent item representations give us a sense of which items are 'similar to' which others (likewise for users), we would like to find latent word representations γ_w that tell us which words are similar, or are likely to appear in the same context as each other.

These types of 'distributed' word representations are potentially useful for a variety of reasons:

- Unlike bag-of-words models (**??**), distributed representations offer a natural mechanism to handle synonyms. That is, words w and v that are synonyms of each other ought to have nearby representations γ_w and γ_v, since they will tend to appear in the same contexts.
- Beyond synonyms, distributed representations might allow us better understand *relationships* among words.
- In certain settings, distributed representations allow us to avoid dimensionality issues associated with bag-of-words models. For instance, when developing generative models of text (which we'll touch on briefly in **??**), documents are typically represented as sequences of low-dimensional word vectors γ_w, rather than as vectors of (e.g.) word IDs via a bag-of-words model.

Below we briefly outline *word2vec* as described in Mikolov et al. (2013); in **??** we describe how this idea applies to learning item representations γ_i for recommendation. Although the latter and the former are essentially equivalent, the latter may feel more familiar, as it is similar to the way we learned item representations γ_i using sequence models in **??**.

Methodologically, *word2vec* seeks to model the probability that a word in a sequence w_t appears nearby words w_{t+k}, i.e., $p(w_t|w_{t+k})$. So, for a sequence of words $w_1 \dots w_T$, we would like to learn word representations that maximize the (log) probability

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} \log p(w_{t+j}|w_t). \tag{7.11}$$

Here c is the size of a *context window*, which determines how many neighboring words we consider; this is a hyperparameter that may be chosen to balance accuracy and training time, though potentially can vary depending on the word w_t. A simple way to define the probability $p(w|v)$ is to say that words w

are likely to appear near words with similar representations. In Mikolov et al. (2013) this is defined in terms of the inner product between representations:

$$p(w_o|w_i) = \frac{e^{\gamma'_{w_o} \cdot \gamma_{w_i}}}{\sum_{w \in \mathcal{W}} e^{\gamma'_w \cdot \gamma_{w_i}}} \tag{7.12}$$

where $\mathcal{W}$ is the set of words in the dictionary. The numerator in the above encodes the compatibility between the 'input' and 'output' words w_i and w_o; the denominator simply normalizes the value so that it corresponds to a probability (much as we saw in **??**).

Note also that we learn *two* representations, γ_w, and γ'_w for each word (referred to as the 'input' and 'output' representation, respectively). Although doing so doubles the number of parameters, this type of representation avoids symmetries, for example a word is not likely to appear near itself. This is similar to the idea we saw when developing item-to-item recommender systems, and sequential recommender systems in **??**, again the intuition being that an item is not likely to be co-purchased with itself.

Since the denominator in Equation (7.12) requires normalizing across all words in the dictionary $\mathcal{W}$, it is not efficient to compute. Mikolov et al. (2013) suggests a few schemes to overcome this issue, though the most straightforward is simply to sample a small number of 'negative' words, rather than normalizing over the whole dictionary. As such each computation of $p(w_o|w_i)$ is replaced by an approximation:

$$\log p(w_o|w_i) \simeq \sigma(\gamma'_{w_o} \cdot \gamma_{w_i}) + \sum_{w \in \mathcal{N}} \log \sigma(-\gamma'_w \cdot \gamma_{w_i}) \tag{7.13}$$

where $\mathcal{N}$ is a sampled set of negative words. Mikolov et al. (2013) proposes various schemes for choosing the sample $\mathcal{N}$, though most critically argues that the sampling probability should be proportional to the overall frequency of each word.

7.2.1 Item2Vec

Item2Vec Barkan and Koenigstein (2016) adapts the basic idea behind *word2vec* as a means of learning item representations γ_i. The main difference between item2vec and word2vec is simply that sequences of words in sentences/documents are replaced by ordered sequences of items that each user has consumed. In practice this simply means that the probability in Equation (7.13) is replaced by

$$\log p(i|j) \simeq \sigma(\gamma'_i \cdot \gamma_j) + \sum_{i' \in \mathcal{N}} \log \sigma(-\gamma'_{i'} \cdot \gamma_j), \tag{7.14}$$

where $\mathcal{N}$ is a set of negative items, again sampled proportionally to overall item frequency.

Barkan and Koenigstein (2016) discusses the effectiveness of this type of item representation in the setting of item-to-item recommendation on a corpus of song listens from *Microsoft Xbox Music*. It was shown that the method naturally identifies latent dimensions that are associated with song genres; and it was argued qualitatively that related items are semantically more meaningful than those produced by alternate item-to-item recommendation techniques.

7.2.2 Word2Vec and Item2Vec with Gensim

Finally, we show how *word2vec* and *item2vec* work in practice, using interaction and review data from beer reviews, much as we did in **??**.

To learn word representations, the input to the model is a list of documents (in this case reviews), each of which is a list of tokens (words). For this example we first remove capitalization and punctuation before tokenizing, as in **??**.

Here we use the *Gensim* implementation of *word2vec*.[6] The model takes as input our tokenized reviews (in this case, we use a corpus of 50,000), a minimum word frequency, a dimensionality (i.e., $|\gamma_i|$), and a window size (i.e., c in Equation (7.11)). The final argument specifies which specific version of the model is used, which corresponds to the model presented above:

```
1 from gensim.models import Word2Vec
2
3 model = Word2Vec(reviewTokens, # Tokenized documents
4                  min_count=5, # Minimum frequency before words are
                       discarded
5                  size=10, # Model dimensionality
6                  window=3, # Window size
7                  sg=1) # Skip-gram model
8
9 model.wv.similar_by_word("grassy")
```

In the final line, we retrieve the most similar words for a particular query; in *Gensim* this is based on the cosine similarity (**??**) between the two word vectors:

$$\max_w \frac{\gamma_w \cdot \gamma_{grassy}}{||\gamma_w|| ||\gamma_{grassy}||} = \text{`citrus'}, \tag{7.15}$$

followed by 'citric', 'floral', 'flowery', 'piney', 'herbal', etc.

Similarly, we can use the same code to run *item2vec*, where our tokenized reviews are replaced by lists of items (i.e., product IDs) that each user has consumed (ordered by time).

[6] https://radimrehurek.com/gensim/

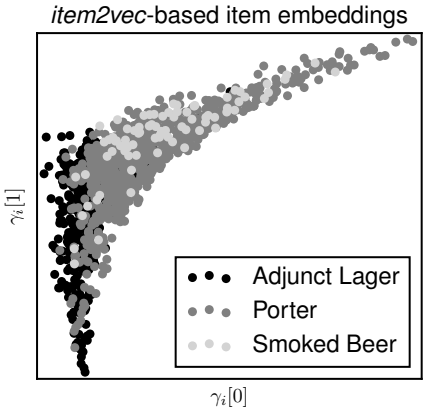


Figure 7.6: Item representations (γ_i) using a two-dimensional *item2vec* model. Representations for items from three distinct categories are shown.

After training a model on review histories, we find that the most similar beers to *Molson Canadian Light* are other light beers such as *Miller Light*, *Molson Golden*, *Piels*, *Coors Extra Gold*, *Labatt Canadian Ale*, (etc.) In Figure 7.6 we train a two-dimensional item2vec model for the sake of visualizing the data, which reveals that beers belonging to different categories tend to occupy different parts of the item space.[7]

7.3 Personalized Sentiment and Recommendation

The models for text we have explored so far, although they can be applied to data like reviews, and can be used to recommend related documents, are ultimately not personalized.

Several attempts have been made to combine models of text with models of users and preferences, and in particular with recommendation approaches. For example, just as we saw techniques in **??** that improve recommender systems through the use of side-information, *text* can be useful to efficiently understand the dimensions of user's opinions.

Often there is a significant amount of textual data available in addition to interactions that might be leveraged to fit better models. Other than helping understand sentiment, text can help to understand the dimensions of items and preferences, e.g. the different properties of products and the different aspects that users care about.

[7] A more effective visualization might be produced by using higher-dimensional embeddings, followed by a distance-preserving visualization technique like t-SNE Maaten and Hinton (2008), though for here we plot the embedding dimensions directly for simplicity.

Text can also be helpful for model *interpretability*. So far, the recommender systems we've developed are essentially 'black boxes,' whose predicions (as in **??**) are defined purely in term of latent factors. Models of text can be used to understand what aspects these latent dimensions correspond to, to summarize reviews, or to explain recommendations.

However, extracting *meaningful* information from text is challenging. E.g. most of the simple text representations we've seen so far (**??**) are high-dimensional and not particularly sparse; simply incorporating such features into general-purpose feature-aware models is possibly not effective, and one must instead design methods to work specifically with text. Below we cover a few representative approaches.

7.3.1 Case Studies: Review-Aware Recommendation

Product reviews are often used as a source of information to improve recommendation performance. Conceptually, reviews ought to tell us much more about preferences and opinions than (e.g.) a single rating can. This could be especially true for latent dimensions (γ_u and γ_i for users and items) since product reviews are intended to 'explian' opinions the underlying dimensions behind users' decisions.

Roughly speaking, there are two schools of thought as to how reviews should be incorporated to improve recommendation performance. One option is to treat review text as a form of *regularization*, essentially to encourage the low-dimensional representations of users or items (via γ_u and γ_i) to be similar to low-dimensional representations extracted from text. Others seek to extract representations from text that can be used to improve feature-based matrix factorization methods. We give examples of both below.

Hidden Factors as Topics An early attempt to incorporate text into recommender systems attempted to do so by making use of *topic models* applied to product reviews McAuley and Leskovec (2013a). Topic models Blei et al. (2003) learn low-dimensional representations of text (essentially finding structure in the types of bag-of-words representations we covered in **??**). 'Topics' then correspond to sets or clusters of words that tend to co-occur together. For example, a topic model trained on movie reviews might dicsover that groups of words associated with 'action,' 'comedy,' or 'romance,' might tend to co-occur together, and therefore that these are distinct topics among movie reviews (we give examples of actual topics in **??**).

The basic idea in McAuley and Leskovec (2013a) is that the low dimensional structure among *reviews* should be related to the low-dimensional structure in

ratings—after all, reviews are intended to explain why a user rated a product a certain way. Furthermore, even though a single rating can tell us very little about the underlying dimensions that explain a user's ratings, a single review potentially contains enough information to

The method is somewhat reminiscent of the social recommendation approach we covered in **?? ?**, in which a shared parameter γ_u was tasked with simultaneously explaining rating dimensions as well as social connections. The argument we made in **??** was that in the absence of sufficient rating data, our estimate of γ_u will then be estimated from u's friends.

Likewise, McAuley and Leskovec (2013a) suggested that a shared parameter γ_u could simultaneously explain rating dimensions via a latent factor model, as well as the topics in reviews:

$$\underbrace{\sum_{(u,i)\in\mathcal{T}} (\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - r_{u,i})^2}_{\text{rating error}} + \lambda \underbrace{\sum_{(u,i)\in\mathcal{T}} \sum_{w\in d_{u,i}} \log p(w|\gamma_u, \psi)}_{\text{topic likelihood}}, \tag{7.16}$$

where ψ is a set of additional (non-shared) parameters specific to the topic model (much like the approach in **??** had shared and non-shared social parameters).

Critically, the above model assumes an alignment between latent rating dimensions and review topics. In practice, there may be dimensions in reviews (i.e., topics) which are not related to rating dimensions (e.g. if a user discusses the plot in a book review, it may have little connection to their rating); likewise there may be 'intangible' latent dimensions that don't correspond to topics expressed in reviews. By assuming a one-to-one relationship between topics and latent dimensions, the model is useful in cold- (or cool-) start settings by forcing the topic model to discover *those topics that are capable of explaining the variation in ratings*. These topics in particular will be the ones that help us to quickly understand the dimensions that explain user ratings from a few interactions.

Examples of the types of topics discovered by this model are shown in Figure 7.8; mostly, the discovered factors correspond to fine-grained product categories, which mostly reflect users' tendency to favor certain types of item over others.

Other Topic-Modeling Approaches The above is a simple approach to combine low-dimensional representations of text (via a topic model) with low-dimensional representations of interactions (via a latent factor model). Several others have adopted a similar approach, typically by modifying how user factors γ_u and topic dimensions are related to each other.

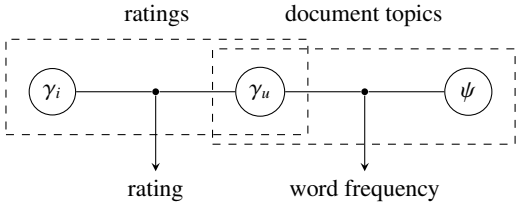


Figure 7.7 Similar to the social recommendation models from **??**, personalized models of text often make use of a *shared* parameter that must simultaneously explain structure in interactions and documents.

theaters	spas	mexican	vietnamese	snacks	italian	medical	donuts	coffee	seafood
theater	massage	mexican	pho	cupcakes	pizza	dr	donuts	coffee	sushi
movie	spa	salsa	vietnamese	cupcake	crust	stadium	donut	starbucks	dish
harkins	yoga	tacos	yogurt	hotel	pizzas	dentist	museum	books	restaurant
theaters	classes	chicken	brisket	resort	italian	doctor	target	latte	rolls
theatre	pedicure	burrito	beer	rooms	bianco	insurance	subs	bowling	server
movies	trail	beans	peaks	dog	pizzeria	doctors	sub	lux	shrimp
dance	studio	taco	mojo	dogs	wings	dental	dunkin	library	dishes
popcorn	gym	burger	shoes	frosting	pasta	appointment	frys	espresso	menu
tickets	hike	carne	froyo	bagel	mozzarella	exam	tour	stores	waiter
flight	nails	food	zoo	bagels	pepperoni	prescription	bike	gelato	crab

Figure 7.8 Example of topics that explain variance in rating dimensions on *Yelp*, from McAuley and Leskovec (2013a).

Ling et al. (2014) and Diao et al. (2014) both consider the same setting as above, in which reviews are used to improve the performance of rating prediction. Both note the limitations of assuming a simple one-to-one mapping between review topics and user preferences as in McAuley and Leskovec (2013a), and

Wang and Blei (2011) proposed a similar approach to the problem of recommending scientific articles, where document representations are extracted from article text, and user preferences are modeled to predict which articles they will include in their libraries. This differs from the above formulation in that text is associated with items (documents) rather than users, and the setting is essentially an instance of 'one-class' recommendation (as in **??**), since one generally doesn't have explicit negative feedback about the articles a user didn't read.

Neural-Network Approaches Zheng et al. (2017) CNN based approach, treat user and item reviews as a document.

Figure 7.9 How Useful are Reviews for Recommendation?
Although user reviews have been used in several academic studies of text-based recommendation, it is worth thinking about how useful such data would be in practice. After all, in real-world settings the vast majority of interactions would not be associated with reviews. Sachdeva and McAuley (2020) studied the settings

7.4 Question Answering Systems

7.5 Personalized Text Generation

In **??** we presented Recurrent Neural Networks as general-purpose models that can be used to estimate the next value in a sequence, or to generate sequences, based on some context and the sequence of tokens seen so far.

Such models are routinely used to sample (or generate) realistic-looking text. Recurrent networks for text generation (see e.g. Graves (2013)) follow essentially the same setup we saw in **??**. At each step t, the network takes an input x_t (either a character, or a word representation), and updates its hidden states h_t based on the current input x_t and the previous step's hidden state h_{t-1} (as in **??**, multiple network layers can be stacked. The sequence of target outputs y is identical to x, but shifted by a single token, i.e., the model is responsible for generating the *next* token in a sequence based on all the tokens seen so far. This type of setup is depicted in **??**.

While the above model will learn to generate realistic-looking samples (i.e., documents that mimic those in the training set), the samples will not be context-dependent, and they will not be *personalized*.

Several papers have sought to adapt RNNs to generate *personalized* text, that is, text that mimics the context, preferences, or writing style of an individual user. We describe several such approaches below. Most are models of product reviews: partly because such data exhibits variation due to the user, item, and interaction between them, but also simply because such data is widely available. We discuss the broader value of this type of technique below.

Why Generate Reviews? Personalized text generation, especially when applied to datasets of product reviews, may seem an unusual task with no obvious application (beyond generating review spam): existing platforms are unlikely to surface generated reviews (such as the one in **??**) to users. However this task has more value when considering the broader context of personalized language generation. In practice, as we've seen elsewhere, reviews are often simply a convenient test-bed for training due to the availability of data. Other applications where personalized language generation could include dialog systems

Table 7.1 *Summary of personalized text generation approaches.*

Goal	Reference	Description
Sentiment analysis	Radford et al. (2017)	Focuses on the task of generating user reviews, though is not personalized; the goal is mainly to learn disentangled representations and discover sentiment.
Attribute-based generation	Dong et al. (2017)	An encoder-decoder approach to generate reviews based on contextual attributes (such as the user, item, or rating).
Abstractive tip generation	Li et al. (2017)	Generates short 'tips' (similar to review summaries), also based on an encoder-decoder approach; reviews are used during training to learn an intermediate representation.
Personalized review generation	Ni et al. (2017)	Generates reviews using a latent-factor approach that takes user and item factors as input.

(**??**), assistive language tools, or natural language processing in other datasets with significant personal variability (e.g. clinical NLP).

Even within the context of reviews, a high-fidelity personalized language model can be used for other functions besides generation. For example the model could be 'reversed' for personalized retrieval or search, i.e., to retrieve items *that a user might describe in a particular way* (e.g. a query such as 'good dress for summer wedding'). Personalized text generation can also be used within systems that explain or justify predictions, as we see in **??**.

RNN-based Review Generation Radford et al. (2017) was among the first papers to explore the use of recurrent neural networks (and in particular, LSTM neural networks, as in **??**) to generate realistic-looking reviews.

Although the model from Radford et al. (2017) is not personalized, the approach shows the effectiveness of recurrent networks to sample realistic reviews; in particular, the model operates at the *character* level (i.e., the sequences as in **??** are

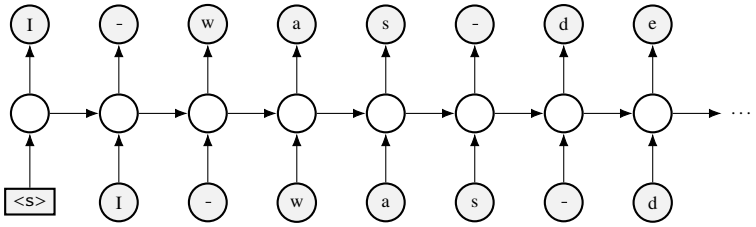


Figure 7.10 Recurrent neural network for text generation. At each step the network is responsible for generating the next token (in this case a character) on the basis of the tokens seen so far, and the network's current hidden state. '<s>' represents a 'start token' to begin generation (and generation would be terminated once the network generates an end token '<\s>').

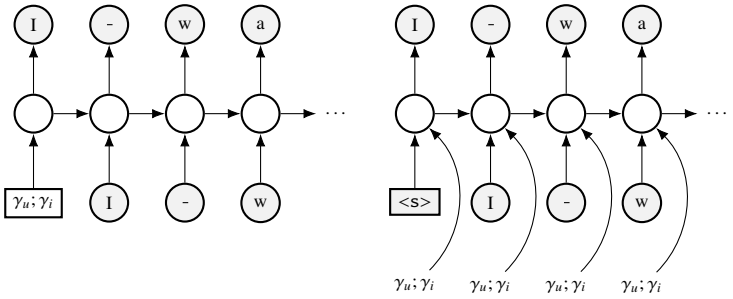


Figure 7.11 Personalized (or contextual) recurrent network architectures. Left: an encoder-decoder architecture; here the start token from **??** is replaced by an input signal produced by (or jointly trained with) a previous model (in this case encoding user and item information). Right: the generative-concatenative network from **?**; here the contextual information is input during every step to help the model 'remember' the context over many steps.

Conditional Review Generation Given the promise of using recurrent neural networks to sample realistic-looking reviews from a background distribution, several papers have followed a similar approach to Radford et al. (2017) to generate reviews that are relevant to a specific context.

Conceptually, generating reviews to match a specific context follows ideas from *encoder-decoder* architectures which have proved useful in (for example) image captioning settings Vinyals et al. (2015). Here, rather than passing a start token to the generator, as in **??**, one passes an encoding of (i.e., a low dimensional representation) the context, such as an embedding representing an image; following this, decoding follows the same approach as in **??**, where the

model's hidden states ought to retain the essential components of the context necessary to generate text conditionally.

Dong et al. (2017) follow this type of encoder-decoder approach to accomplish 'attribute-based' conditional review generation. The approach follows the setting described above, whereby attributes are encoded and passed to an LSTM text generation model. The attributes used in their model include a user ID, an item ID, and the score associated with the review.[8]

Li et al. (2017) follow a similar approach to generate short 'tips,' i.e., short summaries of reviews. The setting also follows an encoder-decoder approach, though is trained on data from *Yelp* and *Amazon* that includes both reviews and summaries (or 'tips' on Yelp). While summaries are used as the model output, reviews are used during training to learn an effective intermediate representation that explains interactions between users and items.

Personalized Review Generation The above methods generate reviews based on specific features or attributes, and as such can essentially be thought of as forms of 'contextual' personalization as we discussed in **??**. Ni et al. (2017) designed text generation methods that directly model users (and items) in order to estimate reviews given only the context of a user and item ID.

The basic setup follows a latent factor approach, i.e., the 'input' to the model is a representation γ_u for the user and γ_i for the item. These latent user and item representations are trained jointly with the language model (in this case an LSTM, as in **??**); in practice these representations are concatenated onto the input tokens x as in **??**. Conceptually, rather than the latent factors γ_u and γ_i explaining user preferences and item properties that predict ratings (as in **??**), user factors must now account for patterns of variation in user writing styles (e.g. the structure used in their reviews), and item factors must account for the overall characteristics of items (e.g. their objective properties) that users are likely to write about. At the same time both the user and item factors must *jointly* explain the user's sentiment toward the item, e.g. the positive or negative language that they will use.

An example of a review generated via this technique is shown in Figure 7.12. The review (in this case of a beer from *beeradvocate*) is surprisingly coherent and seems to capture (a) the user's writing style (e.g. they tend to write their reviews across separate paragraphs describing each aspect); (b) the item's overall characteristics (e.g. the category and flavor profile); and (c) the essential features of the user's preference (i.e., a lukewarm review). Recall that as with a

[8] Though this dependency on having the rating as an input could presumably be overcome by estimating it separately.

12 oz. bottle, excited to see a new Victory product around, **A:** Pours a dark brown, much darker than I thought it would be, rich creamy head, with light lace. **S:** Dark cedar/pine nose with some dark bread/pumpernickel. **T:** This ale certainly has a lot of malt, bordering on Barleywine. Molasses, sweet maple with a clear bitter melon/white grapefruit hop flavour. Not a lot of complexity in the hops here for me. Booze is noticable. **M:** Full-bodied, creamy, resinous, nicely done. **D:** A good beer, it isn't exactly what I was expecting. In the end above average, though I found it monotonous at times, hence the 3. A sipper for sure.

A: Pours a very dark brown with a nice finger of tan head that produces a small bubble and leaves decent lacing on the glass. **S:** Smells like a nut brown ale. It has a slight sweetness and a bit of a woody note and a little cocoa. The nose is rather malty with some chocolate and coffee. The taste is strong but not overwhelmingly sweet. The sweetness is overpowering, but not overwhelming and is a pretty strong bitter finish. **M:** Medium bodied with a slightly thin feel. **D:** A good tasting beer. Not bad.

Figure 7.12

traditional recommender system although both the user and item have been seen during training, though this specific user and item *combination* have not.

Extensions of this work combined latent user and item representations with observed attributes Ni and McAuley (2018). Doing so can help the model to use language that better captures specific item details (such as technical features of an electronics product). Further applications are proposed that use this type of technique for *assistive* review generation, e.g. rather than generating reviews from scratch, the system could be used to help users write reviews based on a template or specific points or attributes, while still following their personalized writing style.

7.5.1 Text-Based Explanations and Justifications

So far, we have examined (a) textual data as a means of improving the *predictive accuracy* of personalized models (**??**); and (b) textual data as the *output* of a predictive model (**??**). Beyond these applications, text is also appealing as a means of *explaining* model predictions.

Text-based explanation connects to the more broad topics of *interpretability*, *explainability*, and *justification* of machine learning models. In the case of text-based explanations, the goal is to retrieve or generate a short text fragment that explains a model prediction. Such models bring us a little closer to the conceptual goal of producing 'human-like' explanations, i.e., mimicking the way one person would justify a decision or recommendation to another.

As with the personalized text generation models above, much of the work in this space is focused on recommendations and review datasets, as review data serves as a convenient testbed to train models for personalized explanation.

The more broad topic of (non-personalized) text-based explanation has been considered in other settings, such as text-based classification Liu et al. (2018).

Extractive vs. Abstractive Approaches Justification (and summarization) approaches can broadly be categorized in *extractive* or *abstractive*. 'Extractive' approaches use retrieval-like approaches to select text fragments (e.g. from a training corpus) that are relevant to a given context or query; 'abstractive' approaches generate novel text, either by paraphrasing the corpus or via a generative approach (as in **??**).

Crowd-sourced Explanations Prior to the use of complex generative models for text-based explanation, Chang et al. (2016) sought to use *crowd-sourcing* to generate personalized explanations for movie recommendations. Since crowd workers are unlikely to be available to generate recommendations in real-time, Chang et al. (2016) adopt a template-based approach, where workers generate justifications based on users' interest in a particular aspect or tag (e.g. why should *Goodfellas* be recommended to a user who likes *drama*?). To help crowd-workers write explanations, workers are shown extracted review segments relevant to a particular tag, and are tasked with abstracting that text into a coherent explanation.[9]

Regardless of the practicality of crowd-sourcing such explanations, a main goal in Chang et al. (2016) is to confirm the overall value of text-based explanations to users (where human-generated explanations can be regarded as something of a gold-standard). They evaluate text-based explanations in terms of *efficiency* (acquainting users with the relevant properties of an item), effectiveness (helping users decide whether they want to watch a movie), *trust*, and *satisfaction*. They find that text-based explanations are preferred over more trivial tag-based explanations in terms of efficiency, trust, and satisfaction (though with negligible change in terms of effectiveness).

? Let Me Explain: Impact of Personal and Impersonal Explanations on Trust in Recommender Systems Kunkel et al. (2019)

Generating Explanations from Reviews Ni et al. (2019a) explores abstractive approaches to justification generation. Overall, the model is similar to the one described in **??**, where the model is trained to generate a review given a particular user/item pair. Ni et al. (2019a) uses a similar training setup, except that the target text used for training is text that has been identified as being

[9] The complete pipeline from Chang et al. (2016) includes a few additional details, such as a step to vote on the best among several crowd-based explanations, among others.

Method	Type	Output
Personalized Retrieval	Extractive	*A great burger and fries*
Generated 'tip'	Abstractive	*This place is awesome*
Generated explanation	Abstractive	*breakfast sandwiches are overall very filling*

Figure 7.13 Examples of generated justifications for a recommendation of *Shake Shack* to a particular user. From Ni et al. (2019a).

suitable to surface as a recommendation justification. Ultimately this text is harvested from reviews, with a main challenge being how to identify suitable justification sentences among review text. Ni et al. (2019a) argues that training on this type of harvested text yields more effective justifications than models trained on reviews, tips, or retrieval-based techniques.

Examples of justifications generated using different techniques and training setups (from Ni et al. (2019a)) are shown in Figure 7.13.

7.5.2 Conversational Recommendation

Survey on conversational recommendation:

Arguably, our implicit goal when developing systems for text-based explanation or justification above is to mimic the ways humans explain or justify their decisions. Following this, perhaps an ideal system for interactive recommendations might mimic the paradigm of *conversation*.

Query Refinement Early approaches for conversational recommendations essentially use conversation as a form of iterative query refinement. In Thompson et al. (2004) users are asked questions that attempt to determine their preferences or constraints toward a fixed set of attributes (e.g. cuisine type, price range, etc.); as such a user model is simply a weighting over potential attribute values. Retrieval then consists of selecting items whose attributes most closely match user preferences.

Other early approaches to 'conversational' recommendation are essentially forms of interactive recommendation, in which a system can query users or gather feedback from users during each round Mahmood and Ricci (2007, 2009). Mahmood and Ricci (2009) adopts a reinforcement learning approach to interactive recommendation. During each step, the system may perform a number of actions, including asking users more detail about specific attributes, or asking them for feedback on a panel of recommendations. Reinforcement learning techniques are used to select an optimal policy in terms of what actions the system should perform under a given state.

Interactive Recommendation Christakopoulou et al. (2016) follows a similar strategy to those above, though embeds the interactive recommendation framework within a latent factor model, similar to those we studied in Chapter 4. Here, the goal is to use a conversational strategy to quickly infer users' preferences γ_u toward item properties γ_i. Interactions consist of a one-sided form of 'converversation' in which the system asks simple questions to the user about their preferences; a main goal of Christakopoulou et al. (2016) is to understand the ideal characteristics of questions that should be asked to user in order to elicit their preferences efficiently. Questions can be *absolute* (ask a user whether they like or dislike an item), or *pairwise* (ask a user which of two items they prefer). Question selection strategies consist of determining which items should be compared during each step. Strategies include selecting random items, items with the highest estimated compatibility, or items whose compatibility has the most uncertainty. After each question, compatibility scores are recomputed based on question responses, resulting in a model that gradually becomes more accurate during successive rounds.

Free-form Conversation More recent approaches try to follow the conversational paradigm more literally. Ideally, conversations should be *free-form*, in which both the system's question and the user's response take the form of free text. Li et al. (2018) attempted to formulate conversational recommendation in this form, for the specific task of movie recommendation. A major challenge which they overcome is to build appropriate ground-truth data for this task. Their approach builds on previous attempts to build dialog datasets, including dialogs specifically focused around movies Dodge et al. (2015). Dialogs are constructed by crowd workers, who assume roles of a *recommender* or *seeker*; conversations between the recommender and the seeker are tagged in terms of the movies mentioned, as well as explicit feedback (has the seeker seen the movies mentioned and did they like them). Around ten thousand such conversations are collected.

Having collected such data, Li et al. (2018) then seek to train a dialog generation model that can fulfil the role of the recommender. Their solution combines ideas from dialog generation with a recommender system, so that users' preferences can be estimated and the output controlled to reference specific movies.

A potential limitation of Li et al. (2018) is that it relies on explicit movie mentions and feedback to learn user preferences; as such their method could not straightforwardly recommend a movie based on a simple goal, such as a user requesting "a good sci-fi movie." Kang et al. (2019a) seek to build conversational recommenders following this 'goal oriented' view of recom-

Reference	Type	Description
Thompson et al. (2004)	Query refinement	Elicits users' preferences and constraints with regard to item attributes.
Mahmood and Ricci (2009)	Reinforcement learning	Queries users about recommendations attributes during each round; learns a policy to choose queries to efficiently yield a desirable recommendation.
Christakopoulou et al. (2016)	Iterative recommendation	Collects feedback about recommended items in order to iteratively learn user preferences; explores various query strategies to elicit preferences quickly.
Li et al. (2018)	Free-form conversation	Collects conversational data in which a 'recommender' suggests movies and a 'seeker' provides feedback. Trains a dialog model to mimic the role of the recommender.
Kang et al. (2019a)	Free-form conversation and RL	Similar to the above, though trained using reinforcement learning so that the 'recommender' and 'seeker' exchange information to arrive at a target recommendation.

Figure 7.14 Summary of approaches for conversational recommendation

mendation. To collect data, a conversation game is conducted, where both the seeker and recommender ('expert') are given prompt movie sets: the seeker's set represents their ostensible interests, while the expert's set is a collection of movies, one of which matches the seeker's preferences (determined offline based on a similarity metric). The expert's goal is to determine *which* movie in their set matches the user's preferences via repeated conversation turns. Kang et al. (2019a) note that while in principle the expert could simply enumerate movies in their set until they reach the 'correct' one, though in practice this rarely happens, and players tend to engage in free-form conversation, asking about general attributes and qualities.

Like Li et al. (2018), Kang et al. (2019a) then train dialog agents to play the game. During play, the expert can either engage in a dialog turn or select one of the movies from their set

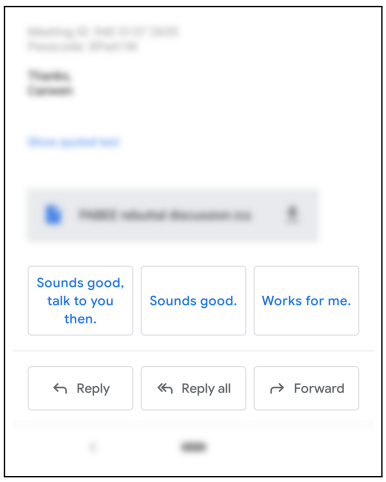


Figure 7.15: Examples of short responses suggested by Google's *Smart Reply*.

Critiquing A Ranking Optimization Approach to Latent Linear Critiquing in Conversational Recommender System Li et al. (2020a)

7.6 Case Study: Google's *Smart Reply*

Google's *Smart Reply* is a system developed for *GMail* to automatically recommend short responses for e-mail (Figure 7.15). Given an e-mail thread as input, the system is tasked with surfacing (three) likely responses; although the goal is ultimately to maximize engagement with the feature, the system is trained by taking a large corpus of thread/response pairs, and learning to predict (or maximize the likelihood of) users' historical responses to a given thread.

As a case-study in personalized text generation, this system is interesting for several reasons:

- As a form of personalized machine learning, Google's solution does not have explicit user parameters (i.e., it is similar to the non-parametric systems described in Rather, 'personalization' is done implicitly by learning from the context already present in the e-mail thread.
- Google describes two successive—and quite different—solutions to this problem in a sequence of papers. The first is based on a sequence-to-sequence language model (i.e., a text generation framework); the second is a seemingly more trivial nearest-neighbor based solution.

- Other interesting facets of the studies include how they deal with scalability, diversification, appropriateness of suggesting a reply for a given context, etc.

The models we will discuss are described in Kannan et al. (2016) and Henderson et al. (2017). We will mainly discuss the latter, more recent paper, as the solution mostly supercedes the former one, though we will make comparisons between both.

7.7 Case Study: Personalized Recipes

Given the significant variation in people's tastes and dietary needs, *recipes* have recently emerged as an interesting source of textual data, both for personalized retrieval, and more recently, generation Li and McAuley (2020).

Early systems to facilitate personalized interactions with recipes did so by helping users to search for recipes whose ingredients target specific health conditions Ueta et al. (2011). Later systems approached the same task with explicit rules, and constraints of ingredients to avoid given specific dietary restrictions Inagawa et al. (2013). Other retrieval-oriented systems have sought to help users find recipes 'in the wild,' e.g. by searching for recipes that correspond to a photo Marin et al. (2019).

More recently, Majumder et al. (2019) considered whether personalized recipes can be synthesized using text generation frameworks. Here, the idea is to generate novel recipes that are consistent with a particular user's preferences (or specifically, their previous interactions). The setup is somewhat similar to that of **??**, where we sought to generate personalized text conditioned on a (representation of a) user and an item. In Majumder et al. (2019), the training objective consists of a set of recipes a particular user has consumed, based on which the system should output (i.e., generate the text of) another recipe the user would consume.[10] In this way, the method can generate recipes that are consistent (in terms of e.g. ingredients, cooking techniques, etc.) with those that the user would normally consume. An example of a recipe generated with this system is included in Figure 7.16.

?

[10] In practice, Majumder et al. (2019) inputs other metadata (such as a recipe title) to the method to overcome the difficulty of generating recipes 'from scratch.'

Name: Pomberrytini; **Ingredients: pomegranate-blueberry juice, cranberry juice, vodka**
Combine all ingredients except for the ice in a blender or food processor. Process to make a smooth paste and then add the remaining vodka and blend until smooth. Pour into a chilled glass and garnish with a little lemon and fresh mint.

Figure 7.16: Example of a personalized recipe, from Majumder et al. (2019).

Chicken Dijon
Ingredients: Butter, Chicken, Chicken broth, Dijon mustard, Flour,Light cream, Onion, Pepper,Salt, Wine
1) Sprinkle the chicken breasts with onion powder, lemon pepper and salt. 2) Saute the chicken in the butter for 20 min, or cooked through and tender. 3) Remove chicken to a platter and keep warm. 4) Measure the pan juices and enough chicken broth to make 1 cup liquid. 5) Return the stock mixture to the pan and add the wine. 6) Stir together the light cream and the flour till smooth. 7) Add to the broth, cook and stir till smooth, thickened and bubbly. 8) Stir in the dijon mustard, stir till smooth. 9) Return the chicken to the pan, or serve the sauce separately.

Low-Calorie Chicken Dijon
Ingredients: Chicken, Dijon mustard, Flour, Pepper, Salt, Carrot, Garlic, Mushroom, Soy sauce
1) Preheat oven to 350 degrees f. 2) Place chicken breasts, breast side up, in a shallow roasting pan. 3) Combine soy sauce, mustard,garlic and salt. 4) Brush over both sides of the chicken breasts. 5) Bake for 40 minutes. 6) Meanwhile, heat a small sauce-pan over medium heat. 7) Stir in carrots, mushrooms and pepper. 8) Cook and stir for 5 minutes. 9) Stir in flour. 10) Pour over chicken breasts, and bake for an additional 10 minutes until chicken is cooked through. 11) Serve chicken with carrots and sauce.

Figure 7.17 Example of a recipe (left) and a modified version (right) targeting a specific dietary goal **?**.

7.8 Exercises

The exercises below may be conducted using any dataset that includes reviews associated with numerical ratings.

Exercises

7.1 Implement a sentiment analysis pipeline based on a bag-of-words model, as in **??**, Equation (7.2). You may follow the code provided as a starting point to build a model based on the 1,000 most common unigrams.

7.2 Several choices must be made when building feature representations

from text. Extending your model from **??** 7.1, and using a validation set, experiment with various modeling choices (in addition to the regularization hyperparameter λ). Possible modeling choices include:

- The dictionary size;
- Whether to use unigrams, bigrams, or a combination of both.[11]
- Whether to remove capitalization, punctuation, or to treat punctuation characters as separate words;
- whether to use tf-idf

7.3 Something with item2vec

7.4 As discussed in **??**, tf-idf representations of documents can be used to retrieve documents that are semantically similar to each other. For this exercise, we'll use all reviews of a single item as a single 'document.' First, compute the tf-idf representations of all items, and compute the most similar item (in terms of cosine similarity between tf-idf vectors) given a particular query (e.g. the first item in the dataset). Compare this to similarity computed on bag-of-words representations.

7.5 In **????**, we considered using similarity functions to predict ratings for user-item pairs. Adapt one of those predictors (e.g. the predictor from **??**) to estimate ratings using the item-to-item similarity function you developed in **??** 7.4.[12]

7.8.1 Project 5: Personalized Document Retrieval

In the above exercises, we have focused on building predictive models using features derived from text. In fact, most of the models we developed were not *personalized*. Here, we'll explore how to personalize these models following the approaches we developed in this chapter.

As a starting point, we'll build a personalized model of sentiment analysis; your model can extend the one you developed in Exercises **??**. Perhaps the simplest form of personalization consists of fitting bias terms for each user, much like the bias terms we included when developing recommender systems in **??**. Such a term can account for the fact that one user may regard (e.g.) a three-star rating as positive (and therefore use positive language) whereas another user may regard it as negative. This term can be incorporated by extending

[11] That is, all unigrams *and* bigrams can be sorted by popularity;

[12] Given that repeatedly computing item-to-item similarities for high-dimensional text features is likely quite computationally intensive, it is most likely feasible to evaluate your method only on a small sample of user-item pairs.

a model like that in **??**:

$$\alpha + \beta_u + \sum_{w \in \mathcal{D}} \text{count}(w) \cdot \theta_w. \tag{7.17}$$

This model could be fit either by (a) treating the user identity as a one-hot vector, and treating the problem as an ordinary linear regression problem, or (b) by gradient descent, much as we fit bias terms when developing recommender systems in **??**.

Having fit this model, investigate the extent to which the addition of the bias term improves performance (e.g. in terms of the MSE), and the extent to which it alters the weights associated with the most positive and negative words.

Following this, develop a more complex personalized model that estimates a user's compatibility with a document in terms of latent factors. Rather than associating latent factors with individual documents, we'll associate them with *words* in documents, augmenting our bag-of-words model from Equation (7.17)

$$\alpha + \beta_u + \sum_{k=1}^{K} \sum_{w \in \mathcal{D}} \text{count}(w) \cdot \gamma_{u,k} \theta_{k,w} \tag{7.18}$$

8

Personalized Models of Visual Data

Many of our decisions are guided by visual factors, and preferences toward visual attributes can be highly subjective. Traditional models of visual data deal with problems like classification, detection, or more recently image generation, though the bulk of approaches are not personalized: discriminative models (classifiers, detectors) are usually concerned with identifying some objective label in an image, or generative models are concerned with learning a background distribution governing the overall dynamics of a large corpus of data.

The situation above is much as we saw when introducing models for text in Chapter 7. Likewise, just as we saw in Chapter 7, many problems involving visual data depend on subjective factors or user context. Just as we saw in **??** that text can be used to improve both the fidelity and interpretability of recommendations, visual data can likewise be included to improve the accuracy of models in settings where personal preferences are significantly guided by visual signals.

Visual data are critical in domains like fashion, where preferences are largely guided by visual factors. Problems like recommendation in such settings are highly personalized, and problems like compatibility among items depend on complex factors that are hard to precisely define. Furthermore, recommendation in scenarios frequently suffers from *cold-start* (or 'cool start') problems, given the long tail of new and rarely consumed items.

We'll begin this chapter by exploring how to incorporate visual data into recommendation approaches. Much of our discussion will be centered around domains like fashion recommendation, where visual features naturally play a key role, though we'll also look at other visually-guided scenarios ranging from art to home decor.

Following this we'll explore new recommendation modalities involving visual data. Item-to-item, or set-based recommendation are particularly impor-

tant in settings involving visual data, again including settings like outfit generation in fashion.

Finally we'll explore personalized *generative* models of images. Just as we saw models that generate personalized text in Chapter 7, there are a few settings where one might wish to generate images that are personalized to a user's preference or context, such as systems for personalized design.

8.1 Visually-Aware Recommendation and Personalized Ranking

The issues we introduced above suggest the possibility of systems for recommendation and ranking that make use of visual features. Much as we saw with text in **??**, visual data is difficult to incorporate directly into recommender systems, given that feature representations are high-dimensional, and dense. Below we discuss a few of the main approaches to incorporate visual data into recommendation and personalized ranking models.

8.1.1 Visual Bayesian Personalized Ranking

Initial attempts to incorporate visual data into ranking models extend the Bayesian Personalized Ranking framework from **??** to incorporate observed image features f_i associated with each item, such as a product image from an e-commerce website.

That is, we want to define a compatibility function $x_{u,i,j} = x_{u,i} - x_{u,j}$ (as in **??**) that estimates which of two items i and j are more compatible with the user.

Starting with a simple latent-factor based compatibility model, and given our goal of building a system that works in (item) cold-start scenarios, we might first consider simply replacing our (latent) item representations γ_i with our observed image features, i.e.,

$$x_{u,i} = \alpha + \beta_u + \beta_i + \gamma_u \cdot f_i. \tag{8.1}$$

In this way, γ_u would now determine which features are most compatible with each user (in fact, this is a linear model as in Chapter 3). Although conceptually reasonable, the issue in doing so becomes quickly apparent once we consider that image features are typically very high dimensional. For instance, visual features extracted from ImageNet (as used in He and McAuley (2015)) are 4,096 dimensional. Incorporating them directly into $x_{u,i}$ as in Equation (8.1)

would thus require fitting thousands of parameters *per user*, which is not feasible in datasets that typically consist of only (e.g.) tens of interactions per user.

Visual Bayesian Personalized Ranking He and McAuley (2015) attempts to address this by projecting images into a low-dimensional embedding space via a matrix $\mathbf{E}$. Here $\mathbf{E}$ is $|f_i| \times K$ matrix (e.g. $4,096 \times K$), which projects the image into a K dimensional space. Following this, the projected image dimensions can be matched to user preference dimensions:

$$x_{u,i} = \alpha + \beta_u + \beta_i + \gamma_u \cdot (\mathbf{E} f_i). \tag{8.2}$$

Note that the projected features $\mathbf{E} f_i$ fulfil much the same role as γ_i in a typical latent factor model, except that they are learned based on observed features.

Note that $\mathbf{E} f_i$ is a *learned* embedding, and is fit so as to maximize the probability of observed interactions, as with all other terms in Equation (8.2). Note also that while $\mathbf{E}$ is high dimensional (e.g. around 40,000 parameters if $|f_i| = 4,096$ and $K = 10$), it is a *global* term that is shared among all items; thus for a large enough dataset it accounts for only a small fraction of the model's parameters.

Because the embedding is low-rank, we are assuming that users' preferences toward these visual dimensions can be explained via a small number of factors. While this is similar to the assumption made by a 'standard' latent factor model (e.g. as in **??**), we are further assuming that these factors can be explained by visual dimensions. However in practice there could be several latent factors *not* explainable by visual features (e.g. factors due to price, material, brand, etc.) To address this the original paper includes *both* latent item factors γ_i as well as visual item factors $\mathbf{E} f_i$:

$$x_{u,i} = \alpha + \beta_u + \beta_i + \underbrace{\gamma_u (\mathbf{E} f_i)}_{\text{visual preference dimensions}} + \underbrace{\gamma'_u \cdot \gamma_i}_{\text{latent preference dimensions}} + \underbrace{\beta^{(f)} \cdot f_i}_{\text{visual bias}}. \tag{8.3}$$

Correspondingly, there are two sets of user terms: γ_u, which explains preferences toward visual factors, and γ'_u, which explains preferences toward non-visual factors. Intuitively, the two terms will play different roles depending on how 'cold' an item is: for a cold (or 'cool') item, visual features will be much more reliable than latent factors; whereas for 'hot' items (i.e., those with many associated interactions) γ_i will be able to capture additional non-visual dimensions. Equation (8.3) also includes a 'visual bias' term $\beta^{(f)} \cdot f_i$ ($\beta^{(f)}$ is a $|f_i|$-dimensional vector) that is able to estimate item biases in cold scenarios.

Several other considerations must be made to implement such an algorithm efficiently. For instance, accessing the (high-dimensional) image features at random (e.g. within a stochastic gradient ascent algorithm), leads to poor caching

performance; likewise computing the projection $\mathbf{E}f_i$ is expensive. In practice these issues are dealt with by pre-computing all projections $\mathbf{E}f_i$ (which can be performed as a single matrix-matrix product), and updating $\mathbf{E}$ only periodically during gradient ascent.

Visual Bayesian Personalized Ranking is effective in settings where items have few associated interactions. In the original paper it is demonstrated on a clothing dataset from *Amazon*, as well as a clothing trading dataset (*tradesy*). The latter is particularly challenging because traded items are not associated with long transaction histories, meaning that model predictions must largely rely on visual signals.

8.1.2 Case Study: Modeling the Visual Evolution of Fashion Trends

He and McAuley (2016) extended the above ideas from Visual Bayesian Personalized Ranking to incorporate temporal dynamics. Modeling temporal dynamics in this setting is interesting partly because the patterns of temporal variation in (e.g.) clothing purchases are different from those that were successful in other settings, such as on *Netflix* (Section 6.2.1).

The main idea in He and McAuley (2016) is simply to break the training dataset into a sequence of *epochs*, each of which have their own parameters. These epochs are somewhat akin to the 'bins' used to model long-term temporal dynamics on Netflix, though a key difference is that the bin sizes are variable and placed at learned intervals (using a dynamic programming procedure); given that the model has a large number of parameters, this helps to ensure that fewer bins are used during time periods with little temporal variation, whereas more (and smaller) bins are used during periods which are more dynamic.

Potentially, models of temporal dynamics in fashion are also interesting because they might reveal historical trends in fashion over time.

8.2 Visual Compatibility of Items

In the previous section we saw how visually-aware recommender systems can be used to match items (or images of items) to users' preferences. In domains like fashion, there is also interest in establishing compatibility between items (or recommending one item on the basis of another), for example to generate outfits, or even wardrobes of mutually compatible items. In Section 4.3 we

discussed several types of recommendation approaches that considered similarity between items; such measures guide 'item-to-item' recommendation approaches, as in Figure 4.2.

Here, we would like to develop similar approaches that establish visual similarity (or compatibility) between items. Rather than basing similarity on interaction histories as we did in Section 4.3, here we can base similarity directly on the visual appearance of items.

8.3 Case Studies: Fashion Compatibility

Many studies on visual compatibility are specifically concerned with *fashion images*. Estimating compatibility in such a domain has obvious applications to specific tasks like outfit generation and recommendation. More broadly, in settings like fashion, compatibility with past interactions or purchases is a strong predictor of future interactions.

Some of the specific characteristics that makes this problem difficult (and different from other forms of (e.g.) item-to-item recommendation, are as follows:

- It is challenging to construct datasets that act as 'groundtruth' for visual compatibility, i.e., pairs of items that are known to 'go well' together.
- Further to the above, any groundtruth of compatible items are bound to be highly noisy, and highly subjective; successful methods need to account for these challenges, and possibly learn compatibility in a personalized way.
- The features that make items visually compatible in settings like fashion could be incredible subtle, and could be quite different from the information available in co-purchase data, or even in most visual feature descriptors.
- Finally, the notion of 'compatibility' is semantically quite different from 'similarity.' E.g. clothing items that go together should be similar in some ways but complementary in others.

Approaches to these problems mainly differ in their specific solutions to the problems above. We describe a few key papers below.

8.3.1 Estimating Compatibility from Co-purchases

Early approaches to estimating visual compatibility built datasets from co-purchases, for example using publicly-available datasets of reviews from *Amazon*.

McAuley et al. (2015a) crawled data from Amazon's surfaced recommendations ('people who bought X also bought Y' etc. as in Figure 4.2), and, in the case of clothing, treated these as 'groundtruth' examples of items that are visually compatible.

Having defined such a compatibility function, the goal is to learn an appropriate distance function, such that frequently co-purchased items tend to be closer together than others. The distance function is then used in a simple binary classification framework (similar to logistic regression) to predict:

$$p(i \text{ co-purchased with } j) = \sigma(c - d(i, j)). \tag{8.4}$$

Previously, we considered how to learn distance functions for problems such as next Point-of-Interest recommendation (**??**). When doing so, items (and users) were projected into a latent space via parameters γ. To recommend compatible clothing, we might instead use features extracted directly from (the product images of) i and j: First, general-purpose visual features are readily available, and are likely to be informative in fashion compatibility scenarios; second, reliance on features is desirable in cold-start settings, which might be common in settings (like fashion) where item vocabularies are large and changing; third, a model based only on visual features can be more straightforwardly *transferred* to settings where user data is not available.

Given image features associated with the items f_i and f_j, McAuley et al. (2015a) discusses several strategies for establishing visual similarity. Trivially, one could directly consider the (squared) distance between f_i and f_j (i.e., $\|f_i - f_j\|_2^2$), however general-purpose image features may not focus on attributes that are relevant to fashion.

A second solution is to learn a weighted distance function that discovers which features are relevant and discards those that are not, i.e., $\sum_k w_k(f_{i,k} - f_{j,k})^2$. However it is argued that in fashion scenarios 'compatibility' cannot be captured by modeling similarity between features—for example, a user generally would not select a shirt because it looks 'similar' to a pair of pants. To address this, a similarity function is proposed which projects the images into a low-dimensional 'style space':

$$d(i, j) = \|s_i - s_j\|_2^2; \quad \text{where} \quad s_i = M \times f_i. \tag{8.5}$$

In the case of McAuley et al. (2015a) f_i is a 4096-dimensional image descriptor, extracted from a model trained on ImageNet Jia et al. (2014); M is then a $4096 \times K$ vector, where K is some small embedding dimension (on the order of $K = 10$). Ultimately the embedded vector $s_i = M \times f_i$ is analogous to the latent vectors γ_i from previous models, in the sense that it captures the underlying dimensions that explain variation in co-purchases.

The method is then trained using a dataset C of complementary pairs, along with a set of *non-complementary* pairs C^- (which in practice are sampled randomly). The model is then trained using a logistic regression-like setup to distinguish complementary and non-complementary pairs:

$$\underbrace{\sum_{(i,j)\in C} \log \sigma(c - d(i,j))}_{\text{complementary pairs}} + \overbrace{\sum_{(i,j)\in C^-} \log(1 - \sigma(c - d(i,j)))}^{\text{non-complementary pairs}} \tag{8.6}$$

Finally, although the approach is mainly designed for item-to-item recommendation (and as such is not *personalized*), a personalized version can be developed by adding a user latent vector γ_u that encodes which dimensions of this 'style space' are important to them:

$$d_u(i,j) = \sum_k (\gamma_{u,k} s_{ik} - \gamma_{u,k} s_{jk})^2 \tag{8.7}$$

(in this case, the model is trained on triples (u, i, j) of co-purchases of items by each user u).

Pairwise Preference Regression for Cold-start Recommendation

Ultimately, the paper shows that this type of model can be used in several ways. First, it can predict co-purchases accurately, especially when predictions are personalized. Second, the use of image data is effective at *visualizing* the parameters of the model, i.e., determining what are the primary dimensions that explain variance in users' 'styles.' Finally, since the model (as in Equation (8.5)) takes only images as input, it can be transfered to assess compatibility (and arguably, 'fashionability') of outfits outside of the original training data.

Veit et al. (2015) made use of the same co-purchase data to solve the same task, but did so directly from the 'pixel level,' i.e., by training a Convolutional Neural Network, rather than using a pre-trained image representation. This type of architecture is depicted in Figure 8.1: two input images (items), labeled as 'compatible' or 'incompatible,' are passed through two CNNs, both of which share the same parameters. The CNNs learn low-dimensional representations $\phi(x)$ and $\phi(y)$ for the two items; these are essentially equivalent to the 'style space' embeddings of Equation (8.5), except that they are learned from the pixel level, and therefore could potentially capture subtle characteristics not available in pre-trained representations. Like Equation (8.5), the model is trained to learn a metric, so that compatible items have nearby embeddings and incompatible items do not.

He et al. (2016a) showed that prediction performance can be improved in

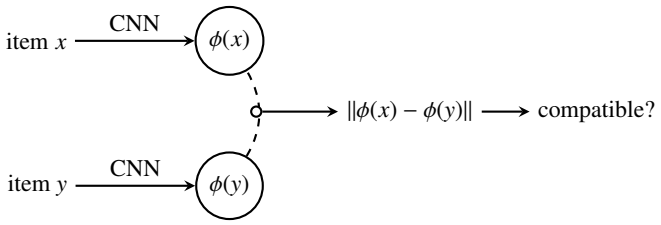


Figure 8.1 Basic Siamese setup for item-to-item compatibility.

fashion settings by using separate embeddings for the 'query' and 'target' items i and j, i.e.,

$$d(i, j) = \|\gamma_i - \gamma'_j\|_2^2. \tag{8.8}$$

Critically, by using two different latent spaces, 'compatibility' need not follow the assumptions of a metric space, e.g. an item need not be compatible with itself. In this way the model can learn which aspects should be systematically different when matching items—such as blue pants going with brown shoes.

8.3.2 Learning Compatibility from Images in the Wild

The above papers showed that visual data can be effective when predicting co-purchases in domains like fashion. However, it is arguable whether such models actually learn a useful notion of 'fashionability.' For one, a co-purchase is a very noisy indicator of whether two items are compatible: in practice, two purchases on one account may not even be for the same person. Secondly, the images in such datasets (e.g. product images from Amazon) are not 'wild' images (they are usually scaled, centered objects on a white background), so could not be used to assess the fashionability of an outfit in a photo (for example).

Kang et al. (2019b) tried to address this issue by developing models of fashionability that operated directly on images in the wild. At training time, the approach consists of an image (or 'scene') that is known to contain a particular item; each item also has a 'clean' product image much like those in Section 8.3.1. The basic idea is then that item must be compatible with other object in the scene.

To build a training set, the known items are cropped from each training image, each resulting in a scene image that does *not* contain the known item, but ought to contain compatible objects. Ultimately, this means that we separately have an item, plus a (cropped) scene that is known to be compatible

with the object, but does not contain it. From here, fashion compatibility can be estimated by learning a relationship between the scene and product images:

$$d(s, p) = \|f(s), f(p)\|_2^2. \tag{8.9}$$

Much like Equation (8.5), this distance function is based on learned embeddings of the scene and product images (s and p). Several differences exist between these embeddings and those used in Section 8.3.1, mostly to account for the fact that the scene image likely contains a large number of *irrelevant* objects. Critically, the method uses an *attention mechanism* (see e.g. Xu et al. (2015)), which is used to identify regions of the scene image that are relevant for compatibility detection. For example, in an outdoor image, the attention mechanism may learn to focus on the person in the image and their outfit, while ignoring 'background' objects in the surrounding scene.

Note that the above is ultimately not a *personalized* model, i.e., there are no parameters associated with the individual users. However the system is arguably more useful than that of Section 8.3.1 from a personalization perspective, in the sense that it would allow a user to upload an image of themselves and receive recommendations that complement their personal style. As such, it is an example of the type of *contextual* or non-parametric personalization that we discussed in **??**.

In addition to experiments on personalized fashion, Kang et al. (2019b) shows that the same technique can be used for complementary item recommendation in other settings, such as recommending compatible furniture based on items in a living room.

8.3.3 Generating Fashionable Wardrobes

The papers above consider *pairwise* compatibility among items as a proxy for selecting sets of items that will form compatible outfits.

Hsiao and Grauman (2018) considers the more challenging problem of finding *collections* of items that can be used to generate compatible outfits (called 'capsule wardrobes'). They note a general limitation of of much of the above work in terms of the difficulty and noise involved in collecting pairwise training data

Their specific notion of a 'wardrobe' is a set of items belonging to fixed sets of categories (or 'layers'), e.g. tops, bottoms, outerwear. The suggested quality of good a wardrobe is that it should be capable of generating many outfits, i.e., a large number of sets of items should be mutually compatible. At the same time, a wardrobe should have a wide *variety* of items (e.g. many nearly-

identical pairs of jeans and t-shirts would generate many outfits but would not be a good wardrobe).

First Hsiao and Grauman (2018) defines a measure to determine the compatibility of items in a single outfit. Their specific approach is based on *topic modeling* which we studied a little in Section 7.3.1. Essentially, an outfit is represented by a low-dimensional vector; each dimension of this vector in turn corresponds to a mixture over certain visual attributes (e.g. an outfit dimension might correspond to 'floral patterns,' and this dimension might be associated with visual attributes describing appearance, shape, cut, etc.).[1] A set of items is said to constitute a good outfit if their collective attributes resemble those of training outfits.

Hsiao and Grauman (2018) explores the relationship between these interacting components (outfits, wardrobes, versatility, as well as personalization of these components), and develops optimization schemes to circumvent the combinatorial nature of the problem.

Perhaps most important contributions in Hsiao and Grauman (2018) are simply the creative use of training data, and the formalization of what it means for a set of items to be 'good' in terms of compatibility. Hsiao and Grauman (2018) make the argument that this type of model and training procedure is preferable to other techniques that rely on e.g. pairwise compatibility relationships; a topic model-based approach results in a holistic notion about the overall qualities of an outfit, and allows for training on complete outfit images rather (e.g.) collecting training data from co-purchases (as in **??**), which may be subject to noise or otherwise not representative of real fashion compatibility.

8.3.4 Domains other than Fashion

The visual dynamics of fashion items are often the focus of studies on personalized visual models, though fashion is not the only domain in which visual features play a key role.

A few others have sought to study personalized visual dynamics in related domains. Bell and Bala (2015) learned models of visual compatibility of home decor items collected from *Houzz*; the model is similar to the fashion compatibility model we studied in **??**, with the main goal being to learn a similarity function between images via a Siamese network. At training time, the problem is essentially cast as a form of visual search: that is, given a scene containing an item, identify the item. This is achieved by training on a dataset of image pairs, where one image consists of an item in a scene while the other consists of a

[1] 'Outfits' and 'attributes' are analogous to 'documents' and 'words' in the original topic model formulation Blei et al. (2003).

clean (or 'iconic') product image. As such, the learned distance metric simply attempts to map both *in-situ* images and clean images to the same point. Although the main application for this task is visual search (i.e., find the item in this image, or find images containing this item), other potential applications are discussed that make use of the learned metric, such as identifying stylistically compatible items across categories.

Kang et al. (2019b) also adapted their model (which we studied in **??**) to the problem of furniture / home decor recommendation. The technique is essentially the same as that described in **??**, but is trained on a dataset of interior design and home decor items from *Pinterest*. Here, rather than estimating a clothing item which completes an outfit (by withholding that item from the scene during training), the method is used to identify home decor items which are visually compatible (or complementary to) others in the scene.

He et al. (2016b) considered visual dynamics in the context of *art* recommendation. Their setting is an online art community (*Behance*), where personalized preferences can potentially be guided by a variety of visual, temporal, and social factors. Spiritually, the modelling approach is similar to that used to model the temporal dynamics of fashion (as in **??**), i.e., by combining pre-trained image embeddings with a variety of application-specific temporal dynamics. The main component of their temporal model is a Markov chain-based approach (as in **??**), whereby interactions with are largely guided by recent context. He et al. (2016b) also observe that art recommendation has a significant social component, whereby preferences can be estimated based on the identity of the artist as much as from the art itself.

8.3.5 Other Techniques for Substitutable and Complementary Product Recommendation

While much of the work on substitutable and complementary item recommendation is motivated by applications in fashion (where such recommendations can naturally be used to generate outfits, etc.), a few others have considered the problem more broadly, either to consider different modalities of data (besides visual features), or to consider complementarity in settings other than fashion.

Substitutes and Complements from Text McAuley et al. (2015b) attempted to estimate substitutable and complementary products using the same dataset of *Amazon* co-purchase and co-browsing data as in **??**

$$\underbrace{\sum_{(i,j)\in C} \log \sigma(\gamma_i \cdot \gamma_j)}_{\text{complementary pairs}} + \overbrace{\sum_{(i,j)\in C^-} \log \sigma(\gamma_i \cdot \gamma_j)}^{\text{non-complementary pairs}} + \lambda \underbrace{\sum_{(u,i)\in \mathcal{T}} \sum_{w\in d_{u,i}} \log p(w|\gamma_u, \psi)}_{\text{topic likelihood}}, \quad (8.10)$$

Learning Asymmetric Product Relationships Wang et al. (2018) also studied the problem of recommending substitutable and complementary products, proposing several modifications to the approach used in McAuley et al. (2015b).

First, as we saw in **??**, a good model for complementary product recommendation ought to recognize that 'complementarity' should ideally not be captured by a similarity function. That is, complementary items have systematically *different* characteristics, and in particular an item is not complementary with itself.

Following this logic, a simple modification of the complementarity function in Equation (8.10) consists of using two sets of factors γ_i and γ'_j for complementary pairs i and j:[2]

$$\underbrace{\sum_{(i,j)\in C} \log \sigma(\gamma_i \cdot \gamma'_j)}_{\text{complementary pairs}} + \overbrace{\sum_{(i,j)\in C^-} \log \sigma(\gamma_i \cdot \gamma'_j)}^{\text{non-complementary pairs}} \quad (8.11)$$

Following this modification, several other extensions are proposed, mostly to handle data sparsity issues and cold-start scenarios. Whereas McAuley et al. (2015b) did so using signals from text, Wang et al. (2018) leverages knowledge about the specific semantics of substitutable items; for example, if complementary pairs tend to belong to specific sub-categories (e.g. shirts are often compatible with jeans), this acts as a weak signal that other products within these categories are also complementary. Likewise, substitutable relationships might be *transitive* (for example), i.e., if i is substitutable for j, and j is substitutable for k, then i is likely substitutable for k; various 'soft' constraints of these types ultimately help to improve performance of the model.

[2] Note that in practice, complementary pairs $i \rightarrow j$ are *directed*, e.g. a large fraction of camera purchases are paired with a memory card, whereas only a small fraction of memory card purchases are paired with a camera.

Diversifying Complementary Product Recommendation Although complementary recommendations are intended to be distinct from the query item, this does not necessarily mean that recommended complements will be distinct from *each other*. For example, it would presumably not be useful to recommend only t-shirts as a complement for a pair of jeans; instead, one might wish to recommend a combination of shirts, belts, shoes, etc.

Although we'll revisit the notion of diversity in depth in Chapter 9, here we discuss some approaches that consider the problem within the specific context of complementary item recommendation.

He et al. (2016a) considered that a good set of complementary items might be represented as a *mixture* over different notions of compatibility. They start with a simple pairwise compatibility model between items i and j of the form

$$d_c(i, j) = \|\gamma_i - \gamma_j^{(c)}\|_2^2 \tag{8.12}$$

where γ_i and $\gamma_j^{(c)}$ are based on image embeddings. This is similar to the model of **??**, though includes *separate* embeddings γ and $\gamma^{(c)}$ for the 'query' item i and complementary item j; using separate embeddings breaks the symmetry between i and j, which is desirable for complementary items (e.g. an item should not be complementary with itself).[3]

Next, He et al. (2016a) notes that Equation (8.12) captures only a single notion of compatibility; if a model is trained using such a function, this will presumably correspond to the predominant mode of compatibility in the data, but will not be diverse. To address this, He et al. (2016a) proposes treat the compatibility relationships in the data as a probabilistic mixture over several competing notions. This idea borrows from a mathematical framework known as a *mixture of experts* Jacobs et al. (1991). Specifically:

$$d(i, j) = \sum_{c=1}^{C} \underbrace{p(c|i)}_{\text{relevance of compatibility function } c \text{ to query } i} \cdot d_c(i, j). \tag{8.13}$$

Here, $p(c|i)$ measures which types of compatibility relationships $d_c(i, j)$ are most likely to be relevant for a query item i; while written as a probability, this can more simply be thought of as a function that is used to combine compatibility relationships with different weights. Here

$$p(c|i) = \frac{\exp(\theta_k f_i)}{\sum_{k'} \exp(\theta_k f_{i'})}, \tag{8.14}$$

where f_i is a feature vector describing the image i, and $theta_k$ is a parameter vector.

[3] As such, Equation (8.12) is no longer a distance function.

Ultimately, this model learns C separate embeddings $\gamma_i^{(c)}$ for each item (along with the query embedding γ_i), each corresponding to a different notion of compatibility or 'complementarity.' In principle, this means the model is able to capture several different modes of compatibility that interact simultaneously. At test time, diverse lists of compatible items can generated by sampling from different compatibility functions $d_c(i, j)$ according to their relevance $p(c|i)$.

Incorporating Product Types Hao et al. (n.d.) noted that both accuracy and diversity of complementary product recommendation can be achieved my making explicit use of available category data. Instead of directly predicting which items j are compatible with a query item i, the approach first attempts to estimate which of several *categories* are relevant to a given query; following this, the method generates complements via several category-specific compatibility functions, similar to $d_c(i, j)$ form Equation (8.12).

8.3.6 Implementing a Compatibility Model in *Tensorflow*

Most of the compatibility models we saw in the previous section are relatively straightforward to implement on top of pre-trained image features.

Below we assume a feature matrix X such that X_i is an image feature describing item i (e.g. a feature from ImageNet as in He and McAuley (2015)), and that each pair (i, j) is associated with a label y determining whether the pair is compatible ($y = 1$) or not ($y = 0$). The model then projects the images into 'style space' via $\gamma_i = E_i x_i$ and $\gamma_j = E_j x_j$; here we use two separate embeddings so that the model can learn asymmetric relationships. Finally compatibility is evaluated via $\sigma(c - d(\gamma_i, \gamma_j))$:

```
class CompatibilityModel(tf.keras.Model):
    def __init__(self):
        super(CompatibilityModel, self).__init__()
        # Embeddings for the query item (Ei) and target item (Ej)
        self.Ei = tf.Variable(tf.random.normal([featDim,styleDim],
            stddev=0.001))
        self.Ej = tf.Variable(tf.random.normal([featDim,styleDim],
            stddev=0.001))
        # Offset term as in ??
        self.c = tf.Variable(0.0)

    # Given image features x1 and x2, and label y (0/1)
    def call(self, xi, xj, y):
        # Style-space embeddings γ_i and γ_j
        gammai = tf.matmul(xi, self.Ei)
        gammaj = tf.matmul(xj, self.Ej)
        # Shorthand for ??
        return -tf.math.log(tf.math.sigmoid(self.c - tf.math.
            squared_difference(gammai,gammaj))*(2*y - 1) - y + 1)

model = CompatibilityModel()
```

Similarly, we could modify the code to use the compatibility function of **??**:

```
def call(self, x1, x2, y):
    s1 = tf.matmul(x1, self.E1)
    s2 = tf.matmul(x2, self.E2)
    return -tf.math.log(tf.math.sigmoid(self.c - tf.matmul(s1,tf.
        transpose(s2)))*(2*y - 1) - y + 1)
```

Finally, we train the model by sampling compatible and incompatible pairs from our training set and computing gradients:[4]

```
def trainingStep(compat):
    with tf.GradientTape() as tape:
        (i1,i2,y) = random.choice(compatiblePairs)
        x1,x2 = X[i1],X[i2]
        objective = model(x1,x2,y)
    gradients = tape.gradient(objective, model.trainable_variables
        )
    optimizer.apply_gradients(zip(gradients, model.
        trainable_variables))
```

[4] Although this example is simple enough to allow reasonably fast training, there are several ways that this code could be made more efficient, e.g. the embeddings of all images in X could be computed simultaneously.

8.4 Personalized Generative Models of Images

In Chapter 7, we examined personalized text models from two directions: first, we used text within *predictive* tasks, e.g. we saw how text can be used for regression problems (**??**), and to improve the performance of recommender systems (**??**). Second, we saw how to personalize *generative* models of text (**??**), i.e., to generate text that matches a user's writing style or preferences.

Likewise, our discussion of visual data has so far considered using images to improve predictive performance; it is worth spending a little time exploring how to personalize *generative* models of images.

Briefly, the basic framework we will consider extending is that of the *Generative Adversarial Network*, or GAN.

Generative Adversarial Networks (GANs) are an unsupervised learning framework in which two components 'compete' to generate realistic looking outputs (in particular, images) Goodfellow et al. (2014). One component (a generator) is trained to generate images, while another (a discriminator) is trained to distinguish real versus generated images. Thus the generated images are trained to look 'realistic' in the sense that they are indistinguishable from those in the dataset.

Such systems can also be conditioned on additional inputs, in order to sample outputs with certain characteristics Mirza and Osindero (2014).

The basic setup of a Generative Adversarial Network is depicted in Figure 8.2. Here x_{real} is an image sampled from a dataset, whereas x_{gen} is a synthetically generated image. The discriminator $D(x)$ is then given an image x, and is responsible for predicting whether it is a sample from the dataset or is a synthetic image. For image generation, the discriminator is typically be a form of convolutional neural network (CNN), whereas the generator is a series of *deconvolution* operators, essentially operating via a similar principle as the CNN, but in reverse. The generator takes as input a latent code z, a random input which allows the generator to produce distinct images (z is essentially a *manifold* which describes patterns of variation in image data so as to capture the variability in the training dataset). The discriminator D and generator G are trained simultaneously, such that the generator gradually becomes better at generating images that are capable of 'fooling' a better and better discriminator.

The above type of architecture has been used to generate various types of realistic images, including artwork, clothing, and human faces. Kang et al. (2017) sought to develop *personalized* GANs, that would generate images that capture the preferences of individual users.

The approach from Kang et al. (2017) essentially combines the GAN frame-

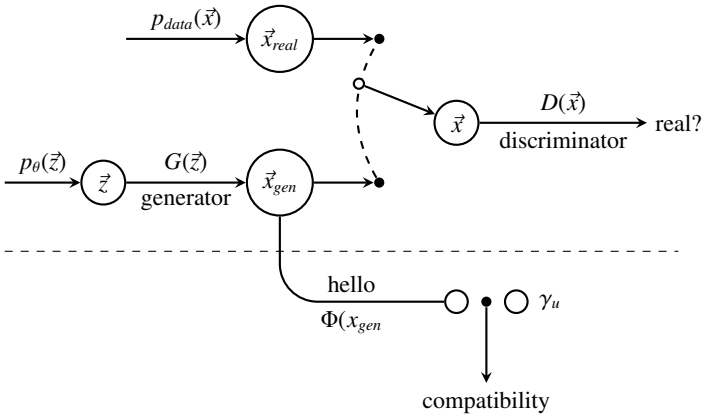


Figure 8.2 Basic setup of a Generative Adversarial Network (GAN), and a personalized GAN. The components above the dotted line depict the 'standard' GAN setup, in which a generator 'competes' with a descriminator to generate images that are indistinguishable from real data. Components below the dotted line are used to develop a 'personalized' GAN that generates images compatible with a particular user.

work with a personalized image preference model, similar to that of Visual Bayesian Personalized Ranking (**??**). The basic idea is depicted in Figure 8.2 (bottom). Once a GAN is trained (as above), the image $G(z)$ generated by a given latent code is passed to both the discriminator (D) as well as a personalized preference model. For the preference model, the image is represented via $\Phi(G(z))$; this is analogous to γ_i, or $\mathbf{E}f_i$ as in **??**, though unlike VBPR the synthetic image is not associated with any particular item i. A user's preference toward this synthetic item is then estimated via $\gamma_u \cdot \phi(G(z))$. Ultimately, the objective is that a generated image should be simultanously plausible (according to $D(G(z))$), but also desirable to the user (according to $\gamma_u \cdot \Phi(G(z))$):

$$\arg\max_z \underbrace{\gamma_u \cdot \Phi(G(z))}_{\text{user preference toward generated image}} - \overbrace{\eta(D(G(z)) - 1)^2}^{\text{'plausability' of generated image}}. \tag{8.15}$$

Kang et al. (2017) argues that a model such as that above can be used in several ways. Most straightforwardly, it can be used to generate designs (or images) that match the preferences of individual users; Equation (8.15) can be straightforwardly modified to find optimal designs for a *population* of users (e.g. by taking an average $\sum_u \gamma_u \cdot \Phi(G(z))$). Alternately, given an existing image (rather than a generated image $G(z)$), Equation (8.15) (or rather, its gradient)

can be used to suggest *local modifications* to the image that will make it preferable to a user or population.

8.5 Personalized Image Search and Retrieval

Having studied the use of visual data in the context of recommendation, it is worth briefly considering how visual data is handled in 'traditional' settings like image search and retrieval.

Paper: Image ranking based on user browsing behavior

Trevisiol et al. (2012)

A basic concept in image retrieval settings, as we saw in **?**, is that of learning a joint embedding space between a query q and an image i:

$$d(q, i) = \|g(q) - g(i)\|_2^2 \tag{8.16}$$

For example, in Pan et al. (2014), $f(q)$ and $f(i)$ were based on simple linear embeddings of (textual) query features f_q and (visual) image features f_i:

$$g(q) = f_q W^{(\text{query})}; \quad g(i) = f_i W^{(\text{image})}. \tag{8.17}$$

Note that this setting is very similar to that of Section 8.3.1, in which a query image is linearly projected into 'style space' (Equation (8.5)) so that neighboring images can be retrieved. This type of image retrieval system operates on the same principle, except that the query features are extracted from text (as we studied in **??**).

Finally, $W^{(\text{query})}$ and $W^{(\text{image})}$ are trained so that distances in Equation (8.16) are minimized based on click-through data, again following a learning approach similar to that from Section 8.3.1. That is, distances should be small between query/image pairs associated with a large number of clicks.[5]

Paper: Learning to personalize trending image search suggestion

Paper: Prism: Concept-preserving social image search results summarization (maybe not...)

8.6 History and Related Work

Personalized recommendation problems involving visual data have been studied for several years. Initial attempts ignore visual data altogether, and long predate solutions that model images explicitly. For example, an early system

[5] Note that this approach assumes that query and click data is available at training time, presumably from a method *not* based on visual embeddings.

for clothing recommendation Hu et al. (2014) learns a user's 'style' in order to recommend clothing, but does so using 'likes' rather than any analysis of visual features. Likewise, *YouTube's* early recommendation approaches Davidson et al. (2010) are based on heuristic 'relatedness scores' based on co-visitation (essentially a form of neighborhood-based approach, as in Section 4.3), though some features based on video metadata are included in the model; newer solutions (based on deep learning) adopt more complicated candidate generation and ranking strategies, though again make little if any use of explicit visual signals Covington et al. (2016).

8.7 Exercises

Exercises

8.1 Starting from the code from **??**, and using a small dataset of compatible (and non-compatible) pairs (e.g. from *Amazon Baby* products), set up a pipeline for estimating compatibility relationships (e.g. 'also bought' or 'also viewed' products). Tune the model (e.g. in terms of the number of embedding dimensions) and measure its accuracy (in terms of its ability to successfully distinguish compatible from non-compatible items).

8.2 The model from **??** 8.1 is based on a distance (or similarity) function of the form $d(i, j) = \|\gamma_i - \gamma'_j\|_2^2$. Note that there is nothing particularly special about this specific choice of similarity function.

8.3 A challenging aspect of learning a compatibility model between images is that of generating negative samples (i.e., pairs of items believed to be incompatible)

8.4 Although models based on visual features are useful in cold-start settings, there is in principle no reason that

8.5 When studying notions like fashion compatibility (or indeed, any latent item representation in a recommender system), it is worth exploring whether the learned representations semantically correspond to our intuitive notion of similarity. To assess this, it helps to visualize latent item representations γ_i in two dimensions (for the sake of plotting them). In Figure 7.6 we did this simply by learning two-dimensional item representations, though by doing so we are likely visualizing a suboptimal model. Various techniques exist

McInnes et al. (2018)

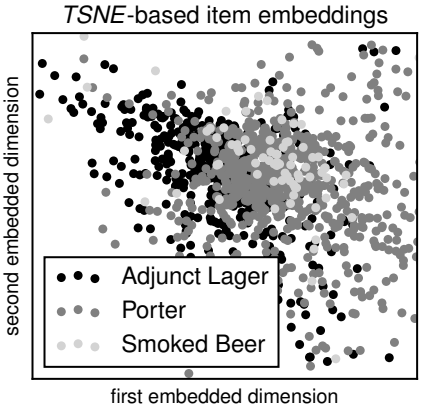


Figure 8.3: Ten-dimensional item representations (γ_i) embedded into two dimensions via *TSNE*; this is a ten-dimensional version of the model from Figure 7.6.

```
1 import numpy as np
2 from sklearn.manifold import TSNE
3
4 X_embedded = TSNE(n_components=2).fit_transform(X) # X is a
      matrix of all item representations γ_i
```

8.7.1 Project 6: Generating Compatible Outfits

First, consider how you would generate a training dataset of compatible items. It is probably most realistic to start by considering *pairwise* compatibility, i.e., to generate a training dataset of pairs of items (i, j) that are mutually compatible (as in **??** or **??**). Even then, several options are available for mining pairwise compatibility data. For example:

- Co-purchase relationships (e.g. 'people who bought i also bought j'), as in **?**.
- Directly mining the co-purchased items from user interaction histories (e.g. if a user u purchased both items i and j, this is an indication that they might be compatible). This strategy was also explored in **?**.
- Both of the above approaches are highly noisy, as items are not necessarily co-purchased with the intention of being worn together. A third approach consists of mining explicit relationships from actual outfit data. e.g.

Consider the advantages and disadvantages of each of the above approaches. For example, which will allow you to collect the most data, and which will be

the least noisy? Further consider how you should select samples, e.g. you probably want to avoid pairs (i, j) where both items belong to the same category, or might further restrict your dataset to certain categories of interest.

Similarly, you should choose an appropriate strategy to generate *negative* samples for training, i.e., pairs (i, j) that do *not* go together. Trivially, such samples could be generated from pairs of random items, though more 'difficult' negatives could be generated by selecting pairs with specific categories.

Having built your dataset, there are several potentially interesting directions for study. For example:

- What is an appropriate model to use to estimate compatibility relationships? A good starting point may a model such as that from **??**, since compatibility relationships are likely to be asymmetrical in this context.
- Consider whether it is worthwhile to incorporate visual features to estimate compatibility via an embedding strategy (e.g. following the code from **??**) or whether it is sufficient to model compatibility in a latent space (e.g. as in **??**).
- Consider whether it is useful to incorporate other features, such as brands, prices, features from text, etc.
- (harder) Is there value to training a *personalized* model for this task, i.e., rather than predicting whether a pair of items (i, j) are compatible, can you predict whether i and j are compatible *for a particular user u*. Think carefully whether the identity of the user explains a significant amount of variation in compatibility relations, and whether enough data can be mined to fit a personalized model.

Finally, consider ways to visualize the model (or its predictions), either by representing items in a low-dimensional space (as in **??**), or by building a simple interface to explore compatible items.

Note that other than the use of visual features, the above steps could be used to build item-to-item compatibility models for *any* types of data (e.g. dishes in a menu, songs in a playlist), and are not limited to outfit generation.

9

The Ethics of Personalized Machine Learning

9.1 How Can Personalized Machine Learning be Improved?

So far, we have largely viewed personalized machine learning as a 'black-box' machine learning task. That is, given a user, their context, and some potential stimulus, can we estimate how the user will react to that stimulus?

This black-box view of machine learning, while effective for building accurate models, ignores the potential real-world consequences of how such models are applied.

Broadly, the dangers of blindly applying machine learning models are well-studied: ML algorithms can perpetuate, mask, or amplify biases in training data, or have low accuracy for underrepresented groups. Detecting and mitigating these types of biases largely describes the study of 'fairness' in machine learning (see e.g. Dwork et al. (2012)).

Within the context of *personalized* machine learning, black-box models, if applied carelessly, can also hide or amplify biases or other issues. Below we highlight a few examples, to be studied throughout this chapter:

- A recommender system, although ostensibly designed to aid *discovery*, may actually have a 'concentration' effect, where users are gradually locked into a 'filter bubble' containing only a narrow set of items (Section 9.2.1).
- Alternately, by recommending content that maximally aligns with a user's interests, a system may gradually push them toward more and more 'extreme' content (Section 9.2.1).
- Recommender systems may have reduced utility for users (or groups of users) who are underrepresented in the training data, for instance 'popular' items that are widely recommended may merely reflect what is popular among the majority group (**??**).

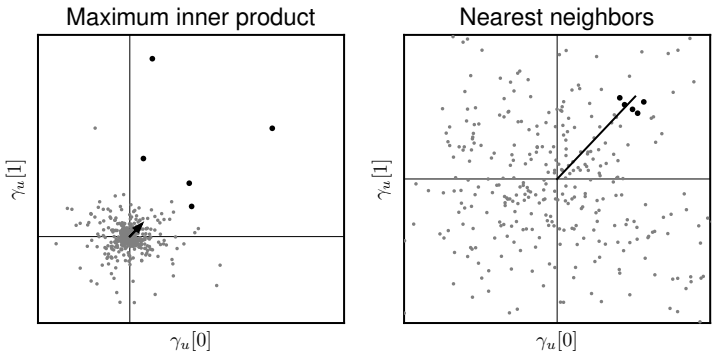


Figure 9.1 Recommendations selected for a user by maximizing an inner product (left) or taking nearest neighbors (right).

- Recommendations may focus only a user's predominant interest, while failing to capture the diversity of breadth of their interactions (**??**).
- Likewise, systems could disadvantage vendors (or content creators, etc.) by failing to recommend products in the long tail (**??**).

As an example to conceptually demonstrate some of the above issues, Figure 9.1 highlights the ways in which a recommender system, if applied naïvely, may lead to a 'concentration' or 'extremification' effect. At left we show personalized recommendations generated by maximizing an inner product ($\gamma_u \cdot \gamma_i$), as in Section 4.5; at right we show item-to-item recommendations generated by finding similar items (i.e., nearest neighbors of γ_i).

When maximizing an inner product (Figure 9.1, left), the recommended items are on the 'fringe' of the item space; roughly speaking, if I know that a user likes action movies (for example), then I might recommend movies with *the most* action. While this might make sense in the context of movie recommendation, when recommending (e.g.) political videos on *YouTube*, such a strategy may drive users toward fringe or 'extreme' content. Alternately, when choosing nearest neighbors (Figure 9.1, right), users' recommendations are concentrated around very similar content, which may lead to a 'filter-bubble' effect.

While the above is merely a conceptual demonstration, in the following sections we'll examine empirical studies (e.g. from *YouTube* and *Facebook*) that analyze filter bubbles and extremification in the context of deployed recommendation settings. We'll also look further into issues of diversity, bias and fairness, exploring a wide variety of potential consequences of personalized model training.

These ideas connect to the broader topic of fairness and bias in machine learning, though the issues in personalized settings are quite different. Much of our focus when introducing these issues is to present strategies to address them, in order to build personalized models that are more diverse, unbiased, and fair.

9.2 Measuring Diversity

Before exploring case studies measuring the effect of recommendations on diversity, and techniques to improve diversity, we first briefly consider the kind of measurements we'll use to assess diversity. We'll look at two main concepts: First, across *all* users, do recommended items follow the same *distribution* as the set of items that were consumed? Second, among individual users, are the recommended items more or less diverse than their historical consumption trends?

We first train a recommender (in this case using comic books from *Goodreads*), following the code presented in **??** (i.e., using a model based on Bayesian Personalized Ranking).

Next we generate a set of example recommendations from the model and compare those with the original interaction data. For each user, we generate as many recommendations as they have interactions in the original data, so that each user is represented the same number of times in both our interaction data and our recommendation data:

```
countsPerItem = defaultdict(int)

for u in range(nUsers):
    # Given a matrix of interactions X, as in ??
    recs = model.recommend(u, X, N = len(itemsPerUser[u]))
    for i, score in recs:
        countsPerItem[i] += 1
```

Next we compare the measurements above to the same measurements derived from the interaction data. In particular, do popular items (based on the number of historical interactions) frequently appear among recommendations; and conversely, are frequently recommended items popular? We plot these comparisons in Figure 9.2.

Surprisingly, neither of the two appear to match very closely (i.e., popular items do not get recommended frequently, and *vice versa*).

There are various ways we could formally measure the discrepancy between these two distributions, or compute summary statistics that help us to compare them. In the context of item recommendations, we might be interested

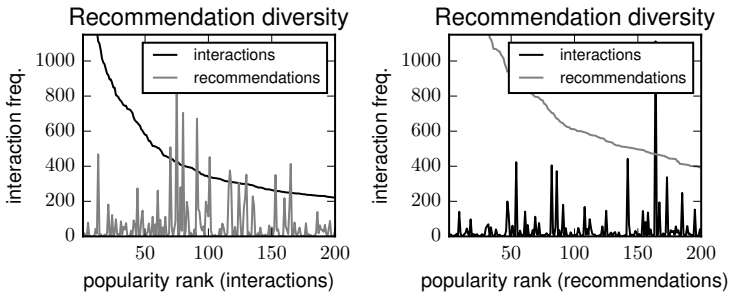


Figure 9.2 Distribution of interactions compared to recommendations (based on an implicit-feedback model trained on comic books from *Goodreads*. The left plot measures the *recommendation* frequency of the 200 most popular items (as measured by the number of interactions in the training set); the right plot measures the *interaction* frequency of the 200 most recommended items.

in whether one of the two distributions is more *concentrated* than the other, i.e., whether recommendation frequencies are highly 'peaked' around a few popular items (versus a flatter, or longer-tailed distribution). One measure (that we'll see used in several of the studies below) is the *Gini coefficient*, a measure of statistical dispersion. Given a set of measurements y (in this case, frequencies associated with each item), the Gini coefficient measures the *average (absolute) difference between frequencies*, i.e.,:

$$G(y) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} |y_i - y_j|}{2N^2 \bar{y}}. \tag{9.1}$$

Highly concentrated data will have a large coefficient, while flatter distributions will have $G(y)$ close to zero (the presence of $2\bar{y}$ in the denominator scales the expression to be in the range $[0, 1]$).

In practice the coefficient is usually computed approximately, i.e., by sampling rather than enumerating all possible pairs of items:

```
1  def gini(y, samples=1000000):
2      m = sum(y) / len(y) # average
3      denom = 2 * samples * m
4      numer = 0
5      for _ in range(samples):
6          i = random.choice(y)
7          j = random.choice(y)
8          numer += math.fabs(i - j)
9      return numer / denom
```

In the case of this particular experiment, the interaction data yields a Gini coefficient of $G = 0.72$ while the recommendations yield $G = 0.80$. In other

words, this particular recommendation algorithm has resulted in recommendation that are somewhat more concentrated compared to historical interaction data.

9.2.1 Filter Bubbles and Extremification

Pariser (2011) (coins the term filter bubble)

The idea that personalized recommendation traps users in 'filter bubbles,' amplifies existing biases toward popular items, or guide users toward extreme content, are often reported in popular media and discussed anecdotally, though such concepts are often not precisely defined. It is also difficult to measure these types of dynamics empirically, as one rarely has the ability to analyze the counterfactual scenario in which the recommender wasn't in place.

Exploring Diversity through Simulation An early paper that attempted to define and analyze the impact that recommender systems have on interaction diversity did so via simulation Fleder and Hosanagar (2009). They noted the existence of two competing hypotheses as to why recommender systems might encourage or discourage diversity: on the one hand, recommender systems can guide content *discovery*, which can increase the diversity of item interactions Brynjolfsson et al. (2006); on the other hand, recommender systems might reinforce the popularity of already-popular products, thus reducing diversity Mooney and Roy (2000). Their attempt to resolve this question built a simple simulation which generates recommendations, in which the probability of an item being recommended, or the probability of a user accepting it, can be controlled. By varying the controllable parameters, they show that under almost all conditions (i.e., except in edge cases), the recommender system leads to a concentration effect (i.e., results in a reduction in interaction diversity), as measured by the Gini coefficient.

Of course, the above is not necessarily true of *every* recommender system; indeed as we'll see in **??**, one can design a recommender system so as to explicitly target the diversity of recommended items. Rather, the above result is simply a demonstration that under fairly minimal conditions, recommender systems *can* lead to a concentration effect.

Directly Measuring Algorithmic Recommendation and Diversity Following Fleder and Hosanagar (2009), which studied the possibility of filter bubbles via simulation, Nguyen et al. (2014) was the first to empirically measure the effect of recommender systems on content diversity in a real setting.

The research questions in Fleder and Hosanagar (2009) are similar to the

ones studied via the simulations above, namely: do recommender systems gradually expose users to narrower content over time; and, how does this effect vary as a function of how receptive users are to recommendations.

Exploring Diversity through *YouTube* Recommendations Following this work, Zhou et al. (2010) conducted an empirical study of recommendations on *YouTube*, and argued that *YouTube's* recommendations (specifically the 'related videos' feature) have a positive impact on content diversity. They showed that recommendations drive a large fraction of views on youtube, and that views driven by recommendations have higher diversity than views from a popularity-driven system (however, it is hard to argue that recommendation-driven views are therefore 'diverse,' given that popularity-driven views would presumably lead to high concentration, and as such are arguably a weak baseline).

Diversity and Extremification However, even if consumption is 'diverse,' as defined by the overall distribution of interactions across an item vocabulary, this does not guarantee that users are necessarily exposed to diverse viewpoints, or are not pushed toward extreme content. For example, returning to the motivating example from Figure 9.1, recommendations based on maximizing an inner product (left) would expose users to more 'diverse' content than the nearest neighbor model (right), but may gradually drive users toward 'fringe' items that are aligned with, but are more extreme versions of, the content they initially consumed.

Ribeiro et al. (2020) attempted to empirically analyze the pathways via which users arrive at extreme content on *YouTube*. The authors used curated lists of channels (for their study, of alt-right political channels) in order to establish a ground-truth of what is meant by 'extreme' content; they also collected less extreme content ('alt-lite,' general media, etc.)

9.3 Diversification Techniques

one can imagine how recommendations based on nearest neighbors or maximum inner-products, as in Figure 9.1, might focus exclusively on a user's predominant interest alone.

Bias Disparity in Recommendation Systems

Diversifying Search Results (Agarwal)

9.3.1 Maximal Marginal Relevance

A simple notion of diversity that is commonly used in document retrieval scenarios is that among a (ranked) list of retrieved documents, each retrieved item should simultaneously be relevant, but at the same time not too similar to the already-returned items.

This notion is captured by the *Maximal Marginal Relevance* Carbonell and Goldstein (1998) procedure (MMR). The approach was originally designed for retrieving sets of text passages that best summarize a document (with respect to some query): each document should be similar to the query, but also dissimilar from the documents already retrieved.

The same concept can straightforwardly be applied in recommendation scenarios, given that we have notions of both *relevance* and *similarity* available (e.g. relevance might be the output of a latent factor model, while similarity could be defined in terms of cosine similarity, or as an inner product between item representations γ_i and γ_j for two products).

To apply the concept to recommendation, we would define the Maximal Marginal Relevance as follows:

$$MMR = \arg\max_{i \in R \setminus S} \Big[\lambda \underbrace{Sim^{\text{user}}(i, u)}_{\text{relevance to the user}} - (1 - \lambda) \overbrace{\max_{j \in S} Sim^{\text{item}}(i, j)}^{\text{similarity to already-recommended items}} \Big], \tag{9.2}$$

where R is an initial candidate set of recommendations (most trivially e.g. a list of items the user hasn't already interacted with), and S is a set of items retrieved so far. Sim^{user} and Sim^{item} are item-to-user and item-to-item similarity functions (respectively); the former is presumably the compatibility function returned by a recommender system, the latter is any item-to-item similarity measure (e.g. cosine similarity or inner product between item representations).

Note that the above is computed iteratively, that is we add one result at a time by maximizing the MMR until the list S has the desired size. Finally, λ trades off the extent to which we care about compatibility versus diversity.

9.3.2 Re-ranking Approaches to Diverse Recommendation

Similar to Maximal Marginal Relevance, several approaches have been specifically designed for re-ranking in recommendation scenarios. *Re-ranking* approaches, including MMR, assume that we are given an initial ranking function which we trust to find items of high relevance, but which lack diversity; thus one seeks a re-ranking procedure that balances the two concerns.

One such approach for re-ranking for recommendation was proposed in Adomavicius and Kwon (2011). The approach assumes the presence of three

components: First, they assume the availability of a compatibility score, e.g. a rating prediction $r(u, i)$. Second, a *relevance-oriented* ranking technique, $rank_u(i)$; this could be any ranking function, though most trivially one might simply order predictions by a predicted rating score $r(u, i)$. And third, a second 'diversity oriented' ranking function; conceptually, this should focus on recommending items to users that they would *not* normally consider.

An example of a diversity-oriented loss from Adomavicius and Kwon (2011) is to sort items by popularity, with the *least* popular items being ranked first:

$$rank^{(\text{pop})} = |U_i|. \tag{9.3}$$

Recommending *unpopular* items does not at first appear to be a particularly effective recommendation strategy; however this ranking is used in conjunction with the prediction score $r(u, i)$. Specifically, to encourage diversity we want to find *unpopular* items that *this* user is likely to enjoy. Spiritually, this is somewhat reminiscent of our *TF-IDF* approach to finding important words in documents (Section 7.1.3).

The specific (re-)ranking objective from Adomavicius and Kwon (2011) then looks like

$$rank'_u(i, t) = \begin{cases} rank^{(\text{pop})}(i) & \text{if } r(u, i) \geq t \\ \alpha_u + rank_u(i) & \text{otherwise} \end{cases}. \tag{9.4}$$

Here t is a threshold term, essentially determining whether one of the low-popularity recommendations has a high enough score to recommend; α_u is an offset term ensuring that the popularity-based recommendations appear first in the ranking before those of $rank_u(i)$.

Adomavicius and Kwon (2011) shows that as the threshold t is changed, the system gradually trades-off between recommendation precision and diversity. Several different ranking functions are considered, for example to replace popularity-based ranking by alternatives based on the average rating, rating variance, etc.

They also note that the type of diversity achieved by this ranking mechanism is quite different from that in Section 9.3.1, as it does nothing to encourage diversity (or dissimilarity) among an individual user's item list. Instead, they discuss the related notion of *aggregate diversity*, which defines diversity across the item vocabulary itself, i.e., recommendations across *all* users should have reasonable coverage of the complete item vocabularity. This is related to the notion of *P*-fairness that we'll discuss in Section 9.6.1.

9.3.3 Topic Diversification

Improving recommendation lists through topic diversification Ziegler et al. (2005)

Avoiding Monotony: Improving the Diversity of Recommendation Lists Zhang and Hurley (2008)

Category-Driven Approach for Local Related Business Recommendations

9.4 Other Metrics Beyond Accuracy: Novelty, Discovery, and Serendipity

So far, we have considered diversity in terms of the trade-off between recommending the highest *relevance* items (e.g. highest click probability) while ensuring that recommended items are not too similar to each other. Other than relevance, diversity among items is only one desirable alternative.

Besides relevance and diversity, other desirable features of a recommendation list might include:

- Items should be *novel* to the user, i.e., the recommender system should balance *discovery* of new items against recommending items with high interaction probability, but which are already known to the user.
- Rather than being internally diverse, we might have goals such as mutual compatibility among items, as in **?**.
- Recommended items should have good *coverage*, i.e., they should represent a broad range of categories or features; or they should be *balanced*, in terms of matching the category distribution from the user's history (as in **??**).
- Other goals could be more nebulous, such as perceived unexpectedness, serendipity, or overall user satisfaction.

Kaminskas and Bridge (2016) broadly surveys these alternate optimization criteria for recommneder systems, focusing in particular on diversity, serendipity, novelty, and coverage. We briefly survey some of their main findings (as well as more recent work) below.

Further Perspectives on Diversity Most of the approaches to diverse recommendation in Kaminskas and Bridge (2016) are focused on *reranking* strategies, similar to maximum marginal relevance (Section 9.3.1) and other techniques we've discussed so far. They also discuss other settings where diversity can be useful, such as conversational recommendation (Kelly and Bridge

(2006), see **??**), and the relationship to more traditional work in 'portfolio optimization' from information retrieval **?**.

Vargas and Castells (2011) and Vargas (2011) studied the relationship between novelty and diversity when making recommendations, and presented several metrics to evaluate existing systems in terms of such characteristics.

9.4.1 Serendipity

Various attempts have been made to define 'serendipity' in the context of recommendations. Kaminskas and Bridge (2016) starts with the core property of 'surprise' (i.e., recommendations should be different from one's expectations); Kotkov et al. (2018) state that serendipity should be a combination of relevance, novelty and unexpectedness.

The exact definitions of each of these competing elements is difficult to define

Ge et al. (2010)

Discovery Discovery-oriented Collaborative Filtering for Improving User Satisfaction (not sure)

Unexpectedness Adamopoulos and Tuzhilin (2014) attempt to define the notion of 'unexpectedness' as it relates to recommendations. They note that one cannot target unexpectedness in isolation, or one could trivially generate poor-quality but unexpected recommendations. As such they seek a notion of *utility* that balances unexpectedness against traditional metrics of recommendation quality. They define unexpectedness (for a user u and item i) as a distance between i and the set of items the user u 'expects' to receive. They further assume that there is some optimal value for this distance (which could be different for each user): recommendations that are too expected are uninteresting, while recommendations that are too unexpected will be regarded as irrelevant (quality is defined more straightforwardly in terms of ratings.

Several quantities must then be determined: each user's personal tolerance for unexpectedness, the ideal trade-off between unexpectedness and utility, and finally the definition of what is 'expected.' For the latter Adamopoulos and Tuzhilin (2014) use a definitions based on content similarity in terms of movie attributes (movies with similar attributes are 'expected'). The goal of Adamopoulos and Tuzhilin (2014) is not to fit these values (which are largely subjective quantities) but to evaluate the performance of different recommendation approaches under various hypothetical scenarios. The most promising

finding is that optimizing this type of joint utility need not harm performance compared to methods that target quality exclusively.

Serendipity in Music Recommendation Zhang et al. (2012) considers how music recommendations can be improved by balancing goals of accuracy, diversity, novelty, and serendipity. Their specific approach combines many of the ideas we've seen already: *diversity* is measured in terms of the cosine similarity between items in a recommendation list (as in **??**); *novelty* or 'unexpectedness' is defined in terms of overall item popularity (as in **??**); serendipity (or 'unserendipity,' since low values mean high serendipity) is defined using a novel function, which essentially measures how similar recommended items are to those in the user's interaction history:

$$\text{Unserendipity} = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \frac{1}{|I_u|} \sum_{i \in I_u} \sum_{j \in R_u} \frac{Cos(i, j)}{|R_u|}. \tag{9.5}$$

Where R_u is a set of items recommended to the user (and I_u is the item history for user u). This measure takes a low value if recommended items are on average different from those that appeared in the user's history.

Given these three metrics (diversity, novelty, and serendipity), Zhang et al. (2012) seeks recommendation techniques that can optimize them without overly compromising accuracy. While metrics such as that of Equation (9.5) cannot straightforwardly be incorporated into the optimization scheme directly, various models are designed to ensure that recommendations are topically diverse or belong to distinct clusters. Quantitively, Zhang et al. (2012) study the tradeoff between accuracy, diversity, novelty, and serendipity under different configurations of this model. They also conduct a user study to evaluate the qualitative aspects of the model, revealing that subjective notions of 'subjectivity' and 'usefulness' can be improved without overly harming user enjoyment.

Investigating Serendipity via User Studies Given the ambiguous nature of the precise definition of serendipity, Kotkov et al. (2018) attempts to assess what it means to users via a survey. Kotkov et al. (2018) surveys commonly proposed notions for diversity, ranging from items the user simply hasn't heard of, didn't expect to be recommended, or is highly disimilar to what they usually consume.

PURS: Personalized Unexpected Recommender System for Improving User Satisfaction Li et al. (2020b)

The Impacts of Item Features and User Characteristics on Users' Perceived Serendipity of Recommendations

9.4.2 Determinantal Point Processes

In Section 9.3.1 and **??** we discussed various approaches that attempt to balance accuracy and diversity via what are essentially 'heuristic' strategies that greedily select items that maximize utility while being sufficiently novel compared to the rest.

Determinantal Point Processes (or DPPs) Kulesza and Taskar (2012) are a *set-based* optimization technique that can be used to identify subsets of items that simultaneously maximize item quality and diversity among items. Specifically, given a set of items I, a DPP assigns a probability $P(S)$ to every subset $S \in \mathbb{P}(I)$, where $\mathcal{P}(I)$ is the *powerset* (i.e., the set of all subsets) of I. The goal is then to model (i.e., parameterize) this probability, either globally or for individual users, such that finding the subset of items which maximizes $P(S)$ has an optimal tradeoff between utility and diversity.

Wilhelm et al. (2018) studied the application of DPPs to diversify recommendations on *YouTube*.

The method assumes a few inputs. First, as with previous diversification techniques, we assume a utility or 'quality' estimate $f(u, i)$ is given (e.g. from a pre-trained recommender system), which encodes the probabilty that user u will interact with item i given i's features; we also assume a predefined distance function $d(i, j)$ between two items.

Next, we have historical sets of items that have been surfaced to the user (i.e., the outputs of an existing system), along with subsets of items that the user selected (i.e., binary labels indicated by $y_{u,i}$). The goal is to select subsets of items that will maximize the total number of interactions, which in practice is trained by maximizing the Cumulative Cain:

$$\sum_u \sum_i \frac{y_{u,i}}{rank(u, i)}, \tag{9.6}$$

where $rank(u, i)$ is the new rank assigned by the proposed algorithm. That is, the items that the user interacted with ($y_{u,i} = 1$) should have high rank.

Note that the above seems reminiscent of 'traditional' approaches to recommendation, i.e., we are ranking items such that positive interactions should have high rank (which seems similar to what we saw in **??**). The main difference here is simply the observation that the total number of interactions will be maximized when utility and diversity are balanced (e.g. a user will quickly become bored if recommendations cover only one of their interests).

Next, given a candidate set of N items, we next define a matrix $L^{(u)}$ such that diaginal entries $L^{(u)}_{i,i}$ encode the utility of an item i and off-diagonal entries $L^{(u)}_{i,j}$ encode the similarity of two items i and j. The specific parameterization used

in Wilhelm et al. (2018) is:

$$L^{(u)}_{i,i} = f(u,i)^2 \tag{9.7}$$

$$L^{(u)}_{i,j} = \alpha f(u,i) f(u_j) \exp(-\frac{d(i,j)}{2\sigma^2}) \quad \text{for } i \neq j. \tag{9.8}$$

Now, the quality of a subset S is proportional to the determinant of the submatrix of L induced by S, $\det(L_S)$. Specifically:

$$P(S) = \frac{\det(L_S)}{\sum_{S' \in \mathbb{P}(I)} \det(L_{S'})}. \tag{9.9}$$

Critically, the denominator of the above equation can be computed efficiently as

$$\sum_{S' \in \mathbb{P}(I)} \det(L_{S'}) = \det(L + I), \tag{9.10}$$

where I is the identity matrix.

To understand (roughly) why the determinant is diversifying, it helps to consider the trivial example where S consists of only two items i and j; then the determinant is given by

$$\det\left(\begin{bmatrix} L_{i,i} & L_{i,j} \\ L_{j,i} & L_{j,j} \end{bmatrix}\right) = L_{i,i}L_{j,j} - L_{i,j}L_{j,i}; \tag{9.11}$$

this value will be maximized when the utility is high ($L_{i,i}L_{j,j}$) and the similarity is low ($L_{i,j}L_{j,i}$).

In spite of the relatively simple form of Equation (9.9), it is still not practical to solve the (NP-hard) problem of finding the optimal subset. In Wilhelm et al. (2018) this is addressed by a simple greedy algorithm (similar to that of **??**, among others), in which one starts with the empty set of videos $S = \emptyset$, and iteratively adds the item i which maximizes the determinant $\det(L_{S \cup \{i\}})$

Note that the parameterization in Section 9.4.2 includes two parameters, α and σ, which in practice are tuned using grid search. Intuitively these parameters control the relative weight of utility versus diversity (α), and the 'tightness' of the similarity function (σ). These parameters are global, but in principle could be learned per-user.

Ultimately, the experiments find that implementing the above DPP on a user's video feed increases user satisfaction (as measured by session duration), compared to various other diversification strategies.

9.4.3 Calibration

A related notion to diversity is that of *calibration* of predictions or recommentations. Whereas a diversity metric might suggest (for example) that we

should expose users to a wide distribution of recommendations, which potentially span beyond their explicit preferences, *calibration* refers to the idea that recommendations should be made *in proportion* to expressed preferences. For instance, if a user watches 40% sci-fi movies and 60% romantic comedies, they should not exclusively be recommended romantic comedies, as might happen when naïvely recommending nearest neighbors or maximizing inner products.

Steck (2018) introduces such a notion of *calibrated recommendations*, in the context of movie recommendations on *Netflix*. Their work discusses both metrics to assess calibration, as well as methods to calibrate the outputs of an existing recommender system.

Their notion of calibration operates over a pre-defined set of item *genres*, described using a stochastic genre vector $p(g|i)$ (e.g. a movie might be categorized as 80% 'action' and 20% 'sci-fi'); this could potentially be adapted to other attributes toward which one desired calibration. The basic idea behind a calibration metric is then that the distribution of genres among a user's history $\imath \in \mathcal{H}$ should match the distribution of *recommended* items $i \in \mathcal{I}$. The two terms are defined (respectively) as:

$$p(g|u) = \frac{\sum_{i \in \mathcal{H}} w_{u,i} \cdot p(g|i)}{\sum_{i \in \mathcal{H}} w_{u,i}} \tag{9.12}$$

$$q(g|u) = \frac{\sum_{i \in \mathcal{I}} w_{r(i)} \cdot p(g|i)}{\sum_{i \in \mathcal{H}} w_{r(i)}}. \tag{9.13}$$

Both expressions include a 'weighting' term w. In the case of the historical distribution $w_{u,i}$ might weight items according to recency (for example), or for the case of recommendations $w_{r(i)}$ might weight recommendations according to their position in a list (i.e., their ranking); either term could also be ignored.

Now, the goal is to generate a set of recommended items $\mathcal{I}$ such that the two distributions should match closely. The difference between the two distributions can be measured by (e.g.) the Kullback Liebler divergence:

$$\mathrm{KL}(p, q) = \sum_{g} p(g|u) \log \frac{p(g|u)}{q(g|u)}. \tag{9.14}$$

Of course, the recommendations should also be highly compatible according to the recommender system itself. This is achieved in Steck (2018) with a simple expression that trades off recommendation utility and calibration (via a trade-off hyperparameter λ):

$$\mathcal{I} = \arg\max_{\mathcal{I}} \underbrace{(1-\lambda) \cdot \sum_{i \in I} s(i)}_{\text{compatibility}} - \underbrace{\lambda \cdot \mathrm{KL}(p, q(\mathcal{I}))}_{\text{calibration}} \tag{9.15}$$

Steck (2018) notes that the above is a hard combinatorial optimization problem, but can be approximated greedily (with a certain optimality guarantee), by

Table 9.1 *Summary of diversification techniques.*

Technique	Reference	Description
Maximum Marginal Relevance	Carbonell and Goldstein (1998), Section 9.3.1	Recommended items should balance utility against diversity compared to already-recommended items
Aggregate Diversity	Adomavicius and Kwon (2011)	Recommended items should be those that have high compatibility for a particular user, but low aggregate compatibility (e.g. popularity); this will lead to *aggregate* diversity of recommendations across the entire population
Determinantal Point Processes	Wilhelm et al. (2018) **??**	Balance utility and diversity (as with MMR above), but using a set-based objective
Calibration	Steck (2018)	Recommendations should exhibit the same distribution of attributes (e.g. in terms of recommended categories) as users' historical interactions

iteratively adding one item at a time to $\mathcal{I}$ so as to optimize the above criterion until the desired number of items is reached.

An appealing property of this approach is that it can be applied in a purely post-hoc fashion to the outputs of *any* recommender system that associates scores between users and items. The experiments in Steck (2018) show (by varying λ in Equation (9.15)) that a reasonable degree of calibration can be achieved with minimal loss in recommendation utility.

9.4.4 Implementing a Diverse Recommender

Here we'll briefly describe an implementation of a diversified recommender, based on the *maximum marginal relevance* method of **??** (though other reranking-based methods are similar).

We start by building a few utility data structures. First we collect the list of candidate recommendations, excluding any items the user has already consumed. Next we compute compatibility scores between a user u and all items.

Here our compatibility scores (i.e., $Sim^{\text{user}}(i,u)$ in Equation (9.2)) are simply the output of a latent-factor recommender (here we use a batch-based prediction function as in **??**). We sort these in order from the highest- to the lowest-rated items. In practice we might want to re-rank only the top few hundred items rather than computing diversity scores for extremely low-compatibility items.

```
candidates = list(itemSet.difference(itemsPerUser[u]))
compatScores = list(zip([float(f)
    for f in model.predictSample([userIDs[u]]*len(candidates),
        [itemIDs[i] for i in candidates])], candidates))

compatScores.sort(reverse=True)
```

Next we implement a function to determine the similarity between a candidate recommendation and others already in the list (i.e.,$\max_{j \in S} Sim^{\text{item}}(i,j)$ from Equation (9.2)). `itemEmbeddings` is a lookup table containing embeddings γ_i for each item. The similarity function (`sim`) is the cosine similarity (not shown), though other similarity functions could be substituted (including simple alternatives such as checking whether items belong to the same category):

```
itemEmbeddings = dict(zip(candidates,
    tf.nn.embedding_lookup(model.gammaI, [itemIDs[i] for i in
        candidates])))

def maxSim(itemEmbeddings,i,seq):
    if len(seq) == 0: return 0
    return max([sim(itemEmbeddings,i,j) for j in seq])
```

To implement the iterative reranker we define a method that takes the list of recommendations generated so far (`seq`), and generates the next item to be added to the list, based on the weighted combination from **??**. λ is passed as an argument to the function to trade-off the importance of compatibility and diversity:

```
def getNextRec(model,compatScores,itemEmbeddings,seq,lamb):
    scores = [(lamb * s - (1 - lamb) * maxSim(itemEmbeddings,i,seq
        ), i)
        for (s,i) in compatScores if not i in seq]
    (maxScore,maxItem) = max(scores)
    print(maxScore)
    return maxItem
```

Note that the above implementation is inefficient and even on a modestly sized dataset (in terms of the size of the item vocabulary) requires several seconds to generate recommendations. Several strategies might be used to improve its performance, including efficient retrieval techniques (as in **??**), or

Table 9.2 *Diversified recommendations (maximum marginal relevance).*

rank	Low diversity	Medium diversity	High diversity
1	Founders KBS (Kentucky Breakfast Stout)	Founders KBS (Kentucky Breakfast Stout)	Founders KBS (Kentucky Breakfast Stout)
2	Two Hearted Ale	Samuel Smith's Nut Brown Ale	Samuel Smith's Nut Brown Ale
3	Bell's Hopslam Ale	Two Hearted Ale	Salvator Doppel Bock
4	Pliny The Elder	Bell's Hopslam Ale	Oil Of Aphrodite - Rum Barrel Aged
5	Samuel Smith's Oatmeal Stout	Kolsch	Great Lakes Grassroots Ale
6	Blind Pig IPA	Drax Beer	Blue Dot Double India Pale Ale
7	Stone Ruination IPA	A Little Sumpin' Extra! Ale	Calistoga Wheat
8	Schneider Aventinus	Odell Cutthroat Porter	Dogwood Decadent Ale
9	The Abyss	Miner's Daughter Oatmeal Stout	Traquair Jacobite
10	Northern Hemisphere Harvest Wet Hop Ale	Rare Bourbon County Stout	Cantillon Gueuze 100% Lambic

by exploiting certain structure in our compatibility or diversity functions that would obviate the need to exhaustively compute all scores.

Examples of Diversified Recommendations Table 9.2 shows examples of diversified recommendations on data from *BeerAdvocate*. Different values of λ are chosen to control the compatibility/diversity trade-off (for a randomly chosen user). The first set of recommendations ($\lambda = 1$) optimizes only for compatibility: the user is recommended a selection of rich stouts and IPAs. Decreasing λ by a little (middle column) introduces a few 'lighter' yet similar beers; decreasing λ further results in beers from a wide variety of categories (wheat beers, lambics, scotch ales, etc.).

Note that the ideal value of λ depends on a variety of factors, including our specific choices of compatibility and diversity functions (which may not even be on the same scale: in our case one is a rating from (in the range $[1, 5]$) and the other is a cosine similarity (in the range $[-1, 1]$). The solution can also be sensitive to hyperparameters (e.g. the number of factors and how strongly we regularize). In practice the optimal amount of diversity may simply be guided by what 'looks right.'

9.5 Case Studies on Recommendation and Consumption Diversity

In **??**, we saw an empirical study of diversity on *YouTube*, which argued that recommender systems led to diverse views, though this analysis was limited in that the point of comparison was a popularity-based alternative (which is almost certainly not diversity-inducing). Below we explore a few additional case-studies that study diversity within the context of music (Section 9.5.1) and news (**??**) recommendation, attempting to characterize users in terms of their consumption patterns, and potentially how to guide users to more diverse content.

9.5.1 Diversity on Spotify

Anderson et al. (2020) tries to study the effect that recommendation algorithms have on diversity, and more critically tries to understand how different types of users respond to diverse recommendations.

The paper considers the listening patterns of around 100 million users on *Spotify*. Unlike (e.g.) Fleder and Hosanagar (2009), which defined 'diversity' in terms of the Gini coefficient (i.e., the statistical dispersion of *which* items get consumed), Anderson et al. (2020) defines diversity in terms of song representations, i.e., essentially the γ_i values in a recommender system.[1]

The specific embeddings γ_i on *Spotify* are estimated using an *item2vec*-like method (as we saw in Section 7.2.1). The *musical diversity* of a user u's listening activity is defined in terms of a score which they term *generalist-specialist* (or *GS*), following previous work Waller and Anderson (2019). Specifically, we start by defining the centroid of a user's listening history (which we'll term γ_u) as

$$\gamma_u = \frac{1}{|H|} \sum_{j=1}^{|H|} \gamma_{H_j}, \tag{9.16}$$

where H is a list of songs in the user's *listening history*, with repetition, such that repeated listens will count more to the average. Then the *GS*-score is defined as the average cosine similarity between the user representation γ_u and the items they listen to:

$$GS(u) = \frac{1}{|H|} \sum_{j=1}^{|H|} \frac{\gamma_{H_j} \cdot \gamma_u}{||\gamma_{H_j}||||\gamma_u||}. \tag{9.17}$$

[1] Of course, this version of 'diversity' has its own limitations, as it assumes that the learned latent space accurately captures the diversity among items.

Intuitively, *specialists* (high $GS(u)$) tend to have songs γ_i in their listening history oriented primarily in a certain direction; *generalists* (low $GS(u)$) do not, ostensibly corresponding to a broader range of prefrences.

Part of the analysis in Anderson et al. (2020) is a study of the relationship between diversity (as measured by $GS(u)$) and various other attributes. For example, less active users tend to be specialists (high $GS(u)$), generalist users are less likely to abandon the system ('churn'), and more likely to subscribe to subscribe to the 'premium' version of the product.

However the main feature in the analysis is to study the relationship between recommendations and diversity, and in particular how generalists and specialists respond differently to algorithmic recommendations. This is measured experimentally by exposing real users of *Spotify* to different recommendation conditions. Three types of recommender system are used: one which merely ranks songs (within a specific predefined subgenre) by popularity; one which is a simple relevance ranker based on user-to-item similarity (essentially a form of heuristic recommendation); and one which is a learned recommender specifically trained to maximize the probability that a user will listen to a song to completion.

First, compared to popularity, recommendation approaches lead to a substantial increase in the number of songs streamed for both groups (they also lead to an increase in the number of songs *skipped*, but this is more than made up for in additional streams). That is, users could be said to be more *engaged* when interacting with recommendations compared to a popularity baseline. Second, the benefit of recommendations appears across both groups (generalists and specialists), though is significantly more pronounced for specialists: this aligns with the paper's hypothesis in the sense that specialists are more sensitive to songs matching their personal relevance criteria. Finally, the learned ranker confers a slight additional benefit over the relevance ranker, though the benefit is surprisingly modest, indicating that simple relevance ranking is sufficient in this context.

Guiding Users to More Diverse Content A follow-up paper, Hansen et al. (2021), also considers consumption patterns on *Spotify*, and broadly explores the trade-offs involved in terms of algorithmic choices, diversity methods, and user satisfaction. They note (as we've seen throughout this chapter) that several ranking approaches bias recommendations toward highly-popular content that closely resembles interactions from users' histories; like Anderson et al. (2020) they also find evidence that users can in many cases be satisfied by recommendations that are more diverse and less popular.

Several diversification techniques are explored, each of which essentially

attempts to trade-off a relevance versus a diversity term. Hansen et al. (2021) explores the merits of each; they broadly favor reinforcement learning-based approaches as a means of swaying users toward diverse content, but note the difficulties involved in productionalizing such systems.

9.5.2 Filter Bubbles and Online News Consumption

Much of the discussion of 'filter bubbles' has been in the context of online news, where concerns generally center around whether recommender systems (or more simply, algorithmic ranking techniques) will limit the ideological diversity of content users consume.

Bakshy et al. (2015) studied the extent to which users on *Facebook* tend to consume news that confirms to their political ideology. The analysis begins by training a supervised learning system to label news articles as 'liberal,' 'conservative,' or 'neutral,' based on shares by users who volunteer their political affiliation as part of their profile.

The main questions of interest center around the extent to which users are exposed to (or choose to interact with) content that is aligned with their own ideology versus content which is 'cross-cutting.' 'Exposure' refers to algorithmic feed ranking choosing to surface the content, whereas 'interaction' refers to a user's choice to click on exposed content.

There are many confounding factors in such an analysis, which the study attempts to control for. For example, users' social networks are primarily composed of friends who share a common ideology, so naturally the content users could potentially be exposed to via their social network is predominantly not cross-cutting. Likewise, users' tendency to interact with (i.e., click on) content is confounded by the fact that the feed ranker already factors in click probability when determining which (and how prominently) content is surfaced to the user in the first place.

After attempting to control for these effects, the study's main findings are that algorithmic ranking indeed exposes users to less ideologically diverse news than would be expected by the ideological makeup of their social group, however users interact with ideologically diverse content even less then they are exposed to it; from this, the authors conclude that individual choice plays the largest role in users' exposure to content that is ideologically homogeneous.

However, the argument above does not refute the possibility of a 'filter bubble' of online news consumption, it merely argues that its primary cause (in the case of *Facebook*) is not necessarily algorithmic.

Flaxman et al. (2016)

Filter Bubbles on Google News Haim et al. (2018) conducted an exploratory study of recommendation on *Google News*, in order to determine the effects of personalization on content diversity. Like Bakshy et al. (2015), they broadly argue that the effects of filter bubbles are somewhat overstated, or otherwise that the patterns of bias in recommendations are not the same as what is anecdotally understood to be a 'filter bubble.'

They conduct two studies, to look at 'explicit' and 'implicit' personalization. Both are based on empirical observation of the actual news recommendations provided by *Google News*, sampled from several synthetic user accounts. Recommendations are then compared against 'traditional' (i.e., non-personalized, curated) news sources in terms of topic and content diversity.

In the 'explicit' setting, they make use of a *Google News* feature that allows users to specify the types of news they are interested in, among a set of broad categories (e.g. sports, entertainment, politics). Annotators then labeled recommended articles according to these categories in order to quantify the alignment between the explicit preferences and the recommended articles.

The first finding is simply that *Google News* does indeed respect users' explicit preferences, in the sense that the proportion of recommended articles matching the desired topic far exceeds their proportion in a non-personalized setting.

Haim et al. (2018) also evaluates recommendations in terms of source diversity (i.e., in terms of the original news sources that *Google News* aggregates). Here, they find surprisingly that a few somewhat niche news sources dominate recommendations, whereas more mainstream sources are underrepresented; this result is relatively consistent across each of the personalized accounts.

In the 'implicit' setting, Haim et al. (2018) made use of several social media accounts, corresponding to users with specified (but synthetic) demographics and preferences (such as a marketing manager, an elderly conservative, etc.). Each of these simulated agents then interacts with social media (liking articles on *Facebook*, *Google+*, etc.), after which their *Google News* recommendations are compared.

The main conclusion of this second study is simply that implicit personalization has little effect on the recommended results (though there is evidence that some results are indeed personalized).

Ultimately while both Bakshy et al. (2015) and Haim et al. (2018) argue against a 'filter bubble' as such, both point to potential issues of bias in recommendations; Bakshy et al. (2015) suggest that recommendations do indeed present an overall more biased perspective compared to users' broader social

groups, while Haim et al. (2018) shows that certain niche sources tend to be over-represented in news recommendations.

Breaking the filter bubble: democracy and design (maybe)

9.6 Fairness

Fairness in machine learning is often defined in terms of predictions and protected characteristics. For example, when building a classifier to aid in hiring decisions, we might be interested in ensuring that men and women are ranked as 'qualified' at approximately the same rate. Or, a system for predicting recidivism should not be biased against individuals of a certain race Chouldechova (2017).

Typically, we might desire that the outputs of our classifier $f(x_i)$ do not depend on some protected feature $x_{i,f}$ (indicating race, gender, etc.). Some common definitions include, for example, *demographic parity*, which states that the probability of a positive prediction (e.g. being ranked as 'qualified') should be the same whether one has the protected feature or not:

$$P(f(x_i) = 1|x_{i,f} = 1) = P(f(x_i) = 1|x_{i,f} = 0). \tag{9.18}$$

The related notion of *equal opportunity* allows for the possiiblity that the target variable depends on the protected feature, and states that *among the qualified* ($y_i = 1$) *or unqualified* ($y_i = 0$) *individuals*, the probability of a positive prediction should be the same whether or not one has the protected features:

$$P(f(x_i) = 1|x_{i,f} = 1, y_i = y) = P(f(x_i) = 1|x_{i,f} = 0, y_i = y). \tag{9.19}$$

These are just two examples out of dozens of possible notions of fairness that may be of interest, see e.g. Mehrabi et al. (2019) for a comprehensive survey.

A classifier may violate the above rules for a variety of reasons. For example, the training data may exhibit historical bias against a certain group; or, a classifier trained on highly imbalanced data may simply make inaccurate (or imbalanced) predictions for groups that are poorly represented in the data. Several machine learning techniques have been proposed to mitigate unfairness in such scenarios, for example by pre-processing the biased data Kamiran and Calders (2009), or by altering the classifier itself Zafar et al. (2017).

In the contexts of recommendations and personalized predictions, one may have slightly different definitions and goals in terms of building a 'fair' model. Yao and Huang (2017) attempts to adapt notions of fairness to personalized recommendation contexts. The paper considers a running example of recommendation in the context of education, where course evaluations in Computer

Science (for example) may primarily represent the preferences of the predominantly male population; models trained on such data (or even simple statistics compiled from the data).

Yao and Huang (2017) introduce several metrics of fairness with respect to the outputs of a recommender system, and show that these metrics can be straightforwardly incorporated into the training objective. Their metrics are defined by dividing users into groups g_u and items into 'genres' or attributes h_i; user groups are assumed to be binary, though in the studied case of gender g_u can simply be divided into the over-represented group (male) and the under-represented group (non-male).

Value Unfairness measures the extent to which one group tends to have their ratings over- or under-predicted compared to the other:

$$U_{\text{val}} = \frac{1}{|I|}\sum_{i=1}^{|I|}\left|\left(\underbrace{\mathbb{E}_g[y]_i}_{\substack{\text{expected } \textit{prediction} \\ \text{for group } g \text{ on item } i}} - \overbrace{\mathbb{E}_g[r]_i}^{\substack{\text{average } \textit{rating} \\ \text{for group } g \text{ on item } i}}\right) - \left(\mathbb{E}_{\neg g}[y]_i - \mathbb{E}_{\neg g}[r]_i\right)\right|. \tag{9.20}$$

Note that since both sides take expectations (or averages), the measure is invariant to size differences between the two groups.

Value unfairness could occur in a latent factor model (like that of **??**) if, for example, predicitons are dominated via the bias terms β_i; in a model in which one group was over-represented, the bias terms may essentially reflect the preferences only of the over-represented group.

Absolute Unfairness replaces the differences in expectation from Equation (9.20) with absolute values:

$$U_{\text{abs}} = \frac{1}{|I|}\sum_{i=1}^{|I|}\left|\left|\mathbb{E}_g[y]_i - \mathbb{E}_g[r]_i\right| - \left|\mathbb{E}_{\neg g}[y]_i - \mathbb{E}_{\neg g}[r]_i\right|\right|. \tag{9.21}$$

Following this change, absolute unfairness now captures the extent to which one group has their ratings *mispredicted* in an absolute sense more than the other. This essentially measures a difference in the system's *utility* between the two groups, in the sense that if one group routinely receives recommendations with high error then the system is unlikely to be useful to them.

Next, Yao and Huang (2017) define *under-* and *over-estimation* unfairness to assess the model's tendency of the model to either under or overpredict the

true ratings:

$$U_{\text{under}} = \frac{1}{|I|}\sum_{i=1}^{|I|}\Big|\max\{0, \mathbb{E}_g[r]_i - \mathbb{E}_g[y]_i\} - \max\{0, \mathbb{E}_{\neg g}[r]_i - \mathbb{E}_{\neg g}[y]_i\}\Big| \quad (9.22)$$

$$U_{\text{over}} = \frac{1}{|I|}\sum_{i=1}^{|I|}\Big|\max\{0, \mathbb{E}_g[y]_i - \mathbb{E}_g[r]_i\} - \max\{0, \mathbb{E}_{\neg g}[y]_i - \mathbb{E}_{\neg g}[r]_i\}\Big| \quad (9.23)$$

These definitions are somewhat analogous to related concepts we saw when evaluating ranking models in Section 3.9; consistently underpredicting is analogous to having low recall (failing to retrieve relevant items), whereas overpredicting is analogous to having low precision (retrieving items that are not relevant). Both can potentially reduce the utility of the recommender system for one of the groups.

Ultimately, Yao and Huang (2017) shows that each of the above metrics can be incorporated into recommender systems of the form given in Section 4.5.3. That is, they can be combined via a trade-off term so that the model is accurate while minimizing unfairness, e.g.:

$$\frac{1}{\mathcal{T}}\sum_{(u,i)\in\mathcal{T}} \underbrace{(\alpha + \beta_i + \beta_u + \gamma_i + \gamma_u - R_{u,i})^2}_{\text{accuracy}} + \lambda \underbrace{U_{\text{abs}}}_{\text{(absolute) fairness}}. \quad (9.24)$$

Optimization remains straightforward, as each fairness metric is differentiable with respect to the model parameters.[2] The main finding of the paper is then that fairness metrics can optimized while paying only a minimal price in terms of overall model accuracy.

9.6.1 Multisided Fairness

A separate attempt to introduce fairness metrics is described in Burke (2017). In comparison to the fairness metrics defined above, the main difference is to consider fairness from both the perspective of users of the system ('consumers') as well as content providers ('producers'). As a motivating example, they consider a hypothetical recommendation scenario on the microfinancing website *Kiva.org*, where there may be an desire to ensure that proposals from different businesses receive somewhat balanced representation among recommendations. More broadly, this is an instance of recommendation in a 'matchmaking' setting where both sides (in this case, users and businesses) are being matched to each other; in such cases, fairness should not be defined in terms of one 'side' only, but should consider the needs of both types of stakeholders.

[2] Or at least, has a subderivative.

Other examples are cited including online advertising, the sharing economy, or online dating Pizzato et al. (2010).

To achieve this notion of fairness, they consider fairness separately from the perspective of 'consumers' and 'producers,' which they term *C*- and *P*-fairness. Following these definitions, the fairness metrics we studied above would be examples of *C*-fairness. Burke (2017) notes that *C*- and *P*-fairness are not merely symmetric definitions, and that *P*-fairness may have requirements not encountered when studying *C*-fairness, such as in the examples above. For example, in a product recommendation setting, if we wanted to encourage sales diversity (as we considered in **??**, and will revisit in **??**), the producers are *passive* in the sense that they are not actively seeking out recommendations in the system.

Finally, Burke (2017) considers settings of *CP*-fairness, where fairness must be considered from the perspectives of both sides simultaneously. We'll revisit examples of *P*-fairness and *CP*-fairness as we examine case studies of gender bias in Section 9.8.

Paper: Balanced Neighborhoods for Multi-sided Fairness in Recommendation (maybe)

9.6.2 Group-Aware Recommendation

Group-aware Recommendation? Paper: Top-N Group Recommendations with Fairness

Fairness-Aware Group Recommendation with Pareto-Efficiency

9.7 Implementing Fairness Objectives in *Tensorflow*

As we discussed, the appeal of the fairness objectives such as those we developed in **??** is that they can straightforwardly be incorporated into the learning objectives of standard recommenders.

Below we'll implement 'absolute fairness' as in **??**. We'll use data from beer reviews (which we'll explore further in Project 9.10, which includes user gender information, and in which male users are substantially over-represented. First, we read in the data, recording user gender along with each interaction:

```
for d in parse("beer.json.gz"):
    g = 'user/gender' in d and d['user/gender'] == 'Male'
    u = d['user/profileName']
    i = d['beer/beerId']
    r = d['review/overall']
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)
    interactions.append((g,u,i,r))
```

Next we build some utility data structures to store interactions for each item according to group membership (G and $\neg G$ for males and females):

```
interactionsPerItemG = defaultdict(list)
interactionsPerItemGneg = defaultdict(list)

for g,u,i,r in interactions:
    if g: interactionsPerItemG[i].append((u,r))
    else: interactionsPerItemGneg[i].append((u,r))
```

We also store item sets for each group for sampling:

```
itemsG = set(interactionsPerItemG.keys())
itemsGneg = set(interactionsPerItemGneg.keys())
itemsBoth = itemsG.intersection(itemsGneg)
```

Finally, we implement the absolute fairness objective. This implementation computes the fairness objective for a single item (i.e., one term in the summation in **??**). During training, this objective can be called for a small sample of items (just as we compute the accuracy over a small sample as in **??**), and added to the accuracy term:

```
def absoluteUnfairness(self, i):
    G = interactionsPerItemG[i]
    Gneg = interactionsPerItemGneg[i]
    # interactions take the form (u,r)
    rG = tf.reduce_mean(tf.convert_to_tensor([r for _,r in G]))
    rGneg = tf.reduce_mean(tf.convert_to_tensor([r for _,r in Gneg
        ]))
    pG = tf.reduce_mean(self.predictSample([userIDs[u] for u,_ in
        G], [itemIDs[i]]*len(G)))
    pGneg = tf.reduce_mean(self.predictSample([userIDs[u] for u,_
        in Gneg], [itemIDs[i]]*len(Gneg)))
    Uabs = tf.abs(tf.abs(pG - rG) - tf.abs(pGneg - rGneg))
    return self.lambFair * Uabs
```

9.8 Case Studies on Gender Bias in Recommendation

Just as Yao and Huang (2017) used gender imbalance as a motivating example to study fairness and bias with regard to under-represented groups in recommender systems, several studies have investigated specific scenarios where recommenders exhibit significant bias, or have reduced utility for a specific gender.

9.8.1 Data Resampling and Popularity Bias

Ekstrand et al. (2018a) studies a similar problem as Yao and Huang (2017) (**??**), also noting that there is a substantial *utility gap* between the majority versus under-represented groups (i.e., closest to *absolute unfairness* as in Equation (9.21)). Bias is reported with respect to both gender and age attributes, both of which are self-reported by uses in datasets of movies and songs (from *MovieLens* Harper and Konstan (2015) and *Last.FM* **?**).

Unlike Yao and Huang (2017), which corrects for this type of bias using a joint objective that balances overall utility with unfairness (Equation (9.24)), Ekstrand et al. (2018a) uses a *data resampling* approach to correct for bias. This type of approach is borrowed from Kamiran and Calders (2009) where it was used in the context of fair classification. The basic idea is to resample the data so as to achieve equal representation among groups; practically speaking this is fairly similar to the reweighting schemes we explored in **??**.

Ekstrand et al. (2018a) also raises the potential issue of *popularity bias* in recommender systems (also discussed in Bellogin et al. (2011)), in which algorithms that work well for popular items will generally be favored over algorithms which *personalize* better, but whose performance is worse for popular items. To address this they introduce evaluation metrics that control for the effect of popularity, so that algorithms can be compared according to their degree of personalization rather than their tendency to select popular items.

9.8.2 Bias and Author Gender in Book Recommendations

Ekstrand et al. (2018b) explores bias from the perspective of book *authors*. This is somewhat analogous to the idea of P-fairness from Section 9.6.1, given that we are interested in how recommendations could be biased against 'producers' (in this case, authors of a certain gender).

Data is collected from *BookCrossing* Ziegler et al. (2005), *Amazon* McAuley et al. (2015a), and *GoodReads* Wan and McAuley (2018). An interesting component of the study is how these datasets can be augmented to incorporate

the gender of each author, which is not a feature immediately available in any of the above datasets; author gender information is compiled from external sources, which is matched to records in each of the datasets.

The main research questions in Ekstrand et al. (2018b) start by analyzing the overall gender distribution of authors in the datasets, as compared to the gender distribution among reading histories of individual users. Beyond this, they seek to study how gender bias is 'propagated' by recommendation algorithms, i.e., the extent to which users who exhibit a moderate tendency toward authors of a certain gender will tend to have recommendations in which that gender is more extremely over-represented. Finally, they analyze the extent to which these issues can be mitigated algorithmically.

Ultimately, the study concludes that authors in all three datasets (at least those whose identities could be resolved) are predominently male. In terms of rating histories by users, the distribution is less skewed. In terms of recommendation algorithms, results are quite mixed, with certain algorithms and datasets leading to more or less skewed recommendations, or otherwise recommendations that mimic users' own gender preferences.

Finally, the authors find that gender imbalance in recommendations can be mitigated easily via simple re-ranking strategies, with minimal impact on performance (the strategy they use is similar to the one of Ziegler et al. (2005), as we studied in **??**). This analysis bears some similarity to our study of filter bubbles (**??**), or the techniques used to calibrate recommendations (in this case to match a desired gender distribution rather than a genre distribution) from **??**.

9.8.3 Gender Bias in Marketing

Wan et al. (2020) investigates bias in terms of how products are *marketed*. For example, a user may be more (or less) inclined to purchase a clothing item if it is modeled by somebody sharing their gender, weight, age, skin-tone, etc. In some instances, these features may be directly relevant to the suitability of the item, but in others they may not be. If users are disinclined to interact with items simply because their own identity is not represented, this reduces the utility of the system to the users, represents a missed opportunity in terms of sales, and raises broader issues of representation in marketing. 'Fairness' from this perspective is an instance of *CP*-fairness from Section 9.6.1, as both producers and consumers face consequences from unfair treatment.

Like the above study, Wan et al. (2020) starts by assessing the extent to which these types of bias can be found in historical purchase data. They consider two settings: clothing, using a dataset from *ModCloth*, and electronics,

using data from *Amazon*. On *ModCloth*, they are interested in whether users are less inclined to buy items if the model has a different body type than the user (e.g. the user is plus-size but the model is not, even though the clothing is available in plus-sizes). On *Amazon*, they are interested in whether ostensibly 'genderless' products have different sales patterns among male and female users.

Again, the study faces difficult issues of augmenting the data, since gender and size attributes of users are not readily available. *ModCloth* specifies the size of the models in marketing images, and user sizes are inferred from their historical tendency to purchase only items of a certain size. Data augmentation is more difficult on *Amazon*: gender attributes in marketing images must be inferred using computer vision techniques; gender attributes of users are inferred from their purchases in the *clothing* category.[3]

Indeed, the study determines that there is significant correlation between users' attributes and their purchase patterns (e.g. male users tend to purchase electronics items marketed by male models). Of course, as with gender in book recommendations above (Section 9.8.2), it is hard to disentangle 'bias' or 'unfairness' from users' intrinsic preferences or legitimate marketing choices (e.g. the reason that women tend to buy women's watches may be largely practical). However the goal is to determine whether bias is *amplified* by recommender systems, and whether this effect can be mitigated.

The specific question that is asked is whether recommendation *errors* are correlated with market segments and marketing images. This bears passing similarity to the notion of *absolute unfairness* as in Equation (9.21), though that measure considers unfairness only from the perspective of the *user's* identity, whereas the question asked here concerns both user and item 'identity' simultaneously. Specifically, four possible types of error are investigated:

$$\text{Product Image} \begin{cases} \text{Female} \\ \text{Male} \end{cases} \underbrace{\begin{bmatrix} \bar{e}_{F,F} & \bar{e}_{M,F} \\ \bar{e}_{F,M} & \bar{e}_{M,M} \end{bmatrix}}_{\substack{\text{Female} \quad \text{Male} \\ \text{User Identity}}} . \tag{9.25}$$

Under a null model, the errors should not be correlated with market segments (this can be measured via a specific statistical test).

Finding that these errors are indeed significantly correlated with market segments, Wan et al. (2020) seeks to address this via a loss which balances model

[3] Of course, clothing purchases are a rough proxy for gender identity, and users whose purchases span both gender categories are not considered.

Table 9.3 *Comparison of personalized fairness objectives.*

Objective	Reference	Description
Value Unfairness	Yao and Huang (2017)	Neither of two groups should have their compatibility over- or under-predicted more than the other
Absolute Unfairness	Yao and Huang (2017)	Neither of two groups should have their compatibility mispredicted more than the other
Demographic parity among recommendations	Ekstrand et al. (2018b)	Demographics (e.g. author gender) should be reasonably balanced among the items being recommended
Marketing fairness	Wan et al. (2020)	Individuals underrepresented in marketing (e.g. images) should not have reduced recommendation utility

error and error correlation:

$$\overbrace{\sum_{u,i}(f(u,i)-r_{u,i})^2}^{\text{prediction error}}+\alpha \underbrace{\mathcal{L}_{corr}}_{\text{error parity on market segments}}. \tag{9.26}$$

Again, this joint loss can be optimized much like the one in Equation (9.24), satisfying the fairness objective with minimal loss in prediction accuracy.

9.9 Justification and Trust

Building User Trust in Recommendations via Fairness and Explanations

Designing and evaluating explanations for recommender systems

Amazon papers: ?

Netflix: Big & Personal: data and models behind Netflix recommendations (talks about explainability, but not very good)

9.10 Exercises

Exercises

9.1 In this exercise we'll explore extending a recommender system to balance relevance and diversity. Start by implementing a standard latent-factor recommender, as in Chapter 4 (or borrowing one from a previous exercise). A good dataset for this task might be one that includes category information (such as the beer dataset from Project **??**, below), or some other feature that allows you to assess recommendation diversity. In this exercise we'll implement the *Maximum Marginal Relevance* (MRR) criterion as in Section 9.3.1 (though using a DPP (**??**) or another diversity criterion would be similar). We'll implement and evaluate the model in the following way:

- *Relevance* can be defined in the usual way, i.e., the inner product of user and item terms $\gamma_u \cdot \gamma_i$. *Diversity* could be defined in various ways, e.g. in terms of the difference between item features $\|\gamma_i - \gamma_j\|$, or even in terms of some simple attribute (e.g. if you wanted to recommend beers with different alcohol levels, you could define diversity in terms of the ABV feature).
- Use the MMR technique to generate recommendation lists for a sample of users. The

9.2 In addition to issues of diversity as we saw in Exercise **??**, we also studied *concentration* effects in **??**, whereby a recommender system can skew the distribution of recommended items toward a smaller of items than those represented in the training data.

9.3 In **??** we developed various fairness objectives for personalized recommender systems; although we'll explore these objectives more in Project **??**, for the moment let's consider the notion of *demographic parity* as in **??**. In **?** the authors measured demographic parity with respect to gender (of book authors), though for the purpose of this exercise you could consider any attribute associated with the items.

9.4 Some other re-ranking task

9.10.1 Project 7: Diverse and Fair Beer Recommendations

In this project we'll consider how we can improve the outputs of the types of recommendation approach we originally developed in Chapter 4. We'll consider the same beer datasets we've been using in various examples throughout the book. We select this dataset for two reasons:

- It contains a gender attribute (like the datasets used in Section 9.8), and in particular is imbalanced with respect to this attribute (the vast majority of users are male); thus we might be concerned that recommendations will have reduced utility for the underrepresented group.
- It contains item metadata, such as beer styles, which can be used to measure recommendation diversity, calibration, etc.

In principle this project could be completed with any similar dataset that includes (a) an attribute of interest with respect to which we can measure bias, such gender, age, etc.; and (b) item metadata with respect to which we can measure diversity.

We'll use this dataset to analyze diversity and fairness from the following perspectives:

(i) Implement a recommender system to predict ratings in the dataset, e.g. a latent factor model as in **??**.

(ii) Using the above model, compute the four fairness metrics from **??** (i.e., value unfairness, absolute unfairness, under- and over-estimation unfairness), comparing male (g) to non-male ($\neg g$) users.

(iii) Next, we'll assess recommended items in terms of diversity. Diversity could be measured in several ways, for example you could measure diversity with respect to the distribution of recommend items, or with respect to some attribute (e.g. the style or brand). This could be a formal measure of dispersion such as the Gini coefficient (as in **??**), or a simple

10
Further Reading

10.1 Online Advertising and News Recommendation

10.1.1 What Makes Ad Recommendation Different?

Superficially, surfacing advertisements to users seems no different from any other form of personalized recommendation. That is, we can imagine learning user 'preferences' and advertisement 'properties' from (e.g.) clicked advertisements in much the same way that any other recommender system is trained.

While ad recommendation does indeed have many similarities to other forms of personalized recommendation, there are several properties that demand different solutions compared to what we have seen so far. In particular:

- Advertisers have budget constraints. In most recommendation settings we can tolerate considerable imbalance among the items that are recommended (e.g. a highly popular movie might be recommended to a substantial fraction of all users); this is impossible when recommending ads, given that each advertiser can afford to surface only a limited number of ads (and at the same time, we want to ensure that all advertisers have some ads recommended).
- Likewise, each user may only be shown a limited number of ads; while this seems a common enough feature in most recommendation scenarios, it is especially apparent when surfacing ads as users are unlikely to explicitly request additional ads.
- Ad recommendations need to be made immediately. Again, this feature is common enough in many recommender systems, but is especially challenging given the considerations above: we cannot find a globally optimal solution that maximizes utility while satisfying advertisers' budget constraints. Instead, we must develop schemes that make local decisions in a way that approximates the globally optimal solution.
- Ad recommendation is highly contextual. Whereas most recommender sys-

tems rely heavily on user interaction histories, these are presumably less reliable in ad recommendation scenarios (where users' interactions with ads are extremely sparse), if an interaction history is available at all. As such, one has to rely more heavily on user *context* (e.g. a user's query to a search engine).

- Even if a user is responsive to a certain type of ad, there is diminishing value in repeatedly showing similar ads. Instead, we must sometimes recommend ads with low expected utility in the hope of discovering new user interests (this is the basic principle behind an explore/exploit tradeoff).

These are the problems we will investigate in order to build systems for online advertising. In Section

Note that in this section we will mostly ignore the question of how 'compatibility' between a user and an ad is estimated—this could itself be the output of a recommender system, or could simply be a bid that an advertiser places on a user or query.

10.2 Matching Problems

First we will consider the problem of incorporating constraints into (ad) recommendation problems. We want to consider situations where each user can only be shown a fixed number of ads, and each ad can only be shown to a fixed number of users. This is already an oversimplification, as advertisers would usually have a *budget*, and may (for example) bid different amounts per user or query (though we consider these issues in **??**).

To begin with, we'll consider the case where each ad is shown to exactly one user, and each user is shown exactly one ad. That is, we would like to select a function $ad(u)$ that maps users to ads, such that

$$ad(u) = ad(v) \rightarrow u = v. \tag{10.1}$$

Alternately, we could write this as an adjacency matrix A such that $A_{u,a} = 1$ if $ad(u) = a$. Then our constraint could be written as

$$\underbrace{\forall a \sum_u A_{u,a} = 1}_{\text{each advertiser shows one ad}} \quad ; \quad \underbrace{\forall u \sum_a A_{u,a} = 1}_{\text{each user sees one ad}}. \tag{10.2}$$

Then, we would like to choose the mapping that maximizes the utility between users and the ads they are shown:

$$\max_A \sum_{u,a} A_{u,a} f(u,a) \tag{10.3}$$

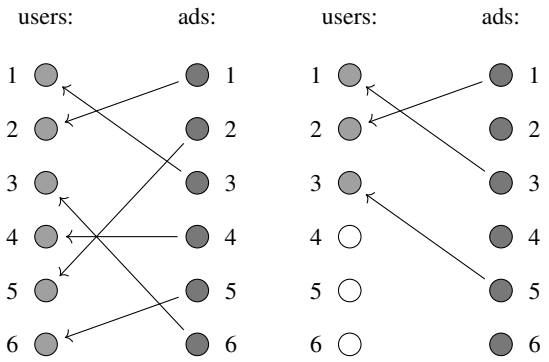


Figure 10.1 Ad recommendation can be viewed as a *bipartite matching problem* (left), where users are shown a fixed number of ads, and each ad is shown to a fixed number of users (in this figure each ad is shown to exactly one user, and vice versa). In an online setting (right), users arrive one at a time; we seek a solution that will be as close as possible to the solution we would have obtained in an offline setting at left.

where $f(u, a)$ is a measure of the compatibiliy between a user and an ad.

This type of problem is known as a *matching problem*. Conceptually, it can be viewed as matching two sets of nodes to form a bipartite graph (Figure 10.1, left), where each possible edge has an associated weight (i.e., the compatibility between a user and an ad). Our constraint above now says that every node should be incident on one edge.

Outside of ad recommendation, this type of matching problem appears in many settings, for example it is equivalent to the US's *National Residency Matching Program*, in which medical school students are matched to residency programs: each student can only be matched to a single residency, and each program has a limited number of slots; the matching should be chosen so as to optimize students' preferences for programs (and vice versa). Similar problems appear in various resource allocation settings Gusfield and Irving (1989); the original paper proposing the solution outlined below considered a setting related to college admissions Gale and Shapley (1962).

Although Equation (10.3) is a combinatorial optimization problem, it admits efficient approximation (and a polynomial exact solution). The approximation we discuss below is known as *stable marriage*, which draws an analogy between bipartite matching and finding (heterosexual) marriage partners: we seek a solution which, while not necessarily optimal, is 'stable,' in the sense that no unmatched pair would have an incentive to 'cheat.'

For each man/woman pair m and w, we assume a compatibility $f(m, w)$.[1] 'Stability' then means that there should be no pairs (m, w) and (m, w') such that m and w' are matched but $f(m, w) > f(m, w')$.

and initially assume that there are an equal number of men and women (and that everyone gets married):

Pseudocode for the stable marriage problem is given below:

- All men and all women are initially 'free' (i.e., not engaged)
- while there is a free man m, and a woman he has not proposed to
  - $w = \max_w f(m, w)$
  - if (w is free):
    - m and w become engaged (and are no longer free)
  - else (w is engaged to m'):
    - if w prefers m to m' (i.e., $f(m, w) > f(m', w)$):
      - m and w become engaged
      - m' becomes free

Theorems:

- The algorithm eventually terminates; each step includes a new proposal, and there are only n^2 possible proposals. Furthermore, each iteration improves the matching score of Equation (10.3) (or does nothing).
- The solution is stable. The proof is relatively straihtforward: one assumes an unstable solution in which m and w would prefer to be married to each other rather than their current spouses m' and w'; in this case m must have proposed to w first (since he always proposes to his most compatible match first); so either w was already engaged to somebody preferable, or w accepted and later broke off the engagement; neither of these two scenarios could have ocurred if m' is less compatible.
- Note that a consequence of the above is the possibly surprising conclusion that every matching problem as a stable solution.
- The solution requires $O(n^2)$ steps, since each step includes a proposal, and there are at most n^2 possible proposals. In practice the running time is often substantially faster.
- The solution is not guanarteed to be optimal, and the optimal solution is not guaranteed to be stable. Examples of both are given in Figure 10.2.

[1] The original solution assumed *different* compatibility functions for men toward women, and women toward men, though we discard this for the application discussed here.

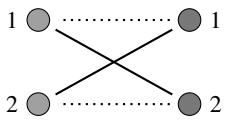


Figure 10.2 Examples of matching solutions that are stable but not optimal (left), and optimal but not stable (right).

10.2.1 AdWords

So far, we have discussed the issue of developing recommendation approaches that consider some notion constraints or 'budgets' for the perspective of users and advertisers. While incorporating constraints is potentially useful in a variety of recommendation scenarios, our solution still doesn't fully address the setting of *ad* recommendation; in particular, when recommending ads, we are unlikely to see the entire set of users (or queries) in advance, before having to select advertisements.

As such, we desire an algorithm that makes decisions (i.e., assigns ads to users) one at a time, while still conforming to matching (or budget) constraints, as in **??** (right).

Formally, the above describes the distinction between an *offline* and an *online* algorithm, i.e., one which sees the entire problem in advance, versus one

AdWords Mehta et al. (2007) is a specific instance of this type of recommendation problem developed for *Google's* online advertising platform.

Mostly, the setting follows the one we described above, though includes a few additional components. Specifically,

- Each advertiser a has a *bid* $f(q,a)$ that they are willing to make for each query q.
- The bid generally refers to how much the advertiser will pay *if* the ad is clicked on; this is determined by an estimated *click-through-rate*, $ctr(q,a)$. As such the expected profit would be $f(q,a) \times ctr(q,a)$, which one can think of as being analogous to the compatibility from **??**.
- Each advertiser has a *budget* $b(a)$ (e.g. for a one-week period).
- As in **??**, there is a limit on the number of ads that can be returned for each query.

Of course the actual implementation of AdWords contains many features not described here, for instance Adwords uses a second-price auction (the winning

advertiser pays the amount that the *second* highest bidder bid), and advertisers don't bid on exact queries, but rather are matched using a 'broad matching' criteria that can include subsets, supersets, or synonyms of keywords being bid on. We refer to Rajaraman and Ullman (2011) for further description of these details, and Mehta et al. (2007) for a more detailed technical description.

10.3 Reinforcement Learning and Bandit Algorithms

So far, when developing matching algorithms in **??** and AdWords in **??**, we've assumed that a compatibility function between between users and ads was available in advance, and merely considered the question of how ads should be *allocated* to users based on budgets and the given compatibility.

However, ad recommendation (among other settings) could present unique challenges in terms of how we estimate compatibility in the first place.

Compared to other recommendation settings, it is unlikely that we could learn user-to-ad compatibility by collecting historical data of ads users have previously interacted with. Some potential issues include:

- Interaction data is extremely sparse; users likely interact with only a tiny fraction of the advertisements that are surfaced to them, making positive feedback extremely rare.
- Even if a user is interested in a particular topic, there are diminishing returns in repeatedly surfacing similar ads (e.g. a user searching for rental tuxedos likely remains interested in the topic for a short time).
- Users do not 'seek out' ads from which we might learn their interests. Thus we must sometimes surface ads before we have collected evidence that a user is interested in them.
- Recommendation is highly contextual. E.g. an ad for rental tuxedos is only worth surfacing if a user is currently searching for a closely related topic.

10.4 Bandit Algorithms and Reinforcement Learning

A Contextual-Bandit Approach to Personalized News Article Recommendation Li et al. (2010)

The basic setup of a *contextual bandit* (following e.g. Li et al. (2010)) is as follows. During each step t, we observe a user u_t and a set of actions (or 'arms') $\mathcal{A}_t$. The actions $a \in \mathcal{A}_t$ reflect, for example, the space of possible advertisements that could be surfaced to a user during a particular timestep.

In the setting of a contextual bandit, each timestep t and action a is associated with a feature vector $x_{a,t}$ representing the current *context*; $x_{a,t}$ is intended to capture any features associated with the user and the action at a particular point in time (which could include information about their search query, etc.).

During every timestep, an action is selected, following which a reward $r_{a,t}$ is received. This reward might measure (e.g.) whether the user clicked on the surfaced advertisement, or the payout received from doing so. Critically, the reward is only observed *for the action a that was selected*, that is, we cannot observe whether the user would have clicked on an ad that *wasn't* shown.

Over time, the goal is to devise a strategy for selecting actions in such a way that the total reward is maximized.

$$E(r_{a,t}|x_{a,t}) = x_{a,t} \cdot \theta_u \tag{10.4}$$

10.4.1 News Recommendation

Doesn't exactly go here but might go somewhere?

Turning down the noise

News search engines and the challenge to traditional journalistic roles

10.5 User Interfaces for Recommendation

Controlling Spotify Recommendations: Effects of Personal Characteristics on Music Recommender User Interfaces Jin et al. (2018)

10.6 Conversation and Explanation

10.6.1 Conversation

Improving recommender systems with adaptive conversational strategies

Enhancing the diversity of conversational collaborative recommendations: a comparison

Towards Conversational Recommender Systems

A Ranking Optimization Approach to Latent Linear Critiquing in Conversational Recommender System

10.6.2 Explanation

Eliciting User Preferences for Personalized Explanations for Video Summaries

10.7 Deployment Considerations

Bibliography

Abdollahpouri, Himan, Burke, Robin, and Mobasher, Bamshad. 2017. Recommender systems as multistakeholder environments. Pages 347–348 of: *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*.

Adamopoulos, Panagiotis, and Tuzhilin, Alexander. 2014. On unexpectedness in recommender systems: Or how to better expect the unexpected. *ACM Transactions on Intelligent Systems and Technology (TIST)*, **5**(4), 1–32.

Adomavicius, Gediminas, and Kwon, YoungOk. 2011. Improving aggregate recommendation diversity using ranking-based techniques. *IEEE Transactions on Knowledge and Data Engineering*, **24**(5), 896–911.

Amer-Yahia, Sihem, Roy, Senjuti Basu, Chawlat, Ashish, Das, Gautam, and Yu, Cong. 2009. Group recommendation: Semantics and efficiency. *Proceedings of the VLDB Endowment*, **2**(1), 754–765.

Anderson, Ashton, Maystre, Lucas, Anderson, Ian, Mehrotra, Rishabh, and Lalmas, Mounia. 2020. Algorithmic effects on the diversity of consumption on spotify. Pages 2155–2165 of: *Proceedings of The Web Conference 2020*.

Bachrach, Yoram, Finkelstein, Yehuda, Gilad-Bachrach, Ran, Katzir, Liran, Koenigstein, Noam, Nice, Nir, and Paquet, Ulrich. 2014. Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces. Pages 257–264 of: *Proceedings of the 8th ACM Conference on Recommender systems*.

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bakshy, Eytan, Messing, Solomon, and Adamic, Lada A. 2015. Exposure to ideologically diverse news and opinion on Facebook. *Science*, **348**(6239), 1130–1132.

Bao, Jie, Zheng, Yu, Wilkie, David, and Mokbel, Mohamed. 2015. Recommendations in location-based social networks: a survey. *GeoInformatica*, **19**(3), 525–565.

Barkan, Oren, and Koenigstein, Noam. 2016. Item2vec: neural item embedding for collaborative filtering. Pages 1–6 of: *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*. IEEE.

Bayer, Immanuel. 2016. fastfm: A library for factorization machines. *The Journal of Machine Learning Research*, **17**(1), 6393–6397.

Bell, Sean, and Bala, Kavita. 2015. Learning visual similarity for product design with

convolutional neural networks. *ACM transactions on graphics (TOG)*, **34**(4), 1–10.

Bellogin, Alejandro, Castells, Pablo, and Cantador, Ivan. 2011. Precision-oriented evaluation of recommender systems: an algorithmic comparison. Pages 333–336 of: *Proceedings of the fifth ACM conference on Recommender systems*.

Bennett, James, Lanning, Stan, et al. 2007. The netflix prize. Page 35 of: *Proceedings of KDD cup and workshop*, vol. 2007. New York.

Bentley, Jon Louis. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, **18**(9), 509–517.

Blei, David M, Ng, Andrew Y, and Jordan, Michael I. 2003. Latent dirichlet allocation. *Journal of machine Learning research*, **3**(Jan), 993–1022.

Bobadilla, Jesús, Ortega, Fernando, Hernando, Antonio, and Gutiérrez, Abraham. 2013. Recommender systems survey. *Knowledge-based systems*, **46**, 109–132.

Bordes, Antoine, Usunier, Nicolas, Garcia-Duran, Alberto, Weston, Jason, and Yakhnenko, Oksana. 2013. Translating embeddings for modeling multi-relational data. Pages 2787–2795 of: *Advances in neural information processing systems*.

Bordes, Antoine, Boureau, Y-Lan, and Weston, Jason. 2016. Learning end-to-end goal-oriented dialog. *arXiv preprint arXiv:1605.07683*.

Bottou, Léon. 2010. Large-scale machine learning with stochastic gradient descent. Pages 177–186 of: *Proceedings of COMPSTAT'2010*. Springer.

Broder, Andrei Z. 1997. On the resemblance and containment of documents. Pages 21–29 of: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE.

Brynjolfsson, Erik, Hu, Yu Jeffrey, and Smith, Michael D. 2006. From niches to riches: Anatomy of the long tail. *Sloan Management Review*, **47**(4), 67–71.

Burke, Robin. 2002. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, **12**(4), 331–370.

Burke, Robin. 2017. Multisided fairness for recommendation. *arXiv preprint arXiv:1707.00093*.

Cai, Chenwei, He, Ruining, and McAuley, Julian. 2017. SPMC: socially-aware personalized markov chains for sparse sequential recommendation. *arXiv preprint arXiv:1708.04497*.

Carbonell, Jaime, and Goldstein, Jade. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. Pages 335–336 of: *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*.

Case, Karl E, and Fair, Ray C. 2007. *Principles of microeconomics*. Pearson Education.

Chang, Shuo, Harper, F Maxwell, and Terveen, Loren Gilbert. 2016. Crowd-based personalized natural language explanations for recommendations. Pages 175–182 of: *Proceedings of the 10th ACM Conference on Recommender Systems*.

Charikar, Moses S. 2002. Similarity estimation techniques from rounding algorithms. Pages 380–388 of: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*.

Chen, Le, Mislove, Alan, and Wilson, Christo. 2016. An empirical analysis of algorithmic pricing on amazon marketplace. Pages 1339–1349 of: *Proceedings of the 25th international conference on World Wide Web*.

Chen, Shuo, Moore, Josh L, Turnbull, Douglas, and Joachims, Thorsten. 2012. Playlist prediction via metric embedding. Pages 714–722 of: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Chouldechova, Alexandra. 2017. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data*, **5**(2), 153–163.

Christakopoulou, Konstantina, Radlinski, Filip, and Hofmann, Katja. 2016. Towards conversational recommender systems. Pages 815–824 of: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*.

Cortes, Corinna, and Vapnik, Vladimir. 1995. Support-vector networks. *Machine learning*, **20**(3), 273–297.

Covington, Paul, Adams, Jay, and Sargin, Emre. 2016. Deep neural networks for youtube recommendations. Pages 191–198 of: *Proceedings of the 10th ACM conference on recommender systems*.

Dacrema, Maurizio Ferrari, Cremonesi, Paolo, and Jannach, Dietmar. 2019. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. Pages 101–109 of: *Proceedings of the 13th ACM Conference on Recommender Systems*.

Davidson, James, Liebald, Benjamin, Liu, Junning, Nandy, Palash, Van Vleet, Taylor, Gargi, Ullas, Gupta, Sujoy, He, Yu, Lambert, Mike, Livingston, Blake, et al. 2010. The YouTube video recommendation system. Pages 293–296 of: *Proceedings of the fourth ACM conference on Recommender systems*.

Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, and Toutanova, Kristina. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Diao, Qiming, Qiu, Minghui, Wu, Chao-Yuan, Smola, Alexander J, Jiang, Jing, and Wang, Chong. 2014. Jointly modeling aspects, ratings and sentiments for movie recommendation (JMARS). Pages 193–202 of: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Ding, Yi, and Li, Xue. 2005. Time weight collaborative filtering. Pages 485–492 of: *Proceedings of the 14th ACM international conference on Information and knowledge management*.

Dodge, Jesse, Gane, Andreea, Zhang, Xiang, Bordes, Antoine, Chopra, Sumit, Miller, Alexander, Szlam, Arthur, and Weston, Jason. 2015. Evaluating prerequisite qualities for learning end-to-end dialog systems. *arXiv preprint arXiv:1511.06931*.

Dong, Li, Huang, Shaohan, Wei, Furu, Lapata, Mirella, Zhou, Ming, and Xu, Ke. 2017. Learning to generate product reviews from attributes. Pages 623–632 of: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*.

Dwork, Cynthia, Hardt, Moritz, Pitassi, Toniann, Reingold, Omer, and Zemel, Richard. 2012. Fairness through awareness. Pages 214–226 of: *Proceedings of the 3rd innovations in theoretical computer science conference*.

Ekstrand, Michael D, Tian, Mucun, Azpiazu, Ion Madrazo, Ekstrand, Jennifer D, Anuyah, Oghenemaro, McNeill, David, and Pera, Maria Soledad. 2018a. All the cool kids, how do they fit in?: Popularity and demographic biases in recommender evaluation and effectiveness. Pages 172–186 of: *Conference on Fairness, Accountability and Transparency*.

Ekstrand, Michael D, Tian, Mucun, Kazi, Mohammed R Imran, Mehrpouyan, Hoda, and Kluver, Daniel. 2018b. Exploring author gender in book rating and recommendation. Pages 242–250 of: *Proceedings of the 12th ACM conference on recommender systems*.

Feng, Shanshan, Li, Xutao, Zeng, Yifeng, Cong, Gao, and Chee, Yeow Meng. 2015. Personalized ranking metric embedding for next new poi recommendation. Pages 2069–2075 of: *IJCAI'15 Proceedings of the 24th International Conference on Artificial Intelligence*. ACM.

Flaxman, Seth, Goel, Sharad, and Rao, Justin M. 2016. Filter bubbles, echo chambers, and online news consumption. *Public opinion quarterly*, **80**(S1), 298–320.

Fleder, Daniel, and Hosanagar, Kartik. 2009. Blockbuster culture's next rise or fall: The impact of recommender systems on sales diversity. *Management science*, **55**(5), 697–712.

Friedman, Jerome, Hastie, Trevor, Tibshirani, Robert, et al. 2001. *The elements of statistical learning*. Vol. 1. Springer series in statistics New York.

Gale, David, and Shapley, Lloyd S. 1962. College admissions and the stability of marriage. *The American Mathematical Monthly*, **69**(1), 9–15.

Ge, Mouzhi, Delgado-Battenfeld, Carla, and Jannach, Dietmar. 2010. Beyond accuracy: evaluating recommender systems by coverage and serendipity. Pages 257–260 of: *Proceedings of the fourth ACM conference on Recommender systems*.

Ge, Rong, Lee, Jason D, and Ma, Tengyu. 2016. Matrix completion has no spurious local minimum. Pages 2973–2981 of: *Advances in Neural Information Processing Systems*.

Ge, Yong, Liu, Qi, Xiong, Hui, Tuzhilin, Alexander, and Chen, Jian. 2011. Cost-aware travel tour recommendation. Pages 983–991 of: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Ge, Yong, Xiong, Hui, Tuzhilin, Alexander, and Liu, Qi. 2014. Cost-aware collaborative filtering for travel tour recommendations. *ACM Transactions on Information Systems (TOIS)*, **32**(1), 1–31.

Godes, David, and Silva, José C. 2012. Sequential and temporal dynamics of online opinion. *Marketing Science*, **31**(3), 448–473.

Golub, Gene H, and Reinsch, Christian. 1971. Singular value decomposition and least squares solutions. Pages 134–151 of: *Linear Algebra*. Springer.

Goodfellow, Ian, Pouget-Abadie, Jean, Mirza, Mehdi, Xu, Bing, Warde-Farley, David, Ozair, Sherjil, Courville, Aaron, and Bengio, Yoshua. 2014. Generative adversarial nets. Pages 2672–2680 of: *Advances in neural information processing systems*.

Gopalan, Prem, Hofman, Jake M, and Blei, David M. 2013. Scalable recommendation with poisson factorization. *arXiv preprint arXiv:1311.1704*.

Graves, Alex. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Guo, Huifeng, Tang, Ruiming, Ye, Yunming, Li, Zhenguo, and He, Xiuqiang. 2017a. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247*.

Guo, Yunhui, Xu, Congfu, Song, Hanzhang, and Wang, Xin. 2017b. Understanding Users' Budgets for Recommendation with Hierarchical Poisson Factorization. Pages 1781–1787 of: *IJCAI*.

Gusfield, Dan, and Irving, Robert W. 1989. *The stable marriage problem: structure and algorithms*. MIT press.

Haim, Mario, Graefe, Andreas, and Brosius, Hans-Bernd. 2018. Burst of the filter bubble? Effects of personalization on the diversity of Google News. *Digital journalism*, **6**(3), 330–343.

Hansen, Christian, Mehrotra, Rishabh, Hansen, Casper, Brost, Brian, Maystre, Lucas, and Lalmas, Mounia. 2021. Shifting Consumption towards Diverse Content on Music Streaming Platforms. Pages 238–246 of: *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*.

Hao, Junheng, Zhao, Tong, Li, Jin, Dong, Xin Luna, Faloutsos, Christos, Sun, Yizhou, and Wang, Wei. P-Companion: A Principled Framework for Diversified Complementary Product Recommendation.

Harper, F Maxwell, and Konstan, Joseph A. 2015. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, **5**(4), 1–19.

He, Ruining, and McAuley, Julian. 2015. VBPR: visual bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1510.01784*.

He, Ruining, and McAuley, Julian. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. Pages 507–517 of: *proceedings of the 25th international conference on world wide web*.

He, Ruining, Packer, Charles, and McAuley, Julian. 2016a. Learning compatibility across categories for heterogeneous item recommendation. Pages 937–942 of: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE.

He, Ruining, Fang, Chen, Wang, Zhaowen, and McAuley, Julian. 2016b. Vista: A visually, socially, and temporally-aware model for artistic recommendation. Pages 309–316 of: *Proceedings of the 10th ACM Conference on Recommender Systems*.

He, Ruining, Kang, Wang-Cheng, and McAuley, Julian. 2017a. Translation-based recommendation. Pages 161–169 of: *Proceedings of the eleventh ACM conference on recommender systems*.

He, Xiangnan, and Chua, Tat-Seng. 2017. Neural factorization machines for sparse predictive analytics. Pages 355–364 of: *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*.

He, Xiangnan, Liao, Lizi, Zhang, Hanwang, Nie, Liqiang, Hu, Xia, and Chua, Tat-Seng. 2017b. Neural collaborative filtering. Pages 173–182 of: *Proceedings of the 26th international conference on world wide web*.

Henderson, Matthew, Al-Rfou, Rami, Strope, Brian, Sung, Yun-Hsuan, Lukács, László, Guo, Ruiqi, Kumar, Sanjiv, Miklos, Balint, and Kurzweil, Ray. 2017. Efficient natural language response suggestion for smart reply. *arXiv preprint arXiv:1705.00652*.

Herlocker, Jonathan L, Konstan, Joseph A, Terveen, Loren G, and Riedl, John T. 2004. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, **22**(1), 5–53.

Hidasi, Balázs, Karatzoglou, Alexandros, Baltrunas, Linas, and Tikk, Domonkos. 2015. Session-based recommendations with recurrent neural networks. *arXiv preprint arXiv:1511.06939*.

Ho, Tin Kam. 1995. Random decision forests. Pages 278–282 of: *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1. IEEE.

Hsiao, Wei-Lin, and Grauman, Kristen. 2018. Creating capsule wardrobes from fashion images. Pages 7161–7170 of: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Hu, Diane, Louca, Raphael, Hong, Liangjie, and McAuley, Julian. 2018. Learning within-session budgets from browsing trajectories. Pages 432–436 of: *Proceedings of the 12th ACM Conference on Recommender Systems*.

Hu, Diane J, Hall, Rob, and Attenberg, Josh. 2014. Style in the long tail: Discovering unique interests with latent variable models in large scale social e-commerce. Pages 1640–1649 of: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Hu, Yifan, Koren, Yehuda, and Volinsky, Chris. 2008. Collaborative filtering for implicit feedback datasets. Pages 263–272 of: *2008 Eighth IEEE International Conference on Data Mining*. Ieee.

Hug, Nicolas. 2020. Surprise: A Python library for recommender systems. *Journal of Open Source Software*, **5**(52), 2174.

Inagawa, Yuma, Hakamta, Junki, and Tokumaru, Masataka. 2013. A support system for healthy eating habits: Optimization of recipe retrieval. Pages 168–172 of: *International Conference on Human-Computer Interaction*. Springer.

Indyk, Piotr, and Motwani, Rajeev. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. Pages 604–613 of: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*.

Ingrande, Jerry, Gabriel, Rodney A, McAuley, Julian, Krasinska, Karolina, Chien, Allis, and Lemmens, Hendrikus JM. 2020. The Performance of an Artificial Neural Network Model in Predicting the Early Distribution Kinetics of Propofol in Morbidly Obese and Lean Subjects. *Anesthesia & Analgesia*, **131**(5), 1500–1509.

Jacobs, Robert A, Jordan, Michael I, Nowlan, Steven J, and Hinton, Geoffrey E. 1991. Adaptive mixtures of local experts. *Neural computation*, **3**(1), 79–87.

Jennings, Andrew, and Higuchi, Hideyuki. 1993. A user model neural network for a personal news service. *User Modeling and User-Adapted Interaction*, **3**(1), 1–25.

Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. 2014. Caffe: Convolutional architecture for fast feature embedding. Pages 675–678 of: *Proceedings of the 22nd ACM international conference on Multimedia*.

Jiang, Yuanchun, Shang, Jennifer, Liu, Yezheng, and May, Jerrold. 2015. Redesigning promotion strategy for e-commerce competitiveness through pricing and recommendation. *International Journal of Production Economics*, **167**, 257–270.

Jin, Yucheng, Tintarev, Nava, and Verbert, Katrien. 2018. Effects of personal characteristics on music recommender systems with different levels of controllability. Pages 13–21 of: *Proceedings of the 12th ACM Conference on Recommender Systems*.

Jones, Karen Sparck. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*.

Joshi, Chaitanya K, Mi, Fei, and Faltings, Boi. 2017. Personalization in goal-oriented dialog. *arXiv preprint arXiv:1706.07503*.

Kabbur, Santosh, Ning, Xia, and Karypis, George. 2013. Fism: factored item similarity models for top-n recommender systems. Pages 659–667 of: *Proceedings of the*

19th ACM SIGKDD international conference on Knowledge discovery and data mining.

Kaminskas, Marius, and Bridge, Derek. 2016. Diversity, serendipity, novelty, and coverage: a survey and empirical analysis of beyond-accuracy objectives in recommender systems. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, **7**(1), 1–42.

Kamiran, Faisal, and Calders, Toon. 2009. Classifying without discriminating. Pages 1–6 of: *2009 2nd International Conference on Computer, Control and Communication*. IEEE.

Kang, Dongyeop, Balakrishnan, Anusha, Shah, Pararth, Crook, Paul, Boureau, Y-Lan, and Weston, Jason. 2019a. Recommendation as a communication game: Self-supervised bot-play for goal-oriented dialogue. *arXiv preprint arXiv:1909.03922*.

Kang, Wang-Cheng, and McAuley, Julian. 2018. Self-attentive sequential recommendation. Pages 197–206 of: *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE.

Kang, Wang-Cheng, Fang, Chen, Wang, Zhaowen, and McAuley, Julian. 2017. Visually-aware fashion recommendation and design with generative image models. Pages 207–216 of: *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE.

Kang, Wang-Cheng, Kim, Eric, Leskovec, Jure, Rosenberg, Charles, and McAuley, Julian. 2019b. Complete the look: Scene-based complementary product recommendation. Pages 10532–10541 of: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Kannan, Anjuli, Kurach, Karol, Ravi, Sujith, Kaufmann, Tobias, Tomkins, Andrew, Miklos, Balint, Corrado, Greg, Lukacs, Laszlo, Ganea, Marina, Young, Peter, et al. 2016. Smart reply: Automated response suggestion for email. Pages 955–964 of: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Kelly, John Paul, and Bridge, Derek. 2006. Enhancing the diversity of conversational collaborative recommendations: a comparison. *Artificial Intelligence Review*, **25**(1-2), 79–95.

Koenigstein, Noam, Ram, Parikshit, and Shavitt, Yuval. 2012. Efficient retrieval of recommendations in a matrix factorization framework. Pages 535–544 of: *Proceedings of the 21st ACM international conference on Information and knowledge management*.

Kolter, J Zico, and Maloof, Marcus A. 2007. Dynamic weighted majority: An ensemble method for drifting concepts. *Journal of Machine Learning Research*, **8**(Dec), 2755–2790.

Konstan, Joseph A, Riedl, John, Borchers, A, and Herlocker, Jonathan L. 1998. Recommender systems: A grouplens perspective. Pages 60–64 of: *Recommender Systems: Papers from the 1998 Workshop (AAAI Technical Report WS-98-08)*. AAAI Press.

Koren, Yehuda. 2009. Collaborative filtering with temporal dynamics. Pages 447–456 of: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Kotkov, Denis, Konstan, Joseph A, Zhao, Qian, and Veijalainen, Jari. 2018. Investigating serendipity in recommender systems based on real user feedback. Pages

1341–1350 of: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*.

Kulesza, Alex, and Taskar, Ben. 2012. Determinantal point processes for machine learning. *arXiv preprint arXiv:1207.6083*.

Kunkel, Johannes, Donkers, Tim, Michael, Lisa, Barbu, Catalin-Mihai, and Ziegler, Jürgen. 2019. Let me explain: impact of personal and impersonal explanations on trust in recommender systems. Pages 1–12 of: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*.

Lakkaraju, Himabindu, McAuley, Julian J, and Leskovec, Jure. 2013. What's in a Name? Understanding the Interplay between Titles, Content, and Communities in Social Media. *ICWSM*, **1**(2), 3.

Li, Hanze, Sanner, Scott, Luo, Kai, and Wu, Ga. 2020a. A Ranking Optimization Approach to Latent Linear Critiquing for Conversational Recommender Systems. Pages 13–22 of: *Fourteenth ACM Conference on Recommender Systems*.

Li, Jing, Ren, Pengjie, Chen, Zhumin, Ren, Zhaochun, Lian, Tao, and Ma, Jun. 2017. Neural attentive session-based recommendation. Pages 1419–1428 of: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*.

Li, Lihong, Chu, Wei, Langford, John, and Schapire, Robert E. 2010. A contextual-bandit approach to personalized news article recommendation. Pages 661–670 of: *Proceedings of the 19th international conference on World wide web*.

Li, Pan, Que, Maofei, Jiang, Zhichao, Hu, Yao, and Tuzhilin, Alexander. 2020b. PURS: Personalized Unexpected Recommender System for Improving User Satisfaction. Pages 279–288 of: *Fourteenth ACM Conference on Recommender Systems*.

Li, Raymond, Kahou, Samira, Schulz, Hannes, Michalski, Vincent, Charlin, Laurent, and Pal, Chris. 2018. Towards deep conversational recommendations. *arXiv preprint arXiv:1812.07617*.

Li, Shuyang, and McAuley, Julian. 2020. Recipes for Success: Data Science in the Home Kitchen. *Harvard Data Science Review*.

Lin, Yankai, Liu, Zhiyuan, Sun, Maosong, Liu, Yang, and Zhu, Xuan. 2015. Learning Entity and Relation Embeddings for Knowledge Graph Completion. Page 2181–2187 of: *AAAI Conference on Artificial Intelligence*.

Linden, Greg, Smith, Brent, and York, Jeremy. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, **7**(1), 76–80.

Ling, Guang, Lyu, Michael R, and King, Irwin. 2014. Ratings meet reviews, a combined approach to recommend. Pages 105–112 of: *Proceedings of the 8th ACM Conference on Recommender systems*.

Liu, Dong C, and Nocedal, Jorge. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, **45**(1-3), 503–528.

Liu, Hui, Yin, Qingyu, and Wang, William Yang. 2018. Towards explainable NLP: A generative explanation framework for text classification. *arXiv preprint arXiv:1811.00196*.

Lovins, Julie Beth. 1968. Development of a stemming algorithm. *Mech. Transl. Comput. Linguistics*, **11**(1-2), 22–31.

Ma, Hao, Yang, Haixuan, Lyu, Michael R, and King, Irwin. 2008. Sorec: social recommendation using probabilistic matrix factorization. Pages 931–940 of: *Proceedings of the 17th ACM conference on Information and knowledge management*.

Maaten, Laurens van der, and Hinton, Geoffrey. 2008. Visualizing data using t-SNE. *Journal of machine learning research*, **9**(Nov), 2579–2605.

Mahmood, Tariq, and Ricci, Francesco. 2007. Learning and adaptivity in interactive recommender systems. Pages 75–84 of: *Proceedings of the ninth international conference on Electronic commerce*.

Mahmood, Tariq, and Ricci, Francesco. 2009. Improving recommender systems with adaptive conversational strategies. Pages 73–82 of: *Proceedings of the 20th ACM conference on Hypertext and hypermedia*.

Majumder, Bodhisattwa Prasad, Li, Shuyang, Ni, Jianmo, and McAuley, Julian. 2019. Generating personalized recipes from historical user preferences. *arXiv preprint arXiv:1909.00105*.

Majumder, Bodhisattwa Prasad, Jhamtani, Harsh, Berg-Kirkpatrick, Taylor, and McAuley, Julian. 2020. Like hiking? You probably enjoy nature: Persona-grounded Dialog with Commonsense Expansions. In: *Empirical Methods in Natural Language Processing*.

Marin, Javier, Biswas, Aritro, Ofli, Ferda, Hynes, Nicholas, Salvador, Amaia, Aytar, Yusuf, Weber, Ingmar, and Torralba, Antonio. 2019. Recipe1m+: A dataset for learning cross-modal embeddings for cooking recipes and food images. *IEEE transactions on pattern analysis and machine intelligence*.

McAuley, Julian, and Leskovec, Jure. 2013a. Hidden factors and hidden topics: understanding rating dimensions with review text. Pages 165–172 of: *Proceedings of the 7th ACM conference on Recommender systems*.

McAuley, Julian, Targett, Christopher, Shi, Qinfeng, and Van Den Hengel, Anton. 2015a. Image-based recommendations on styles and substitutes. Pages 43–52 of: *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*.

McAuley, Julian, Pandey, Rahul, and Leskovec, Jure. 2015b. Inferring networks of substitutable and complementary products. Pages 785–794 of: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*.

McAuley, Julian John, and Leskovec, Jure. 2013b. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. Pages 897–908 of: *Proceedings of the 22nd international conference on World Wide Web*.

McFee, Brian, Bertin-Mahieux, Thierry, Ellis, Daniel PW, and Lanckriet, Gert RG. 2012. The million song dataset challenge. Pages 909–916 of: *Proceedings of the 21st International Conference on World Wide Web*.

McInnes, Leland, Healy, John, and Melville, James. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.

Mehrabi, Ninareh, Morstatter, Fred, Saxena, Nripsuta, Lerman, Kristina, and Galstyan, Aram. 2019. A survey on bias and fairness in machine learning. *arXiv preprint arXiv:1908.09635*.

Mehta, Aranyak, Saberi, Amin, Vazirani, Umesh, and Vazirani, Vijay. 2007. Adwords and generalized online matching. *Journal of the ACM (JACM)*, **54**(5), 22–es.

Mikolov, Tomas, Sutskever, Ilya, Chen, Kai, Corrado, Greg S, and Dean, Jeff. 2013. Distributed representations of words and phrases and their compositionality. Pages 3111–3119 of: *Advances in neural information processing systems*.

Mirza, Mehdi, and Osindero, Simon. 2014. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*.

Mooney, Raymond J, and Roy, Loriene. 2000. Content-based book recommending using learning for text categorization. Pages 195–204 of: *Proceedings of the fifth ACM conference on Digital libraries*.

Narayanan, Arvind, and Shmatikov, Vitaly. 2006. How to break anonymity of the netflix prize dataset. *arXiv preprint cs/0610105*.

Ng, Nathan H, Gabriel, Rodney A, McAuley, Julian, Elkan, Charles, and Lipton, Zachary C. 2017. Predicting surgery duration with neural heteroscedastic regression. Pages 100–111 of: *Machine Learning for Healthcare Conference*. PMLR.

Nguyen, Tan, and Sanner, Scott. 2013. Algorithms for direct 0–1 loss optimization in binary classification. Pages 1085–1093 of: *International Conference on Machine Learning*.

Nguyen, Tien T, Hui, Pik-Mai, Harper, F Maxwell, Terveen, Loren, and Konstan, Joseph A. 2014. Exploring the filter bubble: the effect of using recommender systems on content diversity. Pages 677–686 of: *Proceedings of the 23rd international conference on World wide web*.

Ni, Jianmo, and McAuley, Julian. 2018. Personalized review generation by expanding phrases and attending on aspect-aware representations. Pages 706–711 of: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*.

Ni, Jianmo, Lipton, Zachary C, Vikram, Sharad, and McAuley, Julian. 2017. Estimating reactions and recommending products with generative models of reviews. Pages 783–791 of: *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*.

Ni, Jianmo, Li, Jiacheng, and McAuley, Julian. 2019a. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. Pages 188–197 of: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Ni, Jianmo, Muhlstein, Larry, and McAuley, Julian. 2019b. Modeling heart rate and activity data for personalized fitness recommendation. Pages 1343–1353 of: *The World Wide Web Conference*.

Ni, Jianmo, Hsu, Chun-Nan, Gentili, Amilcare, and McAuley, Julian. 2020. Learning visual-semantic embeddings for reporting abnormal findings on chest x-rays. *arXiv preprint arXiv:2010.02467*.

Ning, Xia, and Karypis, George. 2011. Slim: Sparse linear methods for top-n recommender systems. Pages 497–506 of: *2011 IEEE 11th International Conference on Data Mining*. IEEE.

O'connor, Mark, Cosley, Dan, Konstan, Joseph A, and Riedl, John. 2001. PolyLens: a recommender system for groups of users. Pages 199–218 of: *ECSCW 2001*. Springer.

Pan, Rong, Zhou, Yunhong, Cao, Bin, Liu, Nathan N, Lukose, Rajan, Scholz, Martin, and Yang, Qiang. 2008. One-class collaborative filtering. Pages 502–511 of: *2008 Eighth IEEE International Conference on Data Mining*. IEEE.

Pan, Weike, and Chen, Li. 2013. Gbpr: Group preference based bayesian personalized

ranking for one-class collaborative filtering. In: *Twenty-Third International Joint Conference on Artificial Intelligence*.

Pan, Yingwei, Yao, Ting, Mei, Tao, Li, Houqiang, Ngo, Chong-Wah, and Rui, Yong. 2014. Click-through-based cross-view learning for image search. Pages 717–726 of: *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*.

Pariser, Eli. 2011. *The filter bubble: What the Internet is hiding from you*. Penguin UK.

Park, Seung-Taek, and Chu, Wei. 2009. Pairwise preference regression for cold-start recommendation. Pages 21–28 of: *Proceedings of the third ACM conference on Recommender systems*.

Pizzato, Luiz, Rej, Tomek, Chung, Thomas, Koprinska, Irena, and Kay, Judy. 2010. RECON: a reciprocal recommender for online dating. Pages 207–214 of: *Proceedings of the fourth ACM conference on Recommender systems*.

Porter, Martin F, et al. 1980. An algorithm for suffix stripping. *Program*, **14**(3), 130–137.

Radford, Alec, Jozefowicz, Rafal, and Sutskever, Ilya. 2017. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*.

Rajaraman, Anand, and Ullman, Jeffrey David. 2011. *Mining of massive datasets*. Cambridge University Press.

Rappaz, Jérémie, Vladarean, Maria-Luiza, McAuley, Julian, and Catasta, Michele. 2017. Bartering books to beers: a recommender system for exchange platforms. Pages 505–514 of: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*.

Rashid, Al Mamunur, Albert, Istvan, Cosley, Dan, Lam, Shyong K, McNee, Sean M, Konstan, Joseph A, and Riedl, John. 2002. Getting to know you: learning new user preferences in recommender systems. Pages 127–134 of: *Proceedings of the 7th international conference on Intelligent user interfaces*.

Rendle, Steffen. 2010. Factorization machines. Pages 995–1000 of: *2010 IEEE International Conference on Data Mining*. IEEE.

Rendle, Steffen, Freudenthaler, Christoph, and Schmidt-Thieme, Lars. 2010. Factorizing personalized markov chains for next-basket recommendation. Pages 811–820 of: *Proceedings of the 19th international conference on World wide web*.

Rendle, Steffen, Freudenthaler, Christoph, Gantner, Zeno, and Schmidt-Thieme, Lars. 2012. BPR: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618*.

Rendle, Steffen, Krichene, Walid, Zhang, Li, and Anderson, John. 2020. Neural Collaborative Filtering vs. Matrix Factorization Revisited. *arXiv preprint arXiv:2005.09683*.

Ribeiro, Manoel Horta, Ottoni, Raphael, West, Robert, Almeida, Virgílio AF, and Meira Jr, Wagner. 2020. Auditing radicalization pathways on youtube. Pages 131–141 of: *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*.

Robertson, Stephen. 2004. Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of documentation*.

Roemmele, Melissa. 2016. Writing stories with help from recurrent neural networks. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30.

Ruiz, Francisco JR, Athey, Susan, Blei, David M, et al. 2020. Shopper: A probabilistic model of consumer choice with substitutes and complements. *Annals of Applied Statistics*, **14**(1), 1–27.

Sachdeva, Noveen, and McAuley, Julian. 2020. How Useful are Reviews for Recommendation? A Critical Review and Potential Improvements. *arXiv preprint arXiv:2005.12210*.

Sarwar, Badrul, Karypis, George, Konstan, Joseph, and Riedl, John. 2001. Item-based collaborative filtering recommendation algorithms. Pages 285–295 of: *Proceedings of the 10th international conference on World Wide Web*.

Schlimmer, Jeffrey C, and Granger, Richard H. 1986. Incremental learning from noisy data. *Machine learning*, **1**(3), 317–354.

Sedhain, Suvash, Menon, Aditya Krishna, Sanner, Scott, and Xie, Lexing. 2015. Autorec: Autoencoders meet collaborative filtering. Pages 111–112 of: *Proceedings of the 24th international conference on World Wide Web*.

Smith, Brent, and Linden, Greg. 2017. Two decades of recommender systems at Amazon. com. *Ieee internet computing*, **21**(3), 12–18.

Steck, Harald. 2018. Calibrated recommendations. Pages 154–162 of: *Proceedings of the 12th ACM conference on recommender systems*.

Sugiyama, Kazunari, Hatano, Kenji, and Yoshikawa, Masatoshi. 2004. Adaptive web search based on user profile constructed without any effort from users. Pages 675–684 of: *Proceedings of the 13th international conference on World Wide Web*.

Sun, Fei, Liu, Jun, Wu, Jian, Pei, Changhua, Lin, Xiao, Ou, Wenwu, and Jiang, Peng. 2019. BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer. Pages 1441–1450 of: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*.

Thompson, Cynthia A, Goker, Mehmet H, and Langley, Pat. 2004. A personalized system for conversational recommendations. *Journal of Artificial Intelligence Research*, **21**, 393–428.

Trevisiol, Michele, Chiarandini, Luca, Aiello, Luca Maria, and Jaimes, Alejandro. 2012. Image ranking based on user browsing behavior. Pages 445–454 of: *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*.

Tsymbal, Alexey. 2004. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, **106**(2), 58.

Ueta, Tsuguya, Iwakami, Masashi, and Ito, Takayuki. 2011. A recipe recommendation system based on automatic nutrition information extraction. Pages 79–90 of: *International Conference on Knowledge Science, Engineering and Management*. Springer.

Umberto, Panniello. 2015. Developing a price-sensitive recommender system to improve accuracy and business performance of ecommerce applications. *" International Journal of Electronic Commerce Studies"*, **6**(1), 1–18.

Van Den Oord, Aäron, Dieleman, Sander, and Schrauwen, Benjamin. 2013. Deep content-based music recommendation. In: *Neural Information Processing Systems Conference (NIPS 2013)*, vol. 26. Neural Information Processing Systems Foundation (NIPS).

Van Rijsbergen, Cornelius Joost. 1979. *Information Retrieval. 2nd. Newton, MA*.

Vargas, Saúl, and Castells, Pablo. 2011. Rank and relevance in novelty and diversity metrics for recommender systems. Pages 109–116 of: *Proceedings of the fifth ACM conference on Recommender systems*.

Vargas, Súul. 2011. New approaches to diversity and novelty in recommender systems. Pages 8–13 of: *Fourth BCS-IRSG Symposium on Future Directions in Information Access (FDIA 2011) 4*.

Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Lukasz, and Polosukhin, Illia. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Veit, Andreas, Kovacs, Balazs, Bell, Sean, McAuley, Julian, Bala, Kavita, and Belongie, Serge. 2015. Learning visual clothing style with heterogeneous dyadic co-occurrences. Pages 4642–4650 of: *Proceedings of the IEEE International Conference on Computer Vision*.

Vinyals, Oriol, Toshev, Alexander, Bengio, Samy, and Erhan, Dumitru. 2015. Show and tell: A neural image caption generator. Pages 3156–3164 of: *Proceedings of the IEEE conference on computer vision and pattern recognition*.

Waller, Isaac, and Anderson, Ashton. 2019. Generalists and specialists: Using community embeddings to quantify activity diversity in online platforms. Pages 1954–1964 of: *The World Wide Web Conference*.

Wan, Mengting, and McAuley, Julian. 2018. Item recommendation on monotonic behavior chains. Pages 86–94 of: *Proceedings of the 12th ACM Conference on Recommender Systems*.

Wan, Mengting, Wang, Di, Goldman, Matt, Taddy, Matt, Rao, Justin, Liu, Jie, Lymberopoulos, Dimitrios, and McAuley, Julian. 2017. Modeling consumer preferences and price sensitivities from large-scale grocery shopping transaction logs. Pages 1103–1112 of: *Proceedings of the 26th International Conference on World Wide Web*.

Wan, Mengting, Ni, Jianmo, Misra, Rishabh, and McAuley, Julian. 2020. Addressing Marketing Bias in Product Recommendations. Pages 618–626 of: *Proceedings of the 13th International Conference on Web Search and Data Mining*.

Wang, Chong, and Blei, David M. 2011. Collaborative topic modeling for recommending scientific articles. Pages 448–456 of: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Wang, Xinxi, and Wang, Ye. 2014. Improving content-based and hybrid music recommendation using deep learning. Pages 627–636 of: *Proceedings of the 22nd ACM international conference on Multimedia*.

Wang, Yining, Wang, Liwei, Li, Yuanzhi, He, Di, and Liu, Tie-Yan. 2013. A theoretical analysis of NDCG type ranking measures. Pages 25–54 of: *Conference on Learning Theory*.

Wang, Zhen, Zhang, Jianwen, Feng, Jianlin, and Chen, Zheng. 2014. Knowledge graph embedding by translating on hyperplanes. Pages 1112–1119 of: *Aaai*, vol. 14. Citeseer.

Wang, Zihan, Jiang, Ziheng, Ren, Zhaochun, Tang, Jiliang, and Yin, Dawei. 2018. A path-constrained framework for discriminating substitutable and complementary products in e-commerce. Pages 619–627 of: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*.

Wasserman, Larry. 2013. *All of statistics: a concise course in statistical inference*. Springer Science & Business Media.

Widmer, Gerhard, and Kubat, Miroslav. 1996. Learning in the presence of concept drift and hidden contexts. *Machine learning*, **23**(1), 69–101.

Wilhelm, Mark, Ramanathan, Ajith, Bonomo, Alexander, Jain, Sagar, Chi, Ed H, and Gillenwater, Jennifer. 2018. Practical diversified recommendations on youtube with determinantal point processes. Pages 2165–2173 of: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*.

Wu, Liwei, Li, Shuqing, Hsieh, Cho-Jui, and Sharpnack, James. 2020. SSE-PT: Sequential recommendation via personalized transformer. Pages 328–337 of: *Fourteenth ACM Conference on Recommender Systems*.

Xiang, Liang, Yuan, Quan, Zhao, Shiwan, Chen, Li, Zhang, Xiatian, Yang, Qing, and Sun, Jimeng. 2010. Temporal recommendation on graphs via long-and short-term preference fusion. Pages 723–732 of: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Xiao, Jun, Ye, Hao, He, Xiangnan, Zhang, Hanwang, Wu, Fei, and Chua, Tat-Seng. 2017. Attentional factorization machines: Learning the weight of feature interactions via attention networks. *arXiv preprint arXiv:1708.04617*.

Xu, Kelvin, Ba, Jimmy, Kiros, Ryan, Cho, Kyunghyun, Courville, Aaron, Salakhudinov, Ruslan, Zemel, Rich, and Bengio, Yoshua. 2015. Show, attend and tell: Neural image caption generation with visual attention. Pages 2048–2057 of: *International conference on machine learning*.

Yang, Jaewon, McAuley, Julian, Leskovec, Jure, LePendu, Paea, and Shah, Nigam. 2014. Finding progression stages in time-evolving event sequences. Pages 783–794 of: *Proceedings of the 23rd international conference on World wide web*.

Yao, Sirui, and Huang, Bert. 2017. Beyond parity: Fairness objectives for collaborative filtering. Pages 2921–2930 of: *Advances in Neural Information Processing Systems*.

Yu, Hsiang-Fu, Hsieh, Cho-Jui, Si, Si, and Dhillon, Inderjit. 2012. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. Pages 765–774 of: *2012 IEEE 12th International Conference on Data Mining*. IEEE.

Zafar, Muhammad Bilal, Valera, Isabel, Rogriguez, Manuel Gomez, and Gummadi, Krishna P. 2017. Fairness constraints: Mechanisms for fair classification. Pages 962–970 of: *Artificial Intelligence and Statistics*. PMLR.

Zhang, Jie, and Krishnamurthi, Lakshman. 2004. Customizing promotions in online stores. *Marketing science*, **23**(4), 561–578.

Zhang, Jie, and Wedel, Michel. 2009. The effectiveness of customized promotions in online and offline stores. *Journal of marketing research*, **46**(2), 190–206.

Zhang, Mi, and Hurley, Neil. 2008. Avoiding monotony: improving the diversity of recommendation lists. Pages 123–130 of: *Proceedings of the 2008 ACM conference on Recommender systems*.

Zhang, Shuai, Yao, Lina, Sun, Aixin, and Tay, Yi. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, **52**(1), 1–38.

Zhang, Xingxing, and Lapata, Mirella. 2014. Chinese poetry generation with recurrent neural networks. Pages 670–680 of: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Zhang, Yuan Cao, Séaghdha, Diarmuid Ó, Quercia, Daniele, and Jambor, Tamas. 2012. Auralist: introducing serendipity into music recommendation. Pages 13–22 of: *Proceedings of the fifth ACM international conference on Web search and data mining*.

Zhao, Tong, McAuley, Julian, and King, Irwin. 2014. Leveraging social connections to improve personalized ranking for collaborative filtering. Pages 261–270 of: *Proceedings of the 23rd ACM international conference on conference on information and knowledge management*.

Zheng, Lei, Noroozi, Vahid, and Yu, Philip S. 2017. Joint deep modeling of users and items using reviews for recommendation. Pages 425–434 of: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*.

Zhou, Ke, Yang, Shuang-Hong, and Zha, Hongyuan. 2011. Functional matrix factorizations for cold-start recommendation. Pages 315–324 of: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*.

Zhou, Renjie, Khemmarat, Samamon, and Gao, Lixin. 2010. The impact of YouTube recommendation system on video views. Pages 404–410 of: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*.

Ziegler, Cai-Nicolas, McNee, Sean M, Konstan, Joseph A, and Lausen, Georg. 2005. Improving recommendation lists through topic diversification. Pages 22–32 of: *Proceedings of the 14th international conference on World Wide Web*.