# Predictive Accuracy of Machine Learning Algorithms in Recommender Systems

Marcos Dumón

A Work Submitted for the Degree of Data Science Specialist at
Instituto Tenológico de Buenos Aires

2019

Supervisor: Ph.D. Leticia Gómez.
Director: Ph.D. Alejandro Vaisman.

Typeset using LaTeX

► **To cite this version:**

Marcos Dumón. Predictive Accuracy of Machine Learning Algorithms in Recommender Systems. Artificial Intelligence. Instituto Tecnológico de Buenos Aires - Buenos Aires, 2019. Final Specialization Project. English.

# Predictive Accuracy of Machine Learning Algorithms in Recommender Systems

## Abstract

This work presents a systematic literature review on the application of Machine Learning algorithms in the development of effective movie recommender systems. With the increasing popularity of movie recommender systems in the entertainment industry, selecting appropriate algorithms has become crucial for delivering personalized and accurate recommendations to users. Through an extensive literature search and rigorous methodology, this work identifies and analyzes commonly used Machine Learning algorithms for movie recommendation. The accuracy and performance of these algorithms are evaluated using established evaluation methods and metrics on movie datasets of different sizes. The evaluation takes into account factors such as prediction accuracy, scalability, and robustness. The comparative analysis provides valuable insights into the effectiveness of various Machine Learning algorithms in the context of movie recommendation. The findings contribute to the understanding of algorithmic performance, enabling researchers and practitioners to make informed decisions when developing movie recommender systems. Additionally, the work explores the impact of different hyperparameters and optimization techniques on algorithm performance. The results of this work aim to improve the quality of movie recommendations and enhance user satisfaction. By providing guidelines and recommendations for algorithm selection and optimization, this work contributes to the advancement of movie recommender systems and the overall movie-watching experience.

**Keywords**: Artificial Intelligence - Machine Learning - Recommender Systems - Algorithms - Performance - Collaborative Filtering - Neighbor Based Methods - Matrix Factorization - Supervised Learning - Optimization.

*In loving memory of my grandmother María Elena,*
*who always put the extra into ordinary.*

*It is not knowledge, but the act of learning,*
*not possession but the act of getting there,*
*which grants the greatest enjoyment.*

Carl Friedrich Gauss

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **ML** | Machine Learning |
| **RS** | Recommender Systems or Recomender System |
| **CF** | Collaborative Filtering |
| **ML 100K** | Movielens 100K Dataset |
| **ML 1M** | Movielens 1M Dataset |
| **SGD** | Stochastic Gradient Descent |
| **SVD** | Singular Value Decomposition |
| **K-NN** | $k$-Nearest Neighborss |
| **GP** | Gaussian Process |
| **ERM** | Empirical Risk Minimization |

# Symbols

| | |
|---|---|
| $\bar{r}$ | Average rating of all users and items |
| $b_u$ | The $u$-th user bias, i.e. the $u$-th element of $\mathbf{b}_u$ |
| $b_i$ | The $i$-th item bias, i.e. the $i$-th element of $\mathbf{b}_i$ |
| $r_{ui}$ | The rating of user $u$ on item $i$ |
| $\hat{r}_{ui}$ | The prediction of $r_{ui}$ |
| $d_{uv}$ | Similarity between users |
| $d_{ij}$ | Similarity between items |
| $L$ | Number of neighbors |
| $\mathcal{L}_u$ | Set of neighbors (for user $u$) |
| $\mathcal{L}_i$ | Set of neighbors (for item $i$) |
| $U$ | Number of users |
| $I$ | Number of items |
| $\mathbf{R}$ | The matrix of $r_{ui}$ values, i.e. the matrix of ratings |
| $\mathcal{D}$ | The set of $(u, i)$ indices of $\mathbf{R}$ where a rating is provided |

| | |
|---|---|
| $\mathbf{D}$ | Similarity matrix |
| $\mathbf{P}$ | The user feature matrix |
| $\mathbf{Q}$ | The item feature matrix |
| $\mathbf{x}_i$ | The $i$-th example (input) from a dataset |
| $\mathbf{y}_i$ , $y_i$ | The target associated with $\mathbf{x}_i$ for supervised learning |
| $\mathbf{X}$ | The $m \times n$ matrix with input examples $\mathbf{x}_i$ in row $\mathbf{X}_i$ |
| $\mathbf{b}_u$ | The user biases |
| $\mathbf{b}_i$ | The item biases |
| $\mathbf{r}_u$ | Vector containing the ratings provided by users |
| $\mathbf{r}_i$ | Vector containing the ratings that items have received |
| $\mathbf{p}_u$ | The user feature vector |
| $\mathbf{q}_i$ | The item biases |
| $u, v \in \{1, \ldots, U\}$ | Indices for users |
| $i, j \in \{1, \ldots, I\}$ | Indices for items |
| $\mathbf{R}_{i,j}$ | Element $i$, $j$ of matrix $\mathbf{R}$ |
| $\mathbf{R}_{i,:}$ | Row $i$ of $\mathbf{R}$ |
| $\mathbf{R}_{:,i}$ | Column $i$ of $\mathbf{R}$ |
| $\mathbb{A}$ | A set |

| | |
|---|---|
| $a \in \mathbb{A}$ | Element $a$ in the set $\mathbb{A}$ |
| $\mathbb{A} \setminus \mathbb{B}$ | Complement of set $\mathbb{A}$ in the set $\mathbb{B}$ |
| $\mathbb{A} \subset \mathbb{B}$ | The set $\mathbb{A}$ is a subset of the set $\mathbb{B}$ |
| $\mathbb{A} \cap \mathbb{B}$ | Intersection of sets $\mathbb{A}$ and $\mathbb{B}$ |
| $\mathbb{A} \cup \mathbb{B}$ | Union of sets $\mathbb{A}$ and $\mathbb{B}$ |
| $\lvert\mathbb{A}\rvert$ | Number of elements in the set $\mathbb{A}$ |
| $\mathbb{R}$ | Set of real numbers |
| $\mathbb{R}^+$ | Set of non-negative real numbers |
| $\mathbb{R}^n$ | Set of real-valued $n$-dimensional vectors |
| $\mathbb{R}^{n \times m}$ | Set of real value matrices of $n \times m$-dimensions |
| $\mathbb{N}$ | Set of natural numbers, i.e., $\{0, 1, \dots\}$ |
| $\{0, 1, \dots, n\}$ | The set of all integers between 0 and $n$ |
| $[a, b]$ | Closed interval between $a$ and $b$ |
| $\{a, b, c\}$ | Set that contains elements $a$, $b$ and $c$ |
| $\nabla_{\boldsymbol{\Theta}} f(\boldsymbol{\Theta}) = \frac{\partial f(\boldsymbol{\Theta})}{\partial \boldsymbol{\Theta}} \in \mathbb{R}^n$ | Gradient of scalar field $f(\boldsymbol{\Theta}) : \mathbb{R}^n \to \mathbb{R}$ w.r.t. vector $\boldsymbol{\Theta}$ |
| $\frac{\partial f(\Theta)}{\partial \Theta}$ | Partial derivative of scalar field $f(\Theta) : \mathbb{R}^n \to \mathbb{R}$ w.r.t. the scalar $\Theta$ |
| $\mathbf{R}^\top$ | Transpose of matrix $\mathbf{R}$ |
| $\mathbf{A}^{-1}$ | Inverse of matrix $\mathbf{A}$ |

| | |
|---|---|
| $\mathcal{N}(x, \mu, \sigma)$ | Gaussian distribution over $x$ with mean $\mu$ and covariance $\sigma$ |
| $z \sim \mathcal{D}$ | $z$ is sampled according to $\mathcal{D}$ |
| $\mathbb{P}, \mathbb{E}$ | Probability and expectation of a random variable |
| $\mathcal{X}$ | Instances domain (a set) |
| $\mathcal{Y}$ | Labels domain (a set) |
| $f : \mathcal{X} \to \mathcal{Y}$ | Function $f$ with domain $\mathcal{X}$ and range $\mathcal{Y}$ |
| $\underset{\Delta}{\operatorname{argmax}} f(x \vert \Delta)$ | The argument $\Delta$ for which $f$ has its maximum value |
| $\underset{\Delta}{\operatorname{argmin}} f(x \vert \Delta)$ | The argument $\Delta$ for which $f$ has its minimum value |
| $\Vert\mathbf{R}\Vert_{\mathrm{F}} = \sqrt{\sum_{u=1}^{m} \sum_{i=1}^{n} \vert r_{ui}\vert^2}$ | Frobenius norm |
| $x_i, v_i, w_i$ | The $i$th element of a vector |
| $a \cdot b$ | Dot product between two vectors |
| $\forall x$ | Universal quantifier: for all $x$ |
| $a =: b$ | $b$ is defined as $a$ |
| $\exists x$ | Existential quantifier: exists $x$ |
| $\vert$ | Such that |
| $\Vert\mathbf{x}\Vert_2 = \sqrt{\sum_{i=1}^{d} x_i^2}$ | The $L_2$ norm of $\mathbf{x}$ |
| $\nabla f(\mathbf{\Theta})$ | The gradient of a function $f : \mathbb{R}^d \to \mathbb{R}$ at $\mathbf{\Theta}$ |
| $\mathbb{1}_{[\text{BOOLEAN EXPRESSION}]}$ | Indicator function (equals 1 if expression is true and 0 otherwise) |

# Chapter 1

# Introduction

In this chapter, we will explore the origins of recommender systems in Section 1.1. The primary objective of this work is defined in Section 1.2, followed by a presentation of the research method employed in this study in Section 1.2.3. Lastly, the limitations of this work will be discussed in Section 1.2.4.

## 1.1 Origins and general overview of RS

Recommender systems (RS) play a significant role in enhancing our online experiences by leveraging data analysis to deliver personalized suggestions. These systems are particularly crucial in the current digital landscape, where users are confronted with an abundance of choices. According to [BOHG13]:

> *A recommender system can be defined as a set of programs that attempt to recommend the most suitable items to particular users by predicting a user's interest in an item based on the information about the items, the users and the interactions between items and users.*

Companies in various industries such as e-commerce, news, video-on-demand, professional networking and music streaming, use RS to increase sales, engagement and user experience. The RS field can be traced back to the mid-1990s with the development of Tapestry [GNOT92], the first collaborative filtering manual mail system, at Xerox Palo

Alto Research Center[1]. The goal of RS is to filter and predict vital information from large volumes of data on user preferences and behavior. Learning algorithms take historical data to infer rules, correlations and preferences. This information is then used to select and suggest various items, including songs, books, news, movies, clothes, or restaurants, tailored to the preferences of individual users. Historical data can be heterogeneous, such as specific user information like age and occupation or item characteristics like genre, year and actors. Users' explicit comments, indicating their feelings, or implicit comments; collected through their interactions with items, can also be interpreted as signs of interest.

Figure 1.1 depicts the fundamental concept of a RS. The system takes in a data stream and filters it to deliver the most pertinent recommendations to users. The filtering rules, which depend on the ratings provided, are subject to continuous change due to new data inputs. The users who give the ratings may or may not be the same users who receive the recommendations, as collaborative filtering systems use data from various users or items to make recommendations. The system combines these ratings with various Machine Learning algorithms to construct a model that generates recommendations based on a user's rating history. As users receive recommendations, they continue to provide feedback, and the model is updated and fine-tuned to enhance the accuracy of the recommendations.

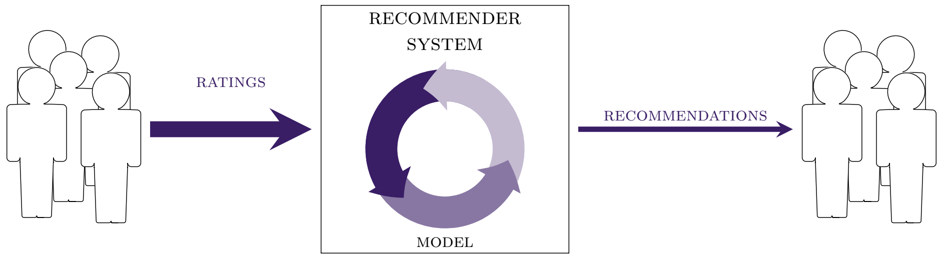


FIGURE 1.1: Concept of a RS.

To illustrate this point, we will examine the recommendation problem from the perspective of Amazon Inc.[2], a United States-based e-commerce company. Amazon's platform provides users with the ability to purchase and sell goods or services. Customers typically come to Amazon with the intention of purchasing a specific item, such as a smartphone. Prior to making a purchase, they may assess their budget and estimated delivery date. Initially, users engage with Amazon's platform by searching for the product by name or category. This generates a list of different products that Amazon offers, sorted or categorized by relevance. By default, products are ordered by relevance, but users have the option to sort them by other factors such as customer reviews, price or product condition (new or

[1]https://www.parc.com
[2]https://www.amazon.com

used). These results are not directly related to the RS but rather are generated by a sorting, search or classification algorithm. Users interact with the RS when they select a product from the list provided by any of the algorithms mentioned above. When users click on the item of interest, they are shown the product details along with additional product groups that are directly related to the item. Below are the additional groups, which represent the direct output of the recommendation algorithm:

*"People who liked this product also liked",*
*"Often bought together",*
*"Based on your current browsing history, you may also be interested in".*

## 1.2 Goal of this work

### 1.2.1 Research objective

This work aims to examine the issue of offline computational accuracy in recommendation algorithms and assess the scalability of their predictions in relation to dataset size. The primary objective is to identify the optimal conditions under which a recommendation algorithm exhibits the highest efficiency as the volume of data increases. While this research does not provide a conclusive solution regarding the selection of a specific algorithm, external factors such as dataset structure and preferred prediction types must be taken into account. Nonetheless, it is anticipated to offer valuable guidelines for algorithm selection based on dataset properties and size.

### 1.2.2 Specific research objectives

To achieve the central objective we compare memory-based methods with model-based methods using the ML 100K and ML 1M datasets. We use multiple error metrics for collaborative filtering, as their efficiency depends on the specific case being analyzed. We evaluate how hyperparameters such as regularization, number of epochs, learning rate, etc., impact model-based methods, while assessing the number of neighbors and similarity metrics for memory-based methods. We also analyze the performance of Bayesian optimization in hyperparameter optimization and compare it with random search. Finally, we evaluate the robustness of the algorithms under different levels of sparsity in the rating matrix. Additionally, we aim to provide evidence supporting the superiority of model-based methods over memory-based methods in the datasets analyzed in this work, utilizing the aforementioned error metrics.

### 1.2.3 Research methodology

The primary research methodology employed in this work is experimental, conducted in a controlled environment and quantitative in nature; as the results are evaluated numerically. We test multiple algorithms using collaborative filtering and other techniques to enable comparison of their accuracy and speed. Once the algorithms are run on the datasets, we employ metrics to evaluate their performance.

### 1.2.4 Limitations

The field of RS encompasses a broad range of topics such as Machine Learning, data extraction and more. As such, the number of methods and theories is extensive, and the scope of this work is limited to the two methods mentioned above. While there are various aspects to RS performance, including accuracy, scalability, memory and time consumption, this work focuses primarily on the precision of each algorithm.

### 1.2.5 Project outline

The present work is structured into several chapters, each addressing to specific aspects of RS. Chapter 2 provides a comprehensive literature review, examining AI, Machine Learning, as well as collaborative filtering, content-based RS and knowledge-based RS. These three approaches represent widely adopted methodologies in constructing RS. Chapter 3 delves into the methodologies commonly employed in collaborative filtering, encompassing both memory-based and model-based methods. Evaluation metrics for assessing their performance are also discussed. In Chapter 4, hyperparameter optimization is defined and explored, covering various techniques such as manual tuning, grid search, random search and Bayesian optimization. Chapter 5 details the experimental setup, including data analysis, dataset partitioning, and training, validation and testing techniques. The chapter also encompasses the reporting, analysis, and evaluation of experimental results. Finally, Chapter 6 presents the key conclusions derived from the conducted research and proposes potential directions for future investigation.

# Chapter 2

# State of the Art

In this chapter, we present a brief overview of key concepts relevant to this work. First, in Section 2.1, we introduce the topics of Artificial Intelligence and Machine Learning and their relationship to RS. Next, in Section 2.3.2, we provide a taxonomy of the current research in the field of RS. We also highlight the challenges and limitations in RS in Section 2.3.3. Furthermore, we formally define the specific recommendation problem that we aim to address in this work in Section 2.3.4. Finally, in Section 2.3.5, we compare the traditional classification problem with collaborative filtering.

## 2.1 Artificial Intelligence

The quest to replicate human intelligence in machines has been a central goal of the field of Artificial Intelligence (AI) since its inception. AI aims to design machines capable of performing tasks that require reasoning, learning, problem-solving, planning, and perception. According to computer scientist John McCarthy [McC98, McC07], AI is:

> *The science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.*

The concept of intelligence itself has been subject to much debate in the field of AI. According to [Got97], *"intelligence is a very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience"*. Various types and degrees of intelligence are observed in humans, animals, and some machines. Intelligence is a fundamental characteristic that allows living organisms to adapt and flourish in their surroundings. The goal of AI is to replicate this characteristic in machines, creating intelligent machines capable of performing a broad range of tasks. Determining whether a machine is intelligent is a fundamental question in AI. Alan Turing addressed this question in his article *"Computational Machinery and Intelligence"*, published in the philosophy journal Mind in 1950 [Tur50]. He proposed the Turing Test, also known as "The Imitation Game", which assesses a machine's intelligence and its ability to "think". He writes:

> *If there is a machine behind a curtain and a human is interacting with it (by any means, for example, audio or writing, etc.) and if the human feels that he is interacting with another human, then the machine is artificially intelligent.*

To pass the test, a machine must sufficiently impersonate a human in a written conversation with an interrogator in real-time such that the interrogator cannot reliably distinguish between the machine and a real human. In Figure 2.1 we can see the interrogator, a human (user) and a machine.

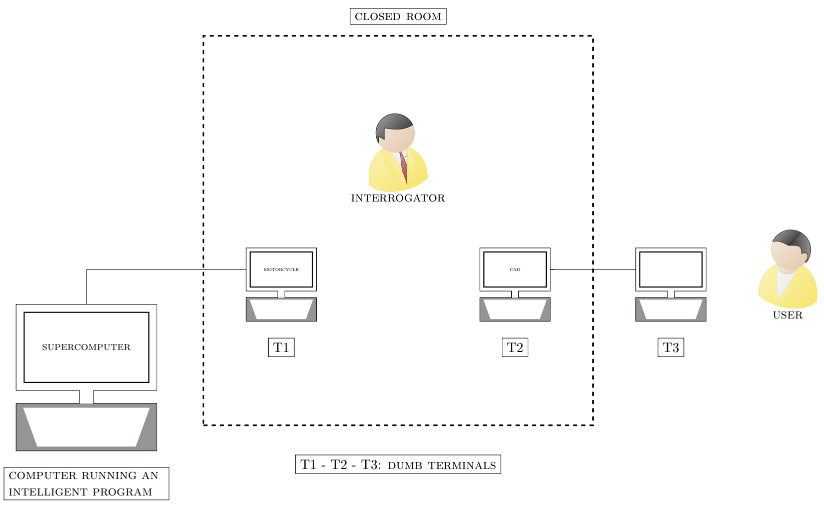

CLOSED ROOM
INTERROGATOR
MOTORCYCLE
CAR
SUPERCOMPUTER
T1
T2
T3
USER
COMPUTER RUNNING AN INTELLIGENT PROGRAM
T1 - T2 - T3: DUMB TERMINALS

FIGURE 2.1: Turing Test (Imitation Game).

The interrogator is in a room separated from the other human and the machine. The interrogator is not able to see or speak directly to any of the other two, and does not know which entity is a machine, and communicates to these two solely by textual devices like a dumb terminal. The goal of the game is for the interrogator to determine which of the other two is the human and which is the machine [OD19]. To determine it, the interrogator is supposed to distinguish the machine from the human solely based on the answers received for the questions asked over the dumb terminal. After having asked the number of questions, if the interrogator is not able to distinguish the machine from the human, then, as per the argument of Alan Turing, the machine can be considered intelligent.

The birth of AI as a discipline can be traced back to the Dartmouth Conference of 1956, where McCarthy, along with Marvin Minsky, Nathan Rochester, and Claude Shannon organized a conference to bring together leading experts to explore the study of intelligent machines [MMRS06]. During the conference, the participants discussed the premise that:

> *Every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it.*

Following the conference, the field of AI grew into a multidisciplinary field that encompasses computer science, engineering, psychology, philosophy, ethics and more. One of the primary goals of AI is to design technology to accomplish highly specialized functions, such as computer vision, speech processing, pattern analysis and prediction in data. This focus on specific intelligent tasks is referred to as “Weak” AI [Vel12]. For example, IBM’s Deep Blue chess-playing system that beat the world chess champion, Garry Kasparov, in 1997, is an example of a Weak AI machine. Instead of simulating how a human would play chess, Deep Blue used brute force techniques to calculate probabilities to determine its offensive and defensive moves. In contrast, the term “Strong AI”, introduced by philosopher John Searle in 1980, refers to the goal of building machines with Artificial General Intelligence, which has intellectual ability indistinguishable from that of human beings [Sea80, CP04]. AI technology can take on many forms, including standalone hardware or software, distributed across computer networks or embodied in a robot. AI can also be in the form of intelligent autonomous agents, such as virtual or robotic entities capable of interacting with their environment and making decisions on their own. Furthermore, AI technology can also be coupled with biological processes, such as brain-computer interfaces (BCIs), made of biological materials (biological AI), or as small as molecular structures (nanotechnology).

## 2.2 Machine Learning

Before discussing Machine Learning (ML), it is crucial to have a clear understanding of "learning", in human terms. Learning is the *process by which we acquire new knowledge, skills, or behaviors through study or experience.* Learning provides us with the flexibility to adapt to new circumstances and learn new strategies, regardless of our age. Generalization is the most important aspect of learning, which involves recognizing similarities between different situations to apply what we have learned in one context to another. For instance, let us consider the acquisition of mathematical skills. While adding natural numbers may be simple for computers, they cannot rely on memorization as each term in the sum has infinitely many values. It would be impossible to store the triple $(x, y, x + y)$ for every combination of the two values $x$ and $y$. So, how do humans learn mathematics? A teacher explains the process and students practice on examples until they can apply what they have learned to new ones without making mistakes. After approximately 30 examples, a student should understand addition. This means that after only 30 examples the student can apply what they have learned to infinitely many new examples, which they have not encountered before. This is what makes learning useful since it allows us to extrapolate our knowledge to different situations. To be considered intelligent, a system that operates in a changing environment must have the ability to learn. If the system can learn and adapt to changes, the system designer does not need to anticipate and provide solutions for all possible scenarios [MRT12].

The concept of human learning differs from that of learning in the context of ML, which involves the use of learning algorithms. A learning algorithm is an algorithm that is able to *learn from data.* Arthur Samuel coined the term "Machine Learning" in 1959, in his article *"Some Studies in Machine Learning Using Checkers"* [Sam59]. Samuel defined ML as a subfield of AI that seeks to enable computers to learn without explicitly being programmed. But what does it mean for a machine to learn? Mitchell [Mit97] offers a widely accepted definition, as shown in Definition 2.1:

**Definition 2.1.** *A computer program learns from experience E, with respect to a class of tasks T and performance measure P, if its performance at tasks in T improves as measured by P, with experience E.*

One practical application of Definition 2.1 is the development of a RS. Here, the task is not the learning process itself, but rather the ability to perform accurate and relevant recommendations to users based on their preferences or past behavior. Learning, in this case, serves as the means to enable the computer to make better recommendations. For example, a ML algorithm could be trained on a dataset of user preferences and past behavior, such as movies or products, to learn how to make recommendations that the user is likely to enjoy or find useful. The goal is to improve the algorithm's ability to make accurate and relevant recommendations over time. Many kinds of tasks can be solved with ML, and RS are just one example. Other common ML tasks include classification of objects into two or more classes, or regression, which involves predicting a continuous value.

To evaluate the effectiveness of a ML algorithm, it is essential to have a quantitative measure that can assess its performance. This measure, denoted as $P$, is, or should be, designed specifically for the task $T$ that the system is performing. By customizing the performance measure to the task at hand, we can obtain a more accurate assessment of the algorithm's capabilities. It is typically crucial to assess the performance of a ML algorithm on unseen data to determine its efficacy in practical applications. To achieve this, we evaluate its performance metrics using a distinct test dataset that is separate from the one used for training the system. This allows us to predict its performance in real-world scenarios accurately.

Lastly, it is worth noting that experience $E$ refers to the data that the ML algorithm is trained on. It is the input that the algorithm uses to learn and improve its performance. The nature of this experience can significantly impact the learning process, as it can be either labeled or unlabeled. In *supervised learning*, the algorithm is trained on *labeled data*, where each example has an associated label that tells the algorithm what the correct output should be. The labeled data provides the algorithm with experience, which it uses to learn the mapping between the input and the output.

On the other hand, *unsupervised learning* involves training the algorithm on *unlabeled data*, which does not have any associated labels. Here, the algorithm is left to find patterns or structure in the data on its own, which makes it well-suited for tasks such as clustering or anomaly detection, where the goal is to find interesting patterns or anomalies in the data. Thus, the type of experience that a ML algorithm receives can have a significant impact on its ability to learn and perform well on various tasks [GBC16].

The term "learning" in this work refers to finding an algorithm that meets the conditions set forth in Definition 2.1 and can be executed by a computer. From now on, when referring to learning, we specifically mean *finding an algorithm that meets the criteria outlined in Definition 2.1 and can be executed by a computer.*

### 2.2.1 Supervised learning framework

This section introduces two primary frameworks in supervised learning: classification and regression. We provide a comprehensive discussion of the essential concepts and algorithms within each framework, which will be essential to our work going forward. By delving into these frameworks, we aim to establish a solid foundation for our subsequent analyses and investigations. This section's presentation intends to lay the groundwork for introducing RS, which constitute the principal application task considered in this work.

#### 2.2.1.1 Datasets, feature selection and extraction

A dataset comprises numerous examples, which are sometimes referred to as data points. Among the earliest datasets analyzed by statisticians and ML researchers is the Iris dataset, which was introduced by Fisher [Fis36]. The dataset comprises measurements of various parts of 150 iris plants, with each plant corresponding to one example. The features of each example consist of the measurements of each part of the plant, such as the sepal length, sepal width, petal length, and petal width. Additionally, the dataset includes the species to which each plant belonged, with three distinct species being represented.

The majority of ML algorithms are designed to process datasets, which consist of collections of examples, each containing a set of features. A dataset can be described in various ways, one of which is by using a matrix of features or a design matrix. This type of matrix contains one example per row, with each column representing a specific feature. The Iris Dataset[1], for instance, comprises 150 examples, each consisting of four features. To represent this dataset as a matrix of features, we can use $\mathbf{X} \in \mathbb{R}^{150\times 4}$, where $\mathbf{X}_{i,1}$ denotes the sepal length of plant $i$, $\mathbf{X}_{i,2}$ represents the sepal width of plant $i$. In many cases, while dealing with a dataset that contains a matrix of features, denoted as $\mathbf{X}$, it is also common to provide a vector of labels, represented by $\mathbf{y}$, where each $y_i$ corresponds to the label for a specific example $i$.

In ML, it is often necessary to perform a pre-processing step to extract or select features that are needed to accurately classify or cluster data. Features refer to individual measurements or characteristics that describe the data. For instance, in Figure 2.2, the Sepal width and Sepal length are physical features that were taken from a real, existing plant, allowing for the accurate discrimination of the three species. The need for feature

[1] https://archive.ics.uci.edu/ml/datasets/iris

extraction or selection arises due to the complexity and noise often present in raw data, which can make processing it without such steps difficult.

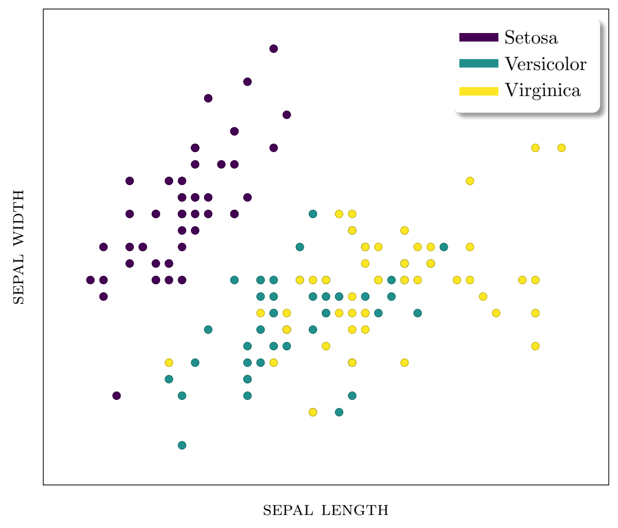


FIGURE 2.2: Example of Iris dataset for two features. Color of points indicate the plant specie.

Feature *selection* involves reducing the number of features by eliminating noisy ones and retaining those that best satisfy a selection criterion. On the other hand, feature *extraction* is a form of dimensionality reduction that involves transforming the raw data into derived values or features that are better suited for modeling purposes.

#### 2.2.1.2 An overview of learning theory

In ML, a supervised learning algorithm is used to learn a predictive model from a set of *examples*, known as a training set. Each example in the this set comprises a pair (*observation, response*). The objective of the learning process is to find a predictive model that can accurately predict the outputs for *new examples* that are not part of the training set. To measure the performance of the predictive model, a *loss* function is used to quantify the disagreement between the predicted and desired output, also known as a label. The goal of the learning algorithm is to select the predictive model that minimizes the average error on the training set, referred to as the *empirical risk*. This approach, known as *Empirical Risk Minimization* (ERM), aims to minimize the empirical risk in the hope that the resulting predictive model will have a low *generalization error* on new examples. The assumption underlying ERM is that the new examples share similarities with the training examples used to learn the predictive model. In the subsequent sections we will provide a detailed description of these concepts.

#### 2.2.1.3 Problem formalization

In supervised learning problems we assume that we have access to a domain $\mathcal{T}$ representing the space $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$, equipped with the probability distribution $\mathcal{P}_{\mathcal{T}}$. In this context, $\mathcal{X} \subseteq \mathbb{R}^d$ denotes the example space, while $\mathcal{Y}$ refers to the label space. Our goal is to discover the correct relationship between examples in $\mathcal{X}$ and their corresponding labels in $\mathcal{Y}$. Specifically, we use a supervised learning algorithm $\mathcal{A}$ to learn the relationship between input and output variables. When provided with a training dataset, namely, a finite-sized sample of data $\mathcal{D} = \{\mathbf{z}_i = (\mathbf{x}_i, y_i)\}_{i=1}^n$, the learning algorithm produces a model or hypothesis $h : \mathcal{X} \to \mathcal{Y}$ from a set of possible hypotheses $\mathcal{H}$ that can associate each example in $\mathcal{X}$ with its corresponding value in $\mathcal{Y}$. Thus, the goal of learning is to identify the hypothesis $h \in \mathcal{H}$ that *best* matches the observed data. This hypothesis is denoted as $h^\star$ and is selected from the set of all possible hypotheses, $\mathcal{H}$.

It is important to note that the distribution $\mathcal{P}_{\mathcal{T}}$ is unknown in ML. The dataset $\mathcal{D}$ is assumed to be representative of the true distribution. Therefore, the main goal of ML is to use $\mathcal{D}$ to learn a model or hypothesis $h$ that can generalize well to new examples drawn from $\mathcal{P}_{\mathcal{T}}$. To introduce the concept of generalization, we first need to define two notions of risk associated with a hypothesis. However, before doing so, we must address the problem of assessing the performance of a hypothesis on a given example.

Assessing the performance of a model for a given problem is essential, and measuring its error is a widely used approach. However, the concept of error may vary across different problems. For instance, consider a scenario where $\mathbf{z}_i = (\mathbf{x}_i, y)$ and the problems of *classification* and *regression* need to be solved. In the *classification* setting, $y_i \in \{1, \ldots, C\}$ belongs to a finite set of categorical variables. The objective is to accurately classify an object, and therefore, an error is the prediction of an incorrect class, i.e., $h(\mathbf{x}) \neq y$. For example, in the case of email messages $\mathcal{Y} = \{\text{SPAM}, \text{NOT SPAM}\}$ could be the categorical labels, and the objective could be to predict whether a new email is spam or not. If $C = 2$, the task is referred to as binary classification, whereas for $C > 2$ it is known as multi-class classification. Another example of multi-class classification is recognizing handwritten digits, where the labels could be the digits $0 - 9$. On the other hand, in regression, the label space $\mathcal{Y}$ is continuous, i.e., $\mathcal{Y} = \mathbb{R}$ and the goal is to predict the correct value for an example. In this case, an error is the prediction of a value significantly different from the ground truth, i.e., $h(\mathbf{x}) \ll y$ or $h(\mathbf{x}) \gg y$. In the *regression* setting, $y_i$ is a real number, such as the price of a house or the amount of rainfall in a region during a given period. The goal of regression is to predict the price of a new house based on its features or to forecast the amount of rainfall for the next month using historical data.

#### 2.2.1.4 Central assumption of statistical learning theory

When dividing a dataset into training, validation and testing subsets, the principle of independence and identical distribution is *implicitly* assumed. Failure to satisfy this assumption due to sampling dependencies or distributional differences may lead to sub-optimal generalization performance of a ML model in real-world scenarios. Statistical learning theory assumes that all examples in a dataset are independently and identically distributed (*i.i.d.*) according to an unknown probability distribution $\mathcal{P}_{\mathcal{T}}$. This assumption is essential for generalizing from the training to the validation set and validating a statistical model. In other words, the *i.i.d.* assumption implies that any set of examples $(x_i, y_i) \in \mathcal{D}$ is generated *i.i.d.* according to $\mathcal{P}_{\mathcal{T}}$, namely, that examples are independent of each other and identically distributed from the same probability distribution. Thus, $\mathcal{D}$ can be considered an *i.i.d.* sample that follows $\mathcal{P}_{\mathcal{T}}$.

#### 2.2.1.5 Loss functions, true risk and empirical risk

Loss functions are a critical component of ML, serving to measure the level of error or risk. In the context of a hypothesis function $h$, it is necessary to evaluate the extent to which the predicted output $h(\mathbf{x})$ matches the desired output $y$, for a given pair $(x, y)$. Specifically, we consider a loss function $\ell : \mathcal{H} \times \mathcal{Z} \to \mathbb{R}^+$, which, given a hypothesis $h \in \mathcal{H}$ and an example $\mathbf{z} = (\mathbf{x}, y) \sim \mathcal{P}_{\mathcal{T}}$, returns a positive real value in $\mathbb{R}^+$. This value serves as a numerical representation of the error committed by the hypothesis on the example. A loss function, such as $\ell(h, \mathbf{z})$, essentially quantifies the discrepancy between the predicted and desired output. Typically, the loss function is defined as a distance metric over the set of possible outputs $\mathcal{Y}$. In the context of classification, we can employ the 0/1 loss, which can be defined as follows:

$$\ell(h, \mathbf{z}) = \begin{cases} 0 & \text{if } h(\mathbf{x}) = y, \\ 1 & \text{otherwise.} \end{cases} \tag{2.1}$$

On the other hand, in regression, we can use the absolute loss function, which can be defined as:

$$\ell(h, \mathbf{z}) = |h(\mathbf{x}) - y| \tag{2.2}$$

The error associated with a prediction hypothesis $h$ on all examples $(x, y)$ from $(\mathcal{Y} \times \mathcal{Y})$ is known as *true risk*. For classification, the true risk error is defined in Definition 2.2.

**Definition 2.2.** True risk (Generalization error). *Given a loss function $\ell : \mathcal{H} \times \mathcal{Z} \to \mathbb{R}^+$ and a distribution $\mathcal{P}_\mathcal{T}$, the true risk $R_\mathcal{T}(h)$ of a hypothesis $h$ is defined as:*

$$R_\mathcal{T}(h) = \mathop{\mathbb{E}}_{\mathbf{z} \sim \mathcal{P}_\mathcal{T}} \ell(h, \mathbf{z}) = \int_{\mathcal{X} \times \mathcal{Y}} \ell(h, \mathbf{z}) d\mathcal{P}_\mathcal{T}(x, y) \tag{2.3}$$

where $\mathbb{E}$ is the expected value, $\ell$ is the loss function and $\mathcal{P}_\mathcal{T}$ represents the probability distribution. The ultimate goal is to find the hypothesis $h$ that makes the fewest prediction errors on new examples, thereby minimizing the generalization error. However, as the probability distribution $\mathcal{P}_\mathcal{T}$ is typically unknown, direct estimation of the generalization error is not feasible. To overcome this challenge, [Vap00] proposed a method to search for the optimal hypothesis $h$ by optimizing its empirical risk on a training set $\mathcal{D}$. The empirical risk is an unbiased estimator of the true risk and is commonly referred to as the *empirical risk* of $h$ on $\mathcal{D}$, defined in Definition 2.3.

**Definition 2.3.** Empirical risk (training error). *Given a loss function $\ell : \mathcal{H} \times \mathcal{Z} \to \mathbb{R}^+$ and a set of examples $\mathcal{D}$, the empirical risk $\hat{R}_\mathcal{D}(h)$ of a hypothesis $h$ is defined as:*

$$\hat{R}_\mathcal{D}(h) = \frac{1}{n} + \sum_{\mathbf{z} \in \mathcal{D}} \ell(h, \mathbf{z}) \tag{2.4}$$

### 2.2.1.6 Empirical Risk Minimization

As stated earlier, a learning algorithm $\mathcal{A}$ is provided with a training set $\mathcal{D}$, which is sampled from an unknown distribution $\mathcal{P}_\mathcal{T}$. The algorithm's objective is to produce a model or hypothesis $h : \mathcal{X} \to \mathcal{Y}$ that minimizes the error when compared to the unknown $\mathcal{P}_\mathcal{T}$. Given the algorithm's lack of knowledge regarding $\mathcal{P}_\mathcal{T}$, the true risk in Equation (2.3) remains inaccessible to $\mathcal{A}$. Hence, the concept underlying *Empirical Risk Minimization* involves the selection of the optimal hypothesis $h^\star$ by minimizing the empirical risk computed on the training set $\mathcal{D}$. This objective entails solving the following *optimization* problem:

$$h^\star = \operatorname*{argmin}_{h \in \mathcal{H}} \hat{R}_\mathcal{D}(h) \tag{2.5}$$

$$h^\star = \operatorname*{argmin}_{h \in \mathcal{H}} \frac{1}{n} + \sum_{i=1}^{n} \ell(h, (\mathbf{x}_i, y_i)) \tag{2.6}$$

Equation (2.6) presents the procedure for determining the optimal hypothesis, denoted as $h^\star$, within the hypothesis space $\mathcal{H}$. Solving the optimization problem in Equation (2.6) provides two key outcomes. Firstly, the optimal hypothesis, $h^\star$, serves as a hypothesis that performs optimally on the training set $\mathcal{D}$. Secondly, the corresponding empirical risk $\hat{R}_{\mathcal{D}}(h)$, or training error, can be utilized to estimate the true risk or expected loss associated with $h^\star$. However, it is important to acknowledge that the training error $\hat{R}_{\mathcal{D}}(h)$ obtained for $\mathcal{D}$ may significantly differ from the expected loss of $h^\star$ when applied to new datapoints not included in $\mathcal{D}$. The assumption of independent and identically distributed data implies that the training error $\hat{R}_{\mathcal{D}}(h)$ only provides a noisy approximation of the true risk $R_{\mathcal{T}}(h)$. Thus, even if the hypothesis $h^\star$ derived through ERM has a small training error, it may still possess an unacceptably large true risk $R_{\mathcal{T}}(h)$. In the pursuit of a model with minimal true risk, it is crucial to address the risks of overfitting and underfitting. Overfitting arises when the function class $\mathcal{H}$ is excessively large, causing the empirical risk minimizer (e.g., an optimization algorithm like SGD) to capture irrelevant patterns in the training data. Although this may yield low empirical risk, it often leads to poor accuracy when dealing with new examples, resulting in high true risk. Conversely, *underfitting* occurs when $\mathcal{H}$ is too small, causing all models within $\mathcal{H}$ to underperform on the training data and leading to *high* empirical risk [BHMM19]. Both overfitting and underfitting are undesirable outcomes as we strive for a model with minimal true risk. In Section 2.2.1.9, we will delve into a detailed exploration of these challenges.

#### 2.2.1.7 Optimization

In ML, when utilizing a parameterized hypothesis denoted by $h_{\mathbf{\Theta}}(\mathbf{x})$, where $\mathbf{\Theta}$ are the model parameters that are learned from $\mathcal{D}$, the optimization problem in Equation (2.6) can be reformulated as an optimization of the model parameters, symbolized as $\hat{\mathbf{\Theta}}$:

$$\hat{\mathbf{\Theta}} = \operatorname*{argmin}_{\mathbf{\Theta} \in \mathbb{R}^n} h(\mathbf{\Theta}) \text{ with } h(\mathbf{\Theta}) := \frac{1}{n} + \sum_{i=1}^{n} \ell(h_{\mathbf{\Theta}}, (\mathbf{x}_i, y_i)) \tag{2.7}$$

The function $h(\mathbf{\Theta})$ given in Equation (2.7) represents the empirical risk $\hat{R}_{\mathcal{D}}(h_{\mathbf{\Theta}})$ acquired from applying the hypothesis $h_{\mathbf{\Theta}}$ to the datapoints in the dataset $\mathcal{D}$. The optimization problems stated in Equations (2.7) and (2.6) are completely equivalent. By obtaining the optimal parameter vector $\hat{\boldsymbol{\Theta}}$ that solves Equation (2.7), the hypothesis $h_{\boldsymbol{\Theta}}$ effectively solves Equation (2.6). The Stochastic Gradient Descent (SGD) algorithm [BB08], is utilized to find the optimal parameters $\hat{\boldsymbol{\Theta}}$ that minimize the empirical risk function. This iterative optimization method is effective in minimizing a function by evaluating its partial derivatives at different points. To illustrate the method, we begin with the context of a scalar function that depends on a parameter vector denoted by $\boldsymbol{\Theta}$. Let us consider a scenario where the function $h(\boldsymbol{\Theta})$ depends on a parameter vector $\boldsymbol{\Theta}$. In this case, the partial derivative or gradient of $h(\boldsymbol{\Theta})$ with respect to $\boldsymbol{\Theta}$, denoted as $\nabla h(\boldsymbol{\Theta})$, provides crucial information about the *slope* of the function at a given point in the parameter space. Just as the slope guides us in descending a hill, the gradient guides us towards the optimal point $\hat{\boldsymbol{\Theta}}$ that minimizes $h(\boldsymbol{\Theta})$. The gradient is formally defined as the vector comprising the partial derivatives of $h(\boldsymbol{\Theta})$ with respect to all components $\Theta_i$ of the parameter vector $\boldsymbol{\Theta}$:

$$\nabla_{\boldsymbol{\Theta}} h(\boldsymbol{\Theta}) \triangleq \frac{\partial h(\boldsymbol{\Theta})}{\partial \boldsymbol{\Theta}} = \begin{bmatrix} \frac{\partial h(\boldsymbol{\Theta})}{\partial \Theta_1} \\ \frac{\partial h(\boldsymbol{\Theta})}{\partial \Theta_2} \\ \vdots \\ \frac{\partial h(\boldsymbol{\Theta})}{\partial \Theta_m} \end{bmatrix} \tag{2.8}$$

By following the gradient, we can adjust each component of the parameter vector to approach the minimum of the function. The SGD algorithm, in its simplest form, starts with a random initial parameter vector $\boldsymbol{\Theta}_0$. It then iteratively adjusts the parameter vector $\boldsymbol{\Theta}$ by moving in the opposite direction of the gradient, scaled by a learning rate, towards the minimum of the function. Each iteration brings us closer to the optimal set of parameters $\hat{\boldsymbol{\Theta}}$ that minimizes the empirical risk function $\hat{R}_{\mathcal{D}}(h_{\boldsymbol{\Theta}})$. To visualize this, le us imagine being blindfolded on a multidimensional terrain. The gradient represents the direction of the steepest descent, indicating the safest path towards the lowest point in the landscape. By following the negative gradient, we update each component of the parameter vector in a way that reduces the loss function and brings us closer to the optimal solution. The function $\delta$ in Algorithm 1 represents the composite function that combines the loss function $\ell$ and the parameterized hypothesis $h_{\boldsymbol{\Theta}}$. The composite function is denoted as $\delta = \hat{R}_{\mathcal{D}}(h_{\boldsymbol{\Theta}})$. By utilizing this composite function, the algorithm emphasizes its dependence on the empirical risk minimization problem, as discussed in the work by Bottou [BCN18].

To start the algorithm, we input the training data $\mathcal{D}$, which comprises $n$ sample pairs $(\mathbf{x}_i, y_i)$. Additionally, we specify the learning rate $\gamma$, which determines the step size for

**Algorithm 1:** Stochastic Gradient Descent

```
Input: Training data $\mathcal{D} = \{\mathbf{z}_i = (\mathbf{x}_i, y_i)\}_{i=1}^n$, learning rate $\gamma$, and the number of
       iterations $K$.
Output: Optimal parameter vector $\hat{\boldsymbol{\Theta}}$.
Initialize $\boldsymbol{\Theta}_0$ randomly
for $k = 0, 1, 2, \ldots, K-1$ do
    Randomly choose an index $i_k$ from $\{1, \ldots, n\}$
    Compute the gradient $\nabla \delta_{i_k}(\boldsymbol{\Theta}_k)$ on the training instance $(\mathbf{x}_{i_k}, y_{i_k})$
    Update the parameter vector $\boldsymbol{\Theta}_{k+1} \leftarrow \boldsymbol{\Theta}_k - \gamma_k \nabla \delta_{i_k}(\boldsymbol{\Theta}_k)$
end for
$\hat{\boldsymbol{\Theta}} \leftarrow \boldsymbol{\Theta}_K$
```

updating the parameter vector, and the number of iterations $K$, that the algorithm will perform. Next, we initialize the parameter vector $\boldsymbol{\Theta}_0$ with random values. This initial vector serves as the starting point for the optimization process. The algorithm then enters a loop where it iterates from 0 to $K-1$. In each iteration, the index $i_k$ is chosen *randomly* from $\{1, \ldots, n\}$. This random selection ensures that each sample has an equal chance of being selected, introducing *stochasticity* in the algorithm. Using the randomly selected index $i_k$, the algorithm computes the gradient $\nabla \delta_{i_k}(\boldsymbol{\Theta}_k)$ of the loss function with respect to the current parameter vector $\boldsymbol{\Theta}_k$. This gradient represents the direction of steepest descent in the loss function space, indicating how the loss function changes with respect to each parameter. The parameter vector is then updated by subtracting the learning rate multiplied by the gradient from the current parameter vector. This update step, $\boldsymbol{\Theta}_{k+1} \leftarrow \boldsymbol{\Theta}_k - \gamma_k \nabla \delta_{i_k}(\boldsymbol{\Theta}_k)$, moves the parameter vector in the direction that reduces the loss. By iteratively adjusting the parameters based on the gradients, the algorithm aims to find the optimal set of parameters that minimize the empirical risk on the given training data. Finally, after completing all iterations, the algorithm returns the final parameter vector $\hat{\boldsymbol{\Theta}}$, which represents the solution obtained by the SGD algorithm. This vector represents the *optimal set of parameters that minimize the loss function based on the given training data and learning rate.*

#### 2.2.1.8 Model selection

It is essential to note that relying solely on the ERM is inadequate for model selection. To obtain a more precise estimation of the true risk of a learning algorithm's model, we often require additional steps. One of these steps is to use a portion of the training data as a validation set to evaluate the algorithm's success. This process is referred to as *validation.* Sampling an additional set of examples independent of the training set is the simplest way to estimate the true risk of the model [SSBD14]. By doing so, we can *use the empirical risk on this validation set as our estimator.* To formalize this approach, let us consider a set of new examples $\mathcal{D}_{\text{VAL}} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{n_{\text{VAL}}}, y_{n_{\text{VAL}}})$ sampled according to $\mathcal{P}_{\mathcal{T}}$ (independently of the $n$ examples of the training set $\mathcal{D}$). The validation process can aid in model selection by training various algorithms, or the same algorithm with different parameters, on the given training set. Let $\mathcal{H} = h_1, \ldots, h_r$ be the collection of all output models generated by the different algorithms.

To illustrate this, let us consider a polynomial regression model of degree 1, which is equivalent to a linear regression model represented as $\hat{y} = wx + b$. We can extend this model to include $x^2$ as an additional feature to learn a quadratic function of $x$, which can be represented as $\hat{y} = b + w_1 x + w_2 x^2$. Although this quadratic model incorporates a quadratic feature ($x^2$), the output value still depends linearly on the model parameters ($w_1$, $w_2$ and $b$). In other words, the model is still a linear combination of the input features and their powers, with the parameters determining the weights of each feature. Thus, we can train this model in closed form[2] using the normal equations. To obtain an even more complex polynomial model of degree 10, we can add additional powers of $x$ as extra features. In this scenario, each $h_r$ would represent the output of polynomial regression of degree $r$. Then, we select a single model from $\mathcal{H}$ that *minimizes the error over the validation set using ERM.*

#### 2.2.1.9 Complexity, overfitting and underfitting

The effectiveness of a ML algorithm hinges on two critical factors: its complexity to minimize the empirical risk and narrowing the gap between empirical and true risk, namely, $\hat{R}_{\mathcal{D}}(h) - R_{\mathcal{T}}(h)$, should be small over all $h \in \mathcal{H}$ [GBC16]. These factors are the two fundamental challenges in ML known as underfitting and overfitting. In the case of *underfitting*, it occurs when the model fails to achieve a sufficiently low error value on the training set, while *overfitting* arises when the difference between empirical risk and true risk is too large. The *complexity* of a model can determine the likelihood of underfitting or overfitting, which refers to its capability to fit a broad range of functions. Models with low complexity may struggle to fit the training set, while models with high complexity can overfit by memorizing characteristics of the training set that are not representative of the validation set. Controlling the complexity of a learning algorithm is possible by selecting

[2]A closed-form solution is one where a mathematical formula can be directly computed to obtain the solution without the need for iterative or numerical methods.

its hypothesis space $\mathcal{H}$. This relates to the set of functions that the learning algorithm can choose as a solution. As an example, the hypothesis space of the linear regression algorithm includes all linear functions of its input. However, this can be extended by incorporating polynomials into the hypothesis space to enhance the model's complexity. This approach enables the learning algorithm to capture more complex relationships between the input and output variables. The performance of ML algorithms depends on their ability to capture the complexity of a task, as well as the size of the training dataset. If the model's complexity is too low, it will struggle to solve complex problems, while a model that is too complex may overfit the data. To illustrate this point, let us consider the plot in Figure 2.3, which shows the results of fitting a quadratic function using three different models: a linear model, a quadratic model, and a model with degree-10 polynomial basis functions.

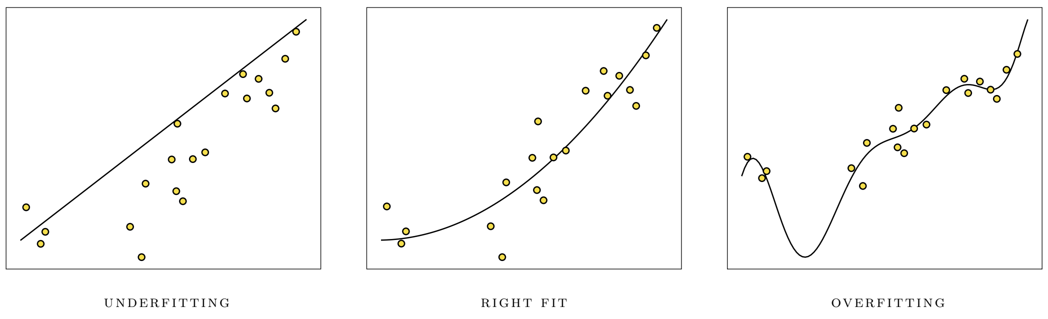


FIGURE 2.3: Illustration of the overfitting and underfitting phenomena.

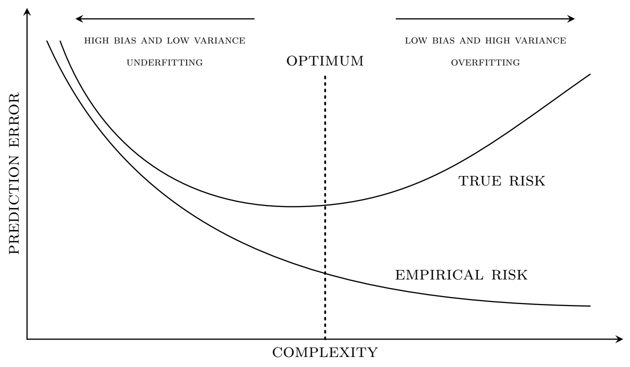


FIGURE 2.4: Graph depicting the trade-off between true and empirical risk as a function of model complexity.

The linear model is too simple to capture the curvature of the underlying quadratic function, resulting in underfitting. The model with degree-10 polynomial basis functions can perfectly fit the training data but may overfit by capturing noise or patterns that do not generalize to new data. In contrast, the quadratic model achieves an appropriate balance by accurately representing the true structure of the task and generalizing well to new data. The *model-selection curve*, depicted in Figure 2.4, displays the typical trend of the true risk and empirical risk as the model complexity varies. As the gap continues to grow, it surpasses the decrease in empirical risk, and we transition into the overfitting phase, where complexity exceeds the optimal level. The optimal model complexity balances the trade-off between overfitting and underfitting, ensuring better generalization and accurate predictions [SSBD14]. The interested reader may refer to Appendix B.1 for regression and Appendix B.2 for classification to further explore these topics. Furthermore, to provide technical insights on the origin of prediction error and the bias-variance trade-off, we present detailed information in Appendix B.3 and B.4, respectively.

## 2.3 Recommender systems

### 2.3.1 Machine Learning in RS

In the recommendation problem, the ML system acquires knowledge by improving its ability to suggest relevant products or services to users. It does this by *learning* from users' historical interactions, specifically, their past ratings of items. The goal is to predict *missing* user-item preferences, namely, the user's preference for items they have not yet rated, based on *observed* user-item preferences. We approach this prediction problem as a *supervised* ML problem. Our objective is to minimize the expected loss (see Equation 2.13) of our preference prediction model by *fitting* previously *observed ratings* using a suitable error or loss function [KB11].

Upon applying Definition 2.1 to our example, we can identify that task $T$ refers to recommending items. Performance metric $P$, which represents the accuracy of the system's recommendations, is defined as the ratio of correctly recommended items to the total number of recommendations. The experience $E$, on the other hand, refers to historical data on user item preferences, including information on highly rated items, purchases and searches. It is essential for the ML system to learn effectively that experience $E$ must be both: *relevant* to task $T$ and of high quality. To present users with relevant items, a data-driven approach is required to determine item relevance in each context. This involves several phases, starting with data collection, followed by model development and ending with prediction and recommendation. Figure 2.5 illustrates the flowchart of these three phases.

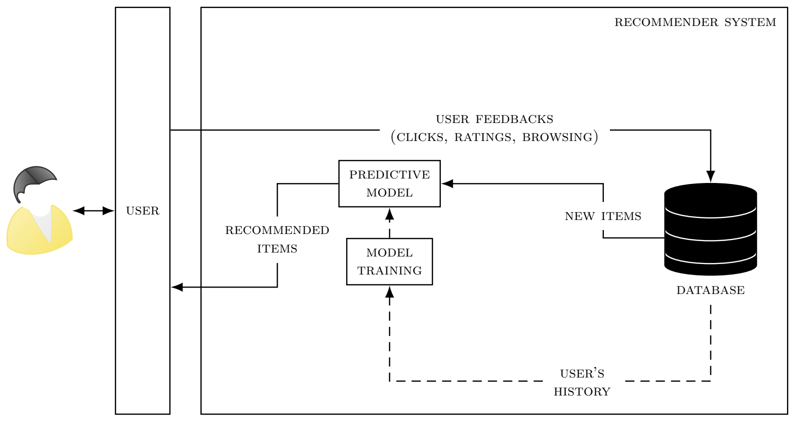


Figure 2.5: High-level architecture of a RS.

In the data collection phase, user interactions with the system and items are collected to build the RS. The system automatically infers user preferences by monitoring their actions, such as purchase and browsing history, time spent on web pages, email content and button clicks. These actions provide explicit or implicit feedback on user tastes, with explicit feedback being explicitly specified by users and implicit feedback assigned automatically based on user-item interaction. The model development phase involves using ML algorithms, such as collaborative filtering, content-based filtering and hybrid techniques, to filter out and use user characteristics from the feedback collected in the previous phase. The objective is to develop a model that accurately predicts user preferences. Finally, in the prediction and recommendation phase, the model is used to recommend relevant items to users. The recommendation algorithm employs the user's model to predict the likelihood of an item being relevant to the user. The recommendation can be in the form of a ranked list, a set of suggested items or personalized content. User interaction with the recommended items provides feedback, which is used to improve the model and enhance the accuracy of future recommendations.

### 2.3.2 Taxonomy of RS

This section presents a taxonomy of RS based on three main filtering techniques: content-based filtering, collaborative filtering and knowledge-based filtering. The taxonomy examines the features and limitations of each method and provides practical guidance for selecting the appropriate technique for different applications. The goal is to provide the reader with a clear understanding of the various filtering methods, as well as their strengths and limitations. Figure 2.6 provides an overview of the different approaches used in RS.

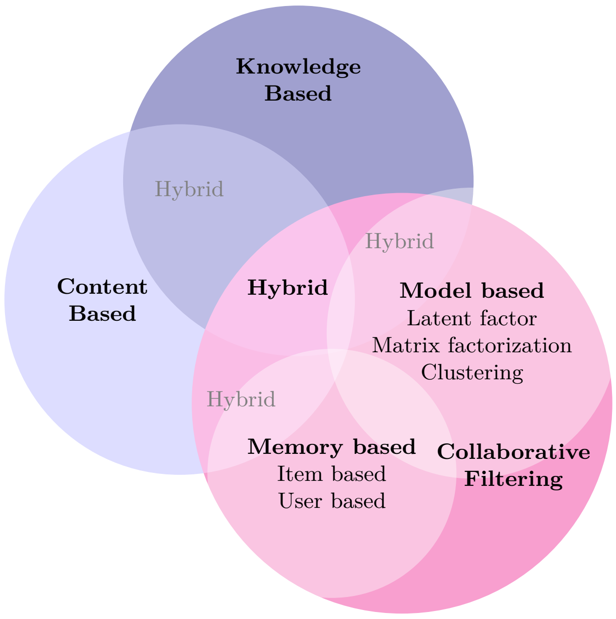


FIGURE 2.6: RS approaches with an emphasis on collaborative filtering methods.

#### 2.3.2.1 Content-based RS

Content-based filtering is a technique used to recommend items based on their content. This method, introduced by [PB07] and further explored by [LDS11], discriminates relevant items by analyzing their characteristics. The approach recommends similar items to those that the user has shown interest in the past. Although many of these approaches concentrate on textual content, any set of features can be considered. This means that new items can be recommended based on their characteristics, even if they have not been previously rated. However, this early filtering approach has certain limitations as it tends to keep the user within their historical data, limiting the recommendation of items that are too different from their usual preferences.

#### 2.3.2.2 Collaborative filtering

Collaborative filtering is a technique that infers the preferences of a user by analyzing the ratings they give to different items. It models the social environment of users to make recommendations based on the behavior of their peers. Formally, given a user

A, collaborative filtering identifies other users who exhibit similar rating behavior and computes the score of an item B for user A utilizing the average score given to B by similar users. This approach predicts user preferences based solely on the ratings given to items in the past. We view user ratings of items as a collaborative process, in which users help each other to discover interesting items, even if they are not consciously collaborating. Thus, the approach is known as collaborative filtering [AC16].

Collaborative filtering involves two-dimensional data in matrix form, with one dimension representing the list of users and the other representing the items they like. The most common approach is to model this environment utilizing historical transactions stored in the system, such as user movie ratings. Users are considered peers if their ratings are similar. If users A and B have similar ratings for a certain number of movies, and if A has not seen a movie that B has liked, then the system recommends it to A. Collaborative filtering is content-independent, as it does not require any knowledge about the items themselves. Additionally, these techniques are serendipitous in nature, as recommendations are based solely on user behavior and do not require knowledge of item characteristics or explicit input from users. However, collaborative filtering suffers from the cold start problem (see Section 2.3.3).

#### 2.3.2.3 Knowledge-based RS

Knowledge-based RS are a type of RS that focuses on addressing the limitations of data availability and customization challenges in certain domains. Unlike collaborative and content-based systems, knowledge-based systems allow users to explicitly specify their requirements for items. They leverage interactive feedback and knowledge bases to provide personalized recommendations. These systems are well-suited for domains where users actively express their preferences and where historical data may be scarce or inadequate. By incorporating domain-specific knowledge and facilitating user exploration of complex product spaces, knowledge-based RS offer greater control and customization, resulting in more tailored recommendations [Agg16].

### 2.3.3 Challenges and limitations

The field of RS has progressed significantly with technological advancements in recent years. However, despite this progress, there remain several challenges and limitations that must be addressed. In this section, we describe the primary limitations of collaborative filtering based systems in RS.

#### 2.3.3.1 Cold start

RS encounter the cold start problem when a new user or item lacks sufficient ratings, rendering it difficult for the system to provide reliable recommendations. This problem can be divided into two categories: the cold start of new users and the cold start of new items. The former arises when a new user requests recommendations before expressing any preferences on any item. The latter arises when a new item is added to the catalog, but none of the users have rated it yet [KEK18].

#### 2.3.3.2 Data sparsity

A significant quantity to consider is the density of the $m \times n$ matrix $\mathbf{R}$ (see Figure 2.7). This density can be measured by:

$$density(\mathbf{R}) = \frac{|\mathcal{D}|}{mn}$$

or equivalently, by its complement $1 - density(\mathbf{R})$, which represents the sparsity. The higher the number of missing entries, the greater the sparsity, making it more challenging to learn user preferences. Since users tend to rate only a few items, the matrix $\mathbf{R}$ is generally very sparse. Consequently, it becomes essential to prevent overfitting, which refers to achieving good performance on past training data but failing to generalize to unobserved data. However, due to computational complexity considerations, most prediction methods are incapable of handling large dense matrices. Exploiting the sparsity, particularly with methods that scale efficiently with $|\mathcal{D}| \ll mn$ becomes advantageous computationally. Here, $\mathcal{D}$ represents the dataset of ratings, as defined in Equation (2.9).

#### 2.3.3.3 Scalability

The growth rate of $k$-nearest neighbors algorithms shows a linear relationship with the number of items and users. Consequently, as the number of users and items increases, the system requires more resources to provide accurate recommendations [KAU16]. The identification of users and items with similar attributes and preferences consumes a significant portion of these resources. Hence, using recommendation algorithms that can scale effectively with the increasing data volume is essential for improving the performance of the system.

#### 2.3.3.4 Limitations of IID assumptions in traditional recommendation approaches

The current state of the art in recommendation research is based on the assumption that users, items and ratings are independently and identically distributed (*i.i.d.*), resulting in the development of *i.i.d.* models and methods. However, this approach overlooks low-level non-*i.i.d.* information about individual users and items that can significantly impact the driving forces of ratings. This oversight is a critical factor contributing to the poor performance of existing RS [Cao16]. Current RS focus solely, as an example, on the rating information provided in Figure 2.8 and fail to consider the underlying reasons behind specific user preferences. Memory based and model based methods assume that users and items are *i.i.d.*, neglecting connections between users and items, as well as influence and heterogeneity between them. Memory-based methods assume that users rate all items independently, ignoring the relationship between the ratings of different items or the connections between users and their ratings. In contrast, model-based methods, such as matrix factorization, assume that the rating dynamics are driven solely by the user and item latent factors, and do not consider user and item properties. However, these assumptions limit the ability of both approaches to provide personalized recommendations that address individual preferences. Therefore, developing more sophisticated recommendation algorithms that consider the driving factors of user and item attributes is necessary to provide more accurate and personalized recommendations.

### 2.3.4 Recommendation as risk minimization

To tackle the recommendation task, we view it as a function regression problem guided by the ERM principle, as outlined in Section 2.2.1.5. This approach emphasizes the need to minimize the discrepancy between the predicted and actual output, which can lead to improved accuracy in the recommended outcomes.

#### 2.3.4.1 Problem formalization and notation

To provide a formal definition of the recommendation task, we introduce the following notation. Let $U$ represent the set of users and $I$ denote the set of items. An individual user is denoted by $u \in U$, an individual item is denoted by $i \in I$ and a specific rating, given by user $u$ for item $i$, is denoted as $r_{ui}$. It should be noted that the ratings are treated as ordinal values and are not necessarily numerical in nature. We assume the existence of a total order among the possible rating values, where the ratings can encompass a range from 1 star (indicating low interest) to 5 stars (indicating high interest), for instance. To capture the number of distinct rating values, denoted by $\mathcal{R}$, we represent the ratings themselves as $1, 2, \ldots, \mathcal{R}$. The rating matrix $\mathbf{R} \in \mathbb{R}^{|U|\times|I|}$, depicted in Figure 2.7, is a sparse matrix where each element $r_{ui}$ corresponds to the rating given by user $u$ to item $i$. This matrix can potentially have $|U| \times |I|$ ratings, where $|U|$ and $|I|$ represent the number of users and items, respectively. We denote $\mathbf{r}_i$ as the vector that contains the ratings given to item $i$ by different users, and $\mathbf{r}_u$ represents the vector containing the ratings provided by user $u$ for different items.

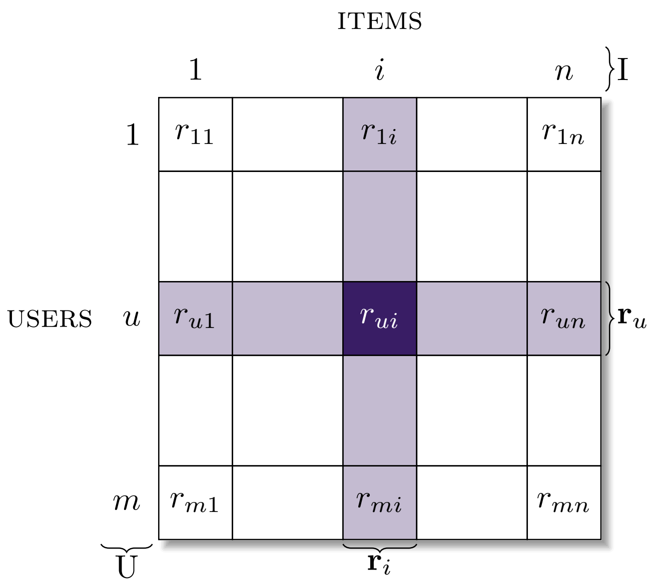


FIGURE 2.7: Representation of users' ratings for all items as the rating matrix.

The subset of items that user $u$ has rated with $r \neq \varnothing$ is named $I_u \subset I$ and $I_{uv} \overset{\text{def}}{=} I_u \cap I_v$ is the subset of items that have been rated by users $u$ and $v$. The subset of users who have rated item $i$ is named $U_i \subset U$ and $U_{ij} \overset{\text{def}}{=} U_i \cap U_j$ is the subset of users who have rated to the items $i$ and $j$. The set of all observed user-item pairs with ratings, denoted by $\mathcal{D}$, is defined as the set of all pairs $(u, i)$ for which $r_{ui}$ is observed in the rating matrix. Thus, $\mathcal{D}$ represents the observed historical ratings in the system. Formally, we define $\mathcal{D}$ as:

$$\mathcal{D} = \{(u, i) \mid r_{ui} \in \mathbf{R}\} \tag{2.9}$$

Then, the recommendation phase can be summarized as follows: utilizing a designated recommendation algorithm, denoted as $\mathcal{A}$, the objective is to estimate the ratings $r_{ui}$ that are initially unknown (i.e., $r_{ui} \notin \mathcal{D}$). This estimation, denoted as $\hat{r}_{ui}$, is derived from the application of $\mathcal{A}(u,i)$. In a broader context, it can be stated that the RS computes an $\hat{\mathbf{R}} \in \mathbb{R}^{|U|\times|I|}$ matrix, which encompasses predictions of $\hat{r}_{ui}$ ratings for all possible combinations of users and items. Building upon this notation, Definition (2.4) formally defines a RS:

**Definition 2.4.** (Recommender system) [AT05]. Let $f$ be a function that evaluates the utility of the item $i$ for the user $u$, i.e.

$$f : U \times I \rightarrow \mathcal{R} \tag{2.10}$$

where $\mathcal{R}$ is a *totally ordered set* such as a 5-star rating scale or a range of real numbers. Hence, a RS is defined as a system that selects a set of top-$k$ items, denoted as $i' \in I$, with the objective of maximizing the utility for a given user $u \in U$, i.e.

$$\forall u \in U, i' = \operatorname*{argmax}_{i \in I} f(u,i) \tag{2.11}$$

where $i' \in I \setminus I_u^+$, and $I_u^+$ is the set of positive items already seen by user $u$.

One of the primary challenges in RS lies in the fact that the utility value of each item $i \in I$ is often unknown. This information is only available for items that the user has previously rated in their history. Consequently, the objective of a RS is to predict the utility function value for unknown data. Once the utility function parameters are learned, the system can generate predicted scores for items that the user has not yet rated. This learned utility function, denoted as $f$, utilizes a set of training examples from $\mathcal{D}$ to recommend items $i$ to a user $u$. To evaluate the accuracy of $f$, a validation set is used, which is a subset of $\mathcal{D}$ that is disjoint from the training set. The root-mean-square error (RMSE), defined in Equation (2.12), is a popular and widely used measure for assessing the accuracy of $f$ on a set $\mathcal{D}$ of ratings:

$$\text{RMSE}(f \mid \mathcal{D}) = \sqrt{\frac{1}{|\mathcal{D}|} \sum_{r_{ui} \in \mathcal{D}} (r_{ui} - \hat{r}_{ui})^2} \tag{2.12}$$

where $|\mathcal{D}|$ represents the number of items rated by user $u$ in the training set. To compute the RMSE, we sum over all user-item pairs $(u,i)$ for which $r_{ui} \in \mathcal{D}$. Minimizing the RMSE is a common approach for *learning* $f$, as it can be justified by the inductive principle of ERM. We can describe the problem of learning a utility function assuming that:

- The user-item pairs $(u, i)$ are drawn from an unknown probability distribution $\mathcal{P}(u, i)$.
- Rating scores $r_{ui} \in \mathcal{R}$ are provided for each user-item pair $(u, i)$ according to an unknown conditional probability distribution $\mathcal{P}(r_{ui} \mid u, i)$.
- The class $\mathcal{F}$ represents the set of utility functions.

The probability distribution $\mathcal{P}(u, i)$ represents the likelihood that a user $u$ will rate an item $i$, while the conditional probability distribution $\mathcal{P}(r_{ui} \mid u, i)$ represents the probability that a given user $u$ will rate a given item $i$ with a rating score of $r_{ui}$. The class $\mathcal{F}$ refers to the set of functions from which we choose our function $f$ for recommending items. The objective of acquiring a utility function is to identify a function $f \in \mathcal{F}$ that can minimize the true risk function:

$$R(f) = \sum_{u,i,r_{ui}} \mathcal{P}(u, i, r_{ui})(f(u, i) - r_{ui})^2 \tag{2.13}$$

The goal is to learn the optimal utility function, denoted as $f^\star$, for a given set of users $U$, items $I$ and rating scale $\mathcal{R}$. This involves finding the function that minimizes $R(f)$, where the sum runs over all possible triples $(u, i, r_{ui}) \in U \times I \times \mathcal{R}$, and $\mathcal{P}(u, i, r_{ui}) = \mathcal{P}(u, i)\mathcal{P}(r_{ui} \mid u, i)$ is the joint probability [SB18]. However, the distribution $\mathcal{P}(u, i, r_{ui})$ is unknown, which makes it impossible to compute $f^\star$ directly. Instead, we aim to approximate $f^\star$ by minimizing the empirical risk as shown in Equation (2.14).

$$\begin{aligned} f^\star &= \operatorname*{argmin}_{f \in \mathcal{F}} \hat{R}_{\mathcal{D}}(f) \\ f^\star &= \operatorname*{argmin}_{f \in \mathcal{F}} \frac{1}{|\mathcal{D}|} \sum_{r_{ui} \in \mathcal{D}} (f(u, i) - r_{ui})^2 \end{aligned} \tag{2.14}$$

The empirical risk can be minimized by minimizing the RMSE. This principle states that the empirical risk provides an unbiased estimate of the true risk, under the assumption that user ratings from $\mathcal{D}$ are *i.i.d.* Furthermore, given that the user and item sets are finite, we can apply the Law of Large Numbers by considering $\mathcal{D}$ as a set of ratings obtained by randomly selecting triples $(u, i, r_{ui})$ based on their joint distribution.

As discussed earlier in Section 2.2.1.7, methods that use a parameterized model or function $f$, typically involve an optimization process to determine the optimal values of the parameters. In model-based RS the primary objective is to learn the unknown parameters $\boldsymbol{\Theta}$

for a prediction function $f_{\mathbf{\Theta}}(u, i)$ capable of accurately generalizing predictions to future instances. Consequently, the optimization problem in Equation (2.14) can be redefined as the task of optimizing the parameter vector $\hat{\mathbf{\Theta}}$:

$$\hat{\mathbf{\Theta}} = \underset{\mathbf{\Theta}}{\operatorname{argmin}} f_{\mathbf{\Theta}}(u, i) \tag{2.15}$$

The function $f_{\mathbf{\Theta}}(u, i)$ in Equation (2.15) represents the empirical risk, denoted as $\hat{R}_{\mathcal{D}}(f_{\mathbf{\Theta}})$, which is the loss incurred by the model $f_{\mathbf{\Theta}}(u, i)$ when applied to the ratings in the dataset $\mathcal{D}$. It is worth noting that the optimization problems presented in Equations (2.14) and (2.15) are completely equivalent. In other words, the optimal weight vector $\hat{\mathbf{\Theta}}$ that solves Equation (2.15) also leads to the solution of Equation (2.14) through the model $f_{\hat{\mathbf{\Theta}}}(u, i)$.

Two primary approaches, *memory-based methods* and *model-based methods*, constitute the field of RS. Memory-based methods, also referred to as neighborhood-based methods, directly exploit similarities among users or items in the rating matrix to generate accurate predictions. Conversely, model-based methods employ an offline process to extract and aggregate information from the rating matrix, constructing a model that is subsequently utilized to generate relevant predictions. In memory-based methods, the model is a function $f(u, i)$ that relies on the complete set of observed data, represented by $u$ and $i$. Conversely, model-based methods employ a parameterized model $f_{\mathbf{\Theta}}(u, i)$ with an unknown parameter vector $\mathbf{\Theta}$. These two main approaches will be extensively explored in Chapter 3.

Following Mitchell's Definition (2.1), we can now redefine the three characteristics for RS incorporating the mathematical formal notation used previously:

- Task $T$: Predict a rating within the range of $\mathcal{R}$ for user $u$ and item $i \in I$.
- Performance $P$: Calculate the percentage of correctly predicted ratings for items $i \in I_u$ by user $u$.
- Experience $E$: User $u$'s item rating vector $\mathbf{r}_i$ and the set of items $I_u$.

Thus, the primary task assigned to the model is to infer the rating $r \in \mathcal{R}$ corresponding to the item $i \in I$. Consequently, the goal is to identify the function $f$ from the class $\mathcal{F}$, encompassing all potential functions that yields the most precise prediction for the rating $r_{ui}$ associated with item $i$ by user $u$. The effectiveness of the function $f$ is evaluated using a performance measure to determine the optimal function $f^{\star}$. The performance of $f$ relies on several factors, including the selection of the class $\mathcal{F}$, which is determined by the discretion of the human designer, as well as the accumulated experience and the number of ratings provided by user $u$.

#### 2.3.4.2 The two phases of RS: Prediction and ranking

The recommendation task can typically be defined in two different ways. The first is the **rating prediction** task, which aims to estimate item ratings for a user. The second is **learning to rank**, which involves recommending a shortlist of top-$k$ items to users. These two tasks have different optimization goals. The rating prediction task seeks to maximize prediction accuracy, while the learning to rank task is focused on generating the best list to present to the user. Although ratings are important in both tasks, there is a significant difference between them. In a rating prediction task, the value associated with the user-item pair is an estimate of the rating. Conversely, in a top-$k$ task, the same value is only used to sort the recommendation list, ordered from the most relevant to the least relevant. Essentially, this value is an estimation of a value associated with the position of the item and has no direct relation to the corresponding rating. The algorithm for the ranking task is presented in Algorithm 2.

**Algorithm 2:** Ranking task

**Input:** Set of users $U$, set of items $I$, function $f(u,i)$ which predicts the utility score of item $i$ for user $u$, and the number of recommended items $k$.
**Output:** Top-$k$ recommendation list for each user.

```
foreach user $u \in U$ do
    foreach not-interacted item $i \in I \setminus I_u^+$ do
        $n \leftarrow 0$ // Reset the counter for each item
        $\hat{r}_{ui} \leftarrow f(u,i)$ // Compute the utility score of item i for user u
        foreach not-interacted item $j \in I \setminus I_u^+$ do
            $\hat{r}_{uj} \leftarrow f(u,j)$ // Compute the utility score of item j for user u
            if $\hat{r}_{uj} \geq \hat{r}_{ui}$ then
                $n \leftarrow n + 1$ // Increment counter
            end if
        end foreach
        $\mathcal{O}_{ui} \leftarrow n + 1$ // Assign rank to item i
    end foreach
    Sort the items in $I \setminus I_u^+$ in descending order of their rank $\mathcal{O}_{ui}$
    Select the top-$k$ items from the sorted list to recommend to user $u$
end foreach
```

Given the set of users $U$, set of items $I$, a function $f(u, i)$ that predicts the utility score of item $i$ for user $u$ and the number of recommended items $k$, Algorithm 2 aims to generate a top-$k$ recommendation list for each user. For each user $u$ in the set of users $U$, the algorithm iterates over the not-interacted items $i$ in the set $I \setminus I_u^+$ (items that the user has not interacted with). For each not-interacted item $i$, the algorithm initializes a counter $n$ to 0. It computes the utility score $\hat{r}_{ui}$ of item $i$ for user $u$ using $f(u, i)$. Then, the algorithm iterates over the not-interacted items $j$ in $I \setminus I_u^+$ and computes the utility score $\hat{r}_{uj}$ of item $j$ for user $u$ using the same function $f(u, j)$. If $\hat{r}_{uj}$ is greater than or equal to $\hat{r}ui$, the counter $n$ is incremented. After processing all not-interacted items the algorithm assigns the rank $\mathcal{O}_{ui}$ to item $i$ as $n + 1$. The rank represents the relative position of item $i$ in terms of utility score compared to other not-interacted items. Next, the algorithm sorts the not-interacted items in $I \setminus I_u^+$ in descending order based on their ranks $\mathcal{O}_{ui}$. Finally, the algorithm selects the top-$k$ items from the sorted list as the recommendation list for user $u$. By applying this algorithm to each user in $U$ we can generate personalized top-$k$ recommendation lists based on their utility scores for the not-interacted items. A complete numerical example demonstrating the application of the algorithm can be found in Appendix C.6.

#### 2.3.4.3 Toy example

Let us elaborate on the above concepts with a simple example consisting of a matrix of six users and eight movies, as shown in Figure 2.8. This matrix captures users' ratings for each movie, where each column represents a movie (or item) and each row represents a user. The number in each entry denotes the rating, measured on a scale of 1 to 5 stars, given by the user for that movie. Although the matrix is large, it is sparse as only a few users rated certain movies. The goal of the recommendation problem is to *predict the unobserved entries*, indicated by question marks in rows 1, 2, 4, 5 and 6, as accurately as possible to construct a high-quality RS [LT15].

### 2.3.5 The complexity of predicting user preferences: Moving beyond traditional classification and regression algorithms in RS

In this section, we will differentiate the classical classification and regression approaches from collaborative filtering in the context of RS. We will explore the distinct characteristics of each approach and highlight how they offer alternative perspectives for addressing

$$\mathbf{R} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccccc} A & B & C & D & E & F & G & H \end{array} \\ \begin{bmatrix} - & 5 & - & 2 & 3 & - & ? & 1 \\ 4 & - & 3 & 1 & - & ? & 3 & - \\ - & 5 & 2 & - & 4 & - & 4 & - \\ ? & 5 & - & - & - & 1 & 1 & 2 \\ 3 & - & ? & - & ? & 3 & - & - \\ - & ? & - & 4 & - & - & ? & - \end{bmatrix} \end{array}$$

FIGURE 2.8: Rating matrix where unobserved ratings must be estimated.

recommendation challenges. By understanding the differences between these techniques, we can gain insights into their suitability and applicability in different recommendation scenarios.

#### 2.3.5.1 Understanding the significance and variation of movie ratings on a 1 to 5 scale

When we assign ratings to movies on a scale of 1 to 5, those numbers carry more meaning than just arbitrary labels. The ratings have a specific order that represents the level of liking or preference a user has for a particular movie. Let us delve into this in greater detail:

- **Order of ratings**: The ratings 1, 2, 3, 4, and 5 have a natural order from lowest to highest, indicating varying degrees of liking or enjoyment. A rating of 1 typically suggests that the movie was disliked or received the lowest level of appreciation from the user. On the other hand, a rating of 5 indicates that the movie was highly enjoyed or received the highest level of appreciation from the user.
- **Level of liking or preference**: Each rating value represents a different level of liking or preference. As the rating increases, it generally implies a greater degree of positive sentiment towards the movie. For example, a rating of 1 implies a low level of liking or a negative sentiment towards the movie. Conversely, a rating of 5 indicates a high level of liking or a very positive sentiment towards the movie.
- **Differences between ratings**: The differences between consecutive rating values may not be equal. The distinction between a 1-star movie and a 2-star movie may not be the same as the difference between a 1-star movie and a 4-star movie. This

implies that the intervals between ratings are not necessarily uniform in terms of user perception. In other words, the gap between two rating values might not reflect an equal change in the user's level of liking or preference. To illustrate this further, consider the following examples:

– A 1-star movie might be one that the user found boring or poorly made.
– A 2-star movie could be slightly better than a 1-star movie, but still lacking in certain aspects.
– A 3-star movie might be viewed as average or decent, but not exceptional.
– A 4-star movie could be highly enjoyable, with the user having a strong positive opinion about it.
– A 5-star movie might be considered outstanding, with the user loving every aspect of it.

Therefore, the ratings 1, 2, 3, 4, and 5 hold a *specific order and represent different levels of liking or preference for a movie.* Each rating carries a distinct meaning, indicating varying degrees of positive or negative sentiment. The differences between consecutive ratings are not necessarily uniform, and the magnitude of change in liking or preference may differ between rating values.

#### 2.3.5.2 Collaborative filtering as a generalization of classification and regression

Classification algorithms are designed to work with target variables that consist of distinct, separate categories. These categories are treated as independent and unordered. In the context of movie ratings, if we were to use classification, we would have to assign fixed categories to represent different levels of liking (e.g., "low liking", "medium liking", "high liking"). However, the inherent limitations of treating ratings as unordered categories become evident when we consider the scenario where a user assigns ratings of 2 and 5 to two different movies, indicating their respective degrees of liking. If we were to erroneously treat these ratings as unordered categories, an arbitrary dichotomy might be imposed to delineate them as "Low Liking" and "High Liking" based on an artificial threshold. Consequently, the movie receiving a rating of 2 could be classified under "Low Liking", whereas the one with a rating of 5 might be designated as "High Liking". Nonetheless, this approach of treating the ratings hides the real differences in how much the user likes the two movies. It fails to acknowledge the subtle differences in their preferences by simply adhering to the assigned categories. In reality, the numerical ratings of 2 and 5 provide crucial insight into the hierarchical and quantitative aspects of the individual's liking. The rating of 2 denotes a comparatively moderate level of satisfaction or enjoyment, whereas the rating of 5 signifies a substantially heightened degree of appreciation. By disregarding the inherent order and magnitude of the ratings when treating them as unordered categories, we overlook the fundamental distinction in the individual's preferences. Such an oversimplified treatment negates the fact that the movie rated 5 evoked a significantly greater liking compared to the one rated 2. To enhance the accuracy of personalized movie recommendations that genuinely align with the individual's preferences, it becomes imperative to meticulously consider the *ordinal nature and magnitudes associated with the ratings*. By paying attention to the differences in how much the individual likes different movies, the RS can offer personalized suggestions that really understand their specific preferences. In summary, when we treat ratings as unordered categories, we oversimplify the task and do not fully understand the differences in how much someone likes different movies. But by considering the order and magnitude of the ratings, we can gain better insights into their preferences and make more accurate personalized movie recommendations.

Hence, applying classification directly to movie ratings requires assigning fixed categories, oversimplifying the continuous and ordinal nature of the ratings. This approach fails to consider the specific order and magnitude of the ratings, leading to a loss of information and a lack of precision in capturing the subtleties of user preferences. Regression models, on the other hand, are better suited for modeling movie ratings as they can handle the continuous and ordinal nature of the target variable, allowing for more accurate predictions. Nevertheless, in classification or regression tasks, it is crucial to note that all instances reside in the same space and share identical features. However, this premise does not hold in the context of recommendation problems. If we were to designate users as instances and consider the items as their feature space, we would encounter a significant limitation: the instances would remain *partially characterized* due to the absence of ratings for the complete set of items. The point we want to emphasize is that the traditional ML framework is not well-suited to address the unique challenges posed by RS. These systems represent a distinct problem domain, requiring specialized approaches. One effective way to conceptualize the prediction task within RS is through the concept of matrix completion. The dataset $\mathcal{D}$ can be envisioned as a $\mathbf{R} \in \mathbb{R}^{|U| \times |I|}$ sparse matrix, with users represented as rows and items as columns. The objective is to accurately predict the missing entries of this matrix, thereby completing the recommendation task. This perspective offers a comprehensive understanding of the underlying structure and intrinsic nature inherent in RS problems.

# Chapter 3

# Recommendation Algorithms

---

This chapter provides an overview of the diverse techniques employed in RS. In Section 3.1, we introduce the Baseline Predictor, a straightforward yet highly effective baseline for RS algorithms. In Section 3.2, we delve into commonly utilized algorithms in RS, with a specific focus on collaborative filtering (CF) methods. Next, Section 3.3 elucidates memory-based CF algorithms, which rely on item or user similarity to generate recommendations. Conversely, Section 3.4 encompasses model-based CF algorithms, which leverage a user-item rating matrix model to generate recommendations. Lastly, in Section 3.5, we discuss the metrics employed to evaluate the performance of these algorithms.

## 3.1 Baseline Predictor

To provide a comprehensive analysis of CF techniques, we will first introduce the Baseline Predictor. This simple approach is used to predict ratings and, while it is rarely the primary prediction algorithm for a RS, it is valuable in certain situations. Due to its straightforward implementation, it is typically more reliable in extreme conditions than more complex algorithms. As a result, its predictions often serve as a backup in cases where a more advanced algorithm fails. We will use this basic algorithm as a reference point for comparing the performance of two CF techniques in subsequent sections.

Let us suppose that we choose not to examine the user-item interactions in the matrix $\mathbf{R}$, shown in Figure 2.8. Instead, we simply calculate the average rating of the entire system, denoted as $\bar{r}$, and use it as the predictor for all $\hat{r}_{ui}$ ratings. This approach is inherently inaccurate. To improve upon it, we can incorporate two new vectors: $\mathbf{b}_i \in \mathbb{R}^{I\times 1}$ to model the quality of each movie $i$ relative to the overall average $\bar{r}$, and $\mathbf{b}_u \in \mathbb{R}^{U\times 1}$ to model the bias of each user $u$ relative to $\bar{r}$. Here, $b_u$ is the $u$-th component of $\mathbf{b}_u$, and $b_i$ is the $i$-th component of $\mathbf{b}_i$. The baseline prediction for an unknown item $i$ by $u$ is then:

$$\hat{r}_{ui} = \bar{r} + b_u + b_i \tag{3.1}$$

Equation (3.1) combines three variables to capture the differences between users and items in a simplistic manner. However, it still serves as a useful starting point for RS. For example, in the matrix represented in Figure 2.8, movie C may have a baseline rating of $b_i = 1.2$, but if user 5 is known to be a harsh critic with a baseline rating of $b_u = -0.5$, we can predict that user 5 would rate movie C with $3.6 + 1.2 - 0.5 = 4.3$ stars, where $\bar{r} = 3.6$ is the global average rating. The goal of this simple algorithm is to find the biases $b_u$ and $b_i$ in a way that produces predictions that are as close as possible to the actual ratings. The biases are calculated by solving a regularized least squares problem, as shown in Equation (3.2). This optimization technique balances the accuracy of the predictions with a penalty term that discourages overfitting to the training data.

$$S(b_u, b_i) = \sum_{\forall(u,i)\in\mathcal{D}} (r_{ui} - \hat{r}_{ui})^2 = \sum_{(u,i)\in\mathcal{D}} [r_{ui} - (\bar{r} + b_u + b_i)]^2 \tag{3.2}$$

Equation (3.2) can be efficiently solved using the SGD technique as we will discuss it in Section 3.1.1. Here, $S$, denotes our *loss function.* For the set of model parameters $\mathbf{\Theta} = (\mathbf{b}_u, \mathbf{b}_i)$, where $\mathbf{b}_u$ is the vector containing the user biases and $\mathbf{b}_i$ the item biases, we obtain the optimal values $\hat{\mathbf{\Theta}} = (\mathbf{b}_u^*, \mathbf{b}_i^*)$ such that:

$$\hat{\mathbf{\Theta}} = \min_{\mathbf{\Theta}} S(b_u, b_i) \tag{3.3}$$

$$\hat{\mathbf{\Theta}} = \min_{(b_u,b_i)} \sum_{(u,i)\in\mathcal{D}} [r_{ui} - (\bar{r} + b_u + b_i)]^2 + \lambda(b_u^2 + b_i^2) \tag{3.4}$$

The optimization problem in Equation (3.4) is solved using the observed ratings in the matrix. However, since only a subset of the ratings is available, the model may be susceptible to overfitting. To address this issue, *regularization* is introduced in the form of the term $\lambda$. This term constrains the magnitudes of the biases $b_u$ and $b_i$, preventing them from becoming too large and improving generalization performance. The regularization term $\lambda$ helps the system avoid overfitting the training data and encourages the learned parameters, $b_u$ and $b_i$, to be small. This penalty on the magnitudes of the biases is important for better generalization to unseen data. By regularizing the learned parameters, the model can learn to make accurate predictions on new data, even when the training set is limited. Therefore, to prevent overfitting and improve generalization, the system regularizes the biases $b_u$ and $b_i$ using the regularization term $\lambda$.

### 3.1.1 Optimization through Stochastic Gradient Descent

The Stochastic Gradient Descent technique, described in Section 2.2.1.7, can be applied to minimize Equation (3.4). To compute the partial derivatives of the function $S(b_u, b_i)$ with respect to $b_u$ and $b_i$, we differentiate each term in the function with respect to the corresponding variable while treating other variables as constants. Let us go through the calculation step by step:

$$S(b_u, b_i) = \frac{1}{2} \sum_{(u,i)\in D} \left[r_{ui} - (\bar{r} + b_u + b_i)\right]^2 + \frac{\lambda}{2}\left(b_u^2 + b_i^2\right) \tag{3.5}$$

To find $\frac{\partial S(b_u,b_i)}{\partial b_u}$, we differentiate each term separately. Differentiating the first term:

$$\frac{\partial}{\partial b_u}\left[\frac{1}{2}\left[r_{ui} - (\bar{r} + b_u + b_i)\right]^2\right] = -\left[r_{ui} - (\bar{r} + b_u + b_i)\right] \tag{3.6}$$

Differentiating the second term:

$$\frac{\partial}{\partial b_u}\left[\frac{\lambda}{2} b_u^2\right] = \lambda b_u \tag{3.7}$$

Now, summing up the derivatives of each term, we have:

$$\frac{\partial S(b_u, b_i)}{\partial b_u} = -\left[r_{ui} - (\bar{r} + b_u + b_i)\right] + \lambda b_u = -\mathrm{err}_{ui} + \lambda b_u \tag{3.8}$$

Similarly, to find $\frac{\partial S(b_u,b_i)}{\partial b_i}$, we differentiate each term with respect to $b_i$. Differentiating the first term:

$$\frac{\partial}{\partial b_i}\left[\frac{1}{2}\left[r_{ui}-(\bar{r}+b_u+b_i)\right]^2\right]=-\left[r_{ui}-(\bar{r}+b_u+b_i)\right] \tag{3.9}$$

Differentiating the second term:

$$\frac{\partial}{\partial b_i}\left[\frac{\lambda}{2}b_i^2\right]=\lambda b_i \tag{3.10}$$

Now, summing up the derivatives of each term, we have:

$$\frac{\partial S(b_u,b_i)}{\partial b_i}=-\left[r_{ui}-(\bar{r}+b_u+b_i)\right]+\lambda b_i=-\mathrm{err}_{ui}+\lambda b_i \tag{3.11}$$

These expressions represent the partial derivatives of $S(b_u,b_i)$ with respect to $b_u$ and $b_i$, respectively. Finally, we update the values of $b_u$ and $b_i$ using the rule:

$$\Theta \leftarrow \Theta-\gamma\frac{\partial S\left(f_{\boldsymbol{\Theta}}(u,i),r_{ui}\right)}{\partial\Theta}$$

Subsequently, we obtain the updates for the parameters $b_u$ and $b_i$

$$b_u \leftarrow b_u+\gamma(\mathrm{err}_{ui}-\lambda b_u) \tag{3.12}$$

$$b_i \leftarrow b_i+\gamma(\mathrm{err}_{ui}-\lambda b_i) \tag{3.13}$$

Algorithm 3 outlines the widely used Stochastic Gradient Descent technique. The algorithm takes into account the rating matrix $\mathbf{R}$, model parameters $\boldsymbol{\Theta}$, a prediction function $f_{\boldsymbol{\Theta}}(u,i)$, a convex differentiable loss function $S(f_{\boldsymbol{\Theta}}(u,i),r_{ui})$ and a learning rate $\gamma$. Its objective is to minimize the chosen loss function by iteratively adjusting the model parameters to find the optimal values denoted as $\hat{\boldsymbol{\Theta}}$. The algorithm initiates the model parameters $\boldsymbol{\Theta}$ by assigning random values drawn from a standard normal distribution. It then proceeds with a fixed number of iterations, referred to as epochs, to update the parameters. During each epoch, a *random training example $r_{ui}$ is selected from the set of ratings*. For every parameter $\Theta \in \boldsymbol{\Theta}$ the algorithm performs an update by subtracting the gradient of the loss function with respect to that specific parameter, scaled by the learning rate $\gamma$. This update step aims to adjust the parameters in a direction that reduces the loss function. The process of randomly selecting training examples and updating the parameters continues until the specified number of epochs is reached. At the end of the algorithm, the resulting learned model parameters $\hat{\mathbf{\Theta}}$ reflect the optimized values that provide the best fit to the observed ratings.

**Algorithm 3:** Stochastic Gradient Descent

```
Input: Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$, model parameters $\mathbf{\Theta}$, prediction function $f_{\mathbf{\Theta}}(u, i)$,
       loss function $S(f_{\mathbf{\Theta}}(u, i), r_{ui})$, learning rate $\gamma$ and a number of iterations $n$.
Output: Learned model parameters $\hat{\mathbf{\Theta}}$.
Initialize model parameters $\mathbf{\Theta}$ with random values drawn from $\mathcal{N}(0, 1)$
for iteration $\in [1, n]$ do
    Draw random training example $r_{ui}$ from $\mathcal{D}$
    foreach parameter $\Theta \in \mathbf{\Theta}$ do
        $\Theta \leftarrow \Theta - \gamma \frac{\partial S(f_{\mathbf{\Theta}}(u, i), r_{ui})}{\partial \Theta}$
    end foreach
end for
```

Algorithm 4 presents the complete training algorithm for the Baseline Predictor. Its main loop centers around the epochs, which represent *complete iterations through the training data*. Within each epoch, the algorithm randomly shuffles the observed ratings in the training set to introduce randomness and prevent bias in the learning process. By considering the ratings in a shuffled order, the algorithm ensures that all user-item pairs have an equal chance of being processed. For each rating $r_{ui}$ in the shuffled order, the algorithm computes the predicted rating $\hat{r}_{ui}$ and then calculates the prediction error $\mathrm{err}_{ui}$. To update the user and item biases, the algorithm utilizes Equations (3.8) and (3.11). These updates aim to minimize the overall prediction error by adjusting the biases associated with each user and item. The learning rate controls the step size during the parameter updates, allowing the algorithm to find the optimal values gradually. After processing all ratings in the training set for a given epoch, the algorithm evaluates the performance on the validation set by computing the RMSE. If the RMSE on the validation set improves compared to any previous epoch, indicating a better predictive performance, the algorithm updates the optimized user biases $\mathbf{b}_u^*$ and item biases $\mathbf{b}_i^*$ accordingly. The algorithm terminates if the RMSE on the validation set does not decrease over consecutive epochs, indicating a stable point or limited potential for further improvements. The learned biases $\mathbf{b}_u^*$ and $\mathbf{b}_i^*$ at the best performing epoch, along with the optimal number of epochs $n^*$, are then returned as the final output. A detailed numerical example of the Baseline Predictor algorithm can be found in Appendix C.1.

**Algorithm 4:** Baseline Predictor

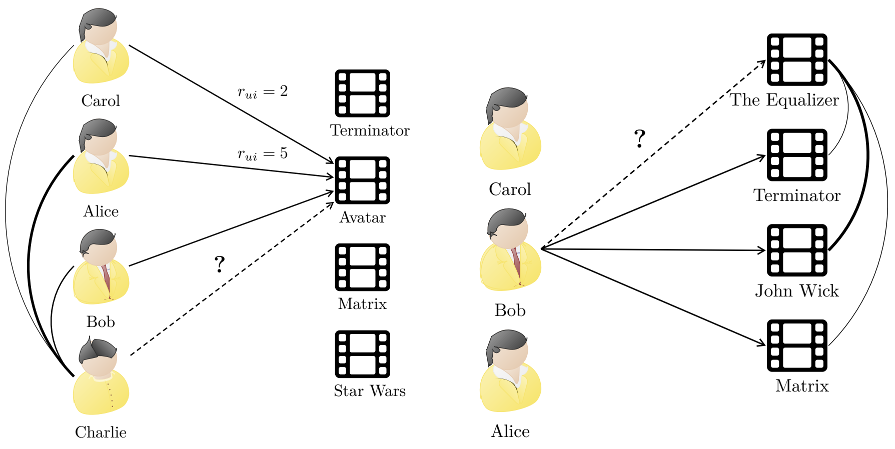
```
Input: Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$, regularization penalty $\lambda$, number of epochs $n$,
       and the learning rate $\gamma$.
Output: $\mathbf{b}_u^*$, $\mathbf{b}_i^*$ and the optimal number of epochs $n^*$.
Construct a training set $\mathcal{D}_{\text{TRAIN}}$
Construct a validation set $\mathcal{D}_{\text{VAL}}$
Initialize $\bar{r} \leftarrow$ global mean of ratings
Initialize $\mathbf{b}_u^*$ and $\mathbf{b}_i^*$ randomly
$n \leftarrow 0$
loop until the terminal condition is met. One epoch:
    $n \leftarrow n + 1$
    Randomly shuffle observed ratings in $\mathcal{D}_{\text{TRAIN}}$
    foreach $(u, i) \in \mathcal{D}_{\text{TRAIN}}$ in shuffled order do
        $\hat{r}_{ui} = \bar{r} + b_u + b_i$
        $\text{err}_{ui} = r_{ui} - \hat{r}_{ui}$
        /* Update variables according to Equations (3.8) and (3.11)   */
        $b_u = b_u + \gamma(\text{err}_{ui} - \lambda b_u)$
        $b_i = b_i + \gamma(\text{err}_{ui} - \lambda b_i)$
    end foreach
    Compute the RMSE on $\mathcal{D}_{\text{VAL}}$ using Equation (2.12)
    if the RMSE on $\mathcal{D}_{\text{VAL}}$ was better than in any previous epoch:
        $\mathbf{b}_u^* \leftarrow \mathbf{b}_u$, $\mathbf{b}_i^* \leftarrow \mathbf{b}_i$, $n^* \leftarrow n$
    end
    Terminal condition: RMSE on $\mathcal{D}_{\text{VAL}}$ does not decrease.
end
```

## 3.2 Collaborative filtering: Taxonomy

In the context of CF algorithms, a widely used taxonomy categorizes them into two main groups: model-based and memory-based algorithms [BHK98]. Memory-based algorithms are data-driven and make rating predictions based on past data. These algorithms memorize the entire rating matrix with existing ratings and utilize nearest neighbor ($k$-NN) methods to predict unobserved ratings. Due to their reliance on past data, memory-based algorithms do not function without sufficient data. Additionally, as they require all ratings, users and items to be stored in memory, they do not have a learning phase. In contrast, model-based algorithms are knowledge-driven and assume a structural relationship between users and their interests. During the learning phase, a model is constructed by adjusting observed ratings and, during the recommendation phase, the model is used to generate recommendations. As such, model-based algorithms are able to generate recommendations even when no past data is available. However, compared to memory-based algorithms, they require a longer learning phase. A more comprehensive analysis and comparison of memory-based and model-based methods will be presented in Section 3.4, specifically when introducing the model-based method. A third algorithm category is the hybrid algorithm, which combines the strengths of both memory-based and model-based algorithms. In practice, most RS go through various stages of operation where rating data is unavailable, and only user profiles are available. In such cases, memory-based algorithms cannot operate effectively. Therefore, hybrid algorithms are designed to start with a model-based approach and then combine the advantages of memory-based algorithms in later stages. By doing so, hybrid algorithms can overcome the limitations of both model-based and memory-based algorithms and provide more accurate recommendations.

## 3.3 Memory-based collaborative filtering

Memory-based CF is a technique that can be classified into two types: *user-based* and *item-based*. The former focuses on the similarities among users, while the latter focuses on the similarities between items [RIS+94, SKKR01]. We provide a toy example in Figure 3.1a and 3.1b to illustrate these approaches.



(A) User-based approach (B) Item-based approach

FIGURE 3.1: The two approaches used in the memory-based method.

The user-based approach involves calculating *similarities between the target user and other users*, selecting the closest neighbors based on these similarities and using the ratings of the neighbors on a specific item to compute a weighted average. *The resulting weighted average is the predicted rating for the target user*, with the similarities serving as weights. The diagram in Figure 3.1a illustrates the correlation strength between users, namely Carol, Alice, Bob and Charlie, based on their previous ratings. The thickness of the black lines connecting them indicates the strength of the correlation, with thicker lines signifying stronger correlations and thinner lines indicating weaker ones. The objective of this scenario is to *predict the rating* that Charlie will give to Avatar. An impersonalized approach would involve calculating the average rating from all users. However, to personalize Avatar's rating for Charlie, the algorithm selects the $k$ most similar users and weighs their ratings based on their correlation with Charlie. Thus, a weighted average based on the most similar users replaces the impersonalized average across all users. This *weighted average is the predicted rating* for Avatar for Charlie. This approach may face scalability issues, particularly when dealing with large datasets such as those found in online video streaming sites with millions of registered users. As a result, it is more suitable for systems where the number of items is greater than the number of users, and where the existing group in the system does not change frequently [YWZ+16].

In contrast, the item-based approach focuses on *similarities between items*. The system first calculates similarities between items, selects the closest neighbors based on these similarities and computes a weighted average of their ratings. *This weighted average is used as the predicted rating for the item.* The similarities between movies are illustrated in Figure 3.1b, with black lines representing these similarities. To predict Bob's rating for The Equalizer, we select the $L$ most similar movies to The Equalizer that Bob has previously rated and compute a weighted average based on their most similar ratings. The most similar movie to The Equalizer is John Wick, and we can recommend it to Bob since the system predicts that he will rate it similarly to John Wick. Similar to the user-based approach, the item-based approach is more suitable for systems with a large number of users compared to the number of items and where the items do not frequently change. By exploiting the similarities between items, the item-based approach provides a complementary perspective to the user-based approach in personalized recommendations.

### 3.3.1 User-based approach: $k$-NN$_{\text{USERS}}$

This method predicts the rating $r_{ui}$ of a user $u$ for a new item $i$ using the ratings given to $i$ by the $L$ users most similar to $u$. Let us suppose that we have for each user $v \neq u$ a value $d_{uv}$ that represents the similarity between $u$ and $v$. Later, in Section 3.3.1.2, we will discuss how we can compute this similarity. The $L$ closest neighbors of $u$, denoted by $\mathcal{L}(u)$, are the $L$ users $v$ with the greatest similarity $d_{uv}$ to $u$. However, only users who have rated the $i$ item can be used in the prediction of $r_{ui}$ and instead we consider the $L$ users most similar to $u$ who have rated $i$. That is, the similarities $u$ and $v$ are computed based on the set of items that both users have rated $I_{uv} = I_u \cap I_v$. To compute the rating prediction $r_{ui}$ of user $u$ for the item $i$ we must select the $L$ most similar users $\mathcal{L}_i^L(u)$ who rated the item $i$, i.e., $\mathcal{L}_i^L(u)$ is the set of the $L$ closest users to $u$ that have interacted with the same item $i$. This set of similar users is a subset of all users $\mathcal{L}_i^L(u) \subset U_i$ who have rated item $i$.

#### 3.3.1.1 Generation of recommendations

The process of generating recommendations involves predicting a rating, a numerical value that reflects the anticipated opinion of the user for an unseen item. To ensure consistency with existing ratings in the original dataset, this value must be within the same numerical scale $\mathcal{R}$. Consequently, a prediction score is calculated using Equation (3.14).

$$\hat{r}_{ui} = \frac{1}{|\mathcal{L}_i^L(u)|} \sum_{v \in \mathcal{L}_i^L(u)} r_{vi} \tag{3.14}$$

One issue with Equation (3.14) is that it disregards the possibility that neighbors may have varying degrees of similarity. To address this problem, a frequently employed approach is to incorporate the contribution of each neighbor according to their *similarity* with user $u$. Nonetheless, if these weights do not sum up to 1, the predicted ratings may fall outside the allowable range. As such, it is a standard practice to normalize these weights to ensure that the rating prediction conforms to the following expression:

$$\hat{r}_{ui} = \frac{\sum\limits_{v \in \mathcal{L}_i^L(u)} d_{uv} \cdot r_{vi}}{\sum\limits_{v \in \mathcal{L}_i^L(u)} |d_{uv}|} \tag{3.15}$$

Figure 3.2 illustrates this concept. Suppose we want to predict the rating of movie B for user 2. We can exploit the similarity between the users 1 and 2 and $r_{1B}$. The rating $r_{ui}$ can be estimated as the average rating given to $i$ by these neighbors.

#### 3.3.1.2 Metrics for quantifying users' similarity

The use of similarity weights in recommendation methods based on nearest neighbors serves a dual purpose: ($i$) selecting trustworthy neighbors whose scores contribute to prediction and ($ii$), assigning varying levels of importance to these neighbors during prediction. The accuracy and performance of a nearest neighbor-based RS heavily depend on the computation of similarity weights [DK11]. To this end, we compute *similarities*

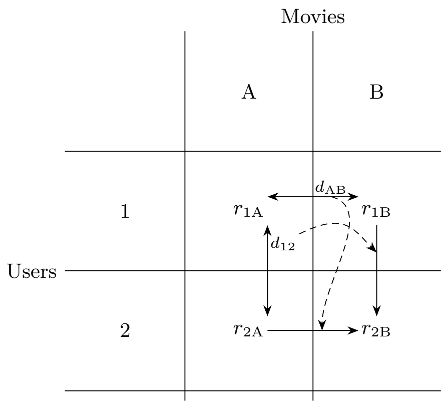


FIGURE 3.2: User-user correlation vs. movie-movie correlation (Reference:[Chi12]).

$(d_{uv})$ between the *target user* and other users in the rating matrix's rows, but *only for items that have been jointly rated.* Below, we describe commonly used metrics for this approach.

**Definition 3.1.** (Cosine similarity). The **cosine similarity** between two users' $u$ and $v$ is defined as:

$$d_{uv}^{Cosine} \overset{\text{def}}{=} \frac{\sum\limits_{i \in I_{uv}} r_{ui} \cdot r_{vi}}{\sqrt{\sum\limits_{i \in I_u} r_{ui}^2 \cdot \sum\limits_{i \in I_v} r_{vi}^2}} \tag{3.16}$$

where $I_{uv}$ is the set of items rated by users $u$ and $v$, $I_u$ is the set of all items rated by user $u$, and $I_v$ is the set of all items rated by the user $v$. One problem with this metric is that it does not consider the differences in the mean and variance of user ratings $u$ and $v$. One metric that compares ratings where the effects of the mean and variance have been eliminated is the Pearson similarity, which can be viewed as a mean-centered version of **cosine similarity**.

**Definition 3.2.** (Pearson similarity). The **Pearson similarity** between two users $u$ and $v$ is defined as:

$$d_{uv}^{PCC} \overset{\text{def}}{=} \frac{\sum\limits_{i \in I_{uv}} (r_{ui} - \bar{r}_u) \cdot (r_{vi} - \bar{r}_v)}{\sqrt{\sum\limits_{i \in I_u} (r_{ui} - \bar{r}_u)^2} \cdot \sqrt{\sum\limits_{i \in I_v} (r_{vi} - \bar{r}_v)^2}} \tag{3.17}$$

where $\bar{r}_u$ and $\bar{r}_v$ are the average rating of users $u$ and $v$, respectively.

**Definition 3.3.** (Pearson similarity centered with the Baseline Predictor). The **Pearson similarity centered on the Baseline Predictor** between two users $u$ and $v$ is defined as:

$$\hat{\beta}_{uv} \overset{\text{def}}{=} \frac{\sum\limits_{i \in I_{uv}} \tilde{r}_{ui} \cdot \tilde{r}_{vi}}{\sqrt{\sum\limits_{i \in I_u} \tilde{r}_{ui}^2} \cdot \sqrt{\sum\limits_{i \in I_v} \tilde{r}_{vi}^2}} = \frac{\sum\limits_{i \in I_{uv}} (r_{ui} - b_{ui}) \cdot (r_{vi} - b_{vi})}{\sqrt{\sum\limits_{i \in I_u} (r_{ui} - b_{ui})^2} \cdot \sqrt{\sum\limits_{i \in I_v} (r_{vi} - b_{vi})^2}} \tag{3.18}$$

To maintain consistency with our previous analysis outlined in Section 3.1.1, we compute the baselines $b_{ui}$ and $b_{vi}$ using the SGD technique, although Alternating Least Squares [BK07b], can also be utilized for this purpose. The $k$-NN$_{\text{USERS}}$ approach is shown in Algorithm 5.

**Algorithm 5:** $k$-NN$_{\text{USERS}}$

```
**Input:** Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$.
**Output:** $\hat{\mathbf{R}}$ an estimation of $\mathbf{R}$.
*Training phase*:
forall $(u, v) \in U^2$ do
    Compute $d_{uv}$
end forall
*Prediction phase*:
num ← 0, denom ← 0
$U_i^s$ ← Sorted($U_i$) // Sort by value of similarity with u, in decreasing order
forall $v \in U_i^s[: L]$ do
    I.e. for the first L users in $U_i^s$
    num ← num + $r_{vi} \cdot d_{uv}$
    denum ← denum + $d_{uv}$
end forall
$\hat{r}_{ui} \leftarrow \frac{\text{num}}{\text{denom}}$
```

### 3.3.1.3 PCCBaseline metric

Since most ratings are unobserved, it is common for users to have few items in common. Therefore, the Pearson similarity calculation, which only considers shared items between two users, is more reliable when the number of shared items is higher. However, when few items are shared, overfitting can occur. To address this issue, we employ a modified version of the correlation coefficient, denoted as **PCCBaseline**, which is based on a reduced correlation coefficient that appears in Equation (3.18).

$$d_{uv}^{PCCBaseline} = \frac{|I_{uv}| - 1}{|I_{uv}| - 1 + \xi} \cdot \hat{\beta}_{uv} \tag{3.19}$$

where $|I_{uv}|$ is the number of items rated by users $u$ and $v$. We call $\xi$ as the shrinkage coefficient [KB11]. Once computed, all the similarities $\{d_{uv}\}$ between users are stored in a $\mathbf{D}$ matrix of $U \times U$. Then for a given user $u$ we rank all other users, indexed by $v$, in descending order of similarity $|d_{uv}|$ and select the top $L$ users to serve as neighbors for inclusion in the neighborhood modeling process. The value of $L$ is an integer between 1 and $U - 1$. Algorithm 6 shows how to compute this metric in practical terms.

**Algorithm 6:** Computation of the PCCBaseline similarity

```
Input: Matrix R, set of ratings $\mathcal{D}$.
Output: Similarity matrix D.
Initialization (n is defined as the number of users):
d = empty_array[$n$][$n$]
sum_prod = empty_array[$n$][$n$]
sum_cuadu = empty_array[$n$][$n$]
sum_cuadv = empty_array[$n$][$n$]
forall $i \in I$ do
    forall $u \in U_i$ do
        forall $v \in U_i$ do
            sum_prod($u, v$)  += sum_prod($u, v$)  + $(r_{ui} - b_{ui}) \cdot (r_{vi} - b_{vi})$
            sum_cuadu($u, v$) += sum_cuadu($u, v$) + $(r_{ui} - b_{ui})^2$
            sum_cuadv($u, v$) += sum_cuadv($u, v$) + $(r_{vi} - b_{vi})^2$
        end forall
    end forall
end forall
forall $u \in U_i$ do
    forall $v \in U_i$ do
        d($u, v$) = $\frac{\text{sum\_prod}(u,v)}{\sqrt{\text{sum\_cuadu}(u,v) \cdot \text{sum\_cuadv}(u,v)}}$
    end forall
end forall
```

### 3.3.1.4 User-based CF with Baseline Predictor: $k$-NN$^{\bigstar}_{\text{USERS}}$

Recently, more sophisticated $k$-NN methods have been developed. As an example, during the Netflix competition a more robust similarity metric was proposed by [BK07a]. In their work, the authors have redefined the standard prediction formula (see Equation (3.15)) to create a new one, as can be seen in Equation (3.20).

$$\hat{r}_{ui} = b_{ui} + \frac{\sum\limits_{v \in \mathcal{L}_i^L(u)} d_{uv} \cdot \tilde{r}_{vi}}{\sum\limits_{v \in \mathcal{L}_i^L(u)} |d_{uv}|} = (\bar{r} + b_u + b_i) + \frac{\sum\limits_{v \in \mathcal{L}_i^L(u)} d_{uv} \cdot (r_{vi} - b_{vi})}{\sum\limits_{v \in \mathcal{L}_i^L(u)} |d_{uv}|} \tag{3.20}$$

The three terms enclosed within the parentheses, namely $(\bar{r} + b_u + b_i)$, from the above formula denote the estimation that is made prior to exploiting the similarities of users. Secondly, the dividend term constitutes a weighted summation of the *knowledge we have gathered from collaborative filtering.* The predicted rating can be expressed as the *Baseline Predictor plus a weighted sum of the ratings from the neighboring users*, which are then normalized by the corresponding weights. Equation (3.20) employs the PCCBaseline similarity. This metric, similar to the Pearson similarity, centers the ratings using the *Baseline Predictors* instead of their averages. The whole process is shown in Algorithm 7.

**Algorithm 7:** $k$-NN$^{\star}_{\text{USERS}}$

```
Input: Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$, regularization penalty $\lambda$, number of epochs $n$,
       learning rate $\gamma$ and the number of neighbors $L$.
Output: $\hat{\mathbf{R}}$ an estimation of $\mathbf{R}$.
Training phase:
Initialize $\bar{r} \leftarrow$ global mean of ratings
Initialize $b_u$ and $b_i$ randomly
for epoch $\in [1, n]$ do
    forall $r_{ui} \in \mathcal{D}$: do
        $\text{err}_{ui} = r_{ui} - (\bar{r} + b_u + b_i)$
        $b_u = b_u + \gamma(\text{err}_{ui} - \lambda b_u)$
        $b_i = b_i + \gamma(\text{err}_{ui} - \lambda b_i)$
    end forall
end for
forall $(u, v) \in U^2$ do
    Compute $d_{uv}$ // Run Algorithm 6 to obtain the similarity matrix
end forall
Prediction phase:
$U_i^s \leftarrow \text{Sorted}(U_i)$ // Sort by value of similarity with u, in decreasing order
forall $v \in U_i^s[: L]$ do
    I.e. for the first L users in $U_i^s$
    num $\leftarrow$ num $+ d_{uv} \cdot (r_{vi} - b_{vi})$
    denom $\leftarrow$ denom $+ d_{uv}$
end forall
$\hat{r}_{ui} \leftarrow b_{ui} + \frac{\text{num}}{\text{denom}}$
```

Algorithm 7 accepts a rating matrix $\mathbf{R}$ as input and proceeds with multiple steps to produce predictions for user-item pairs. Firstly, it employs Algorithm 4 to train the Baseline Predictor, resulting in the prediction matrix $\hat{\mathbf{R}}$ comprising predicted ratings $b_{ui}$ for each user-item pair $(u, i)$. Subsequently, it normalizes the rating matrix by subtracting $\hat{\mathbf{R}}$ from $\mathbf{R}$ yielding the normalized matrix $\tilde{\mathbf{R}}$, thereby ensuring unbiased ratings with a shared reference point.

The algorithm then proceeds to compute the user-user similarity matrix $\mathbf{D}$ by evaluating the similarity between users using Equation (3.19). For each user $u$ in the system, it identifies the $L$ nearest neighbors based on the similarity matrix $\mathbf{D}$. Subsequently, for every item $i$ that user $u$ has not rated, a prediction is generated by combining the Baseline Predictor with the nearest neighbors formula, as illustrated in Equation (3.20). We shall refer to the proposed algorithm as $k$-NN$^{\bigstar}_{\text{USERS}}$, which is distinct from $k$-NN$_{\text{USERS}}$ (refer to Algorithm 5). A detailed numerical example of $k$-NN$^{\bigstar}_{\text{USERS}}$ is provided in Appendix C.3.

### 3.3.2 Item-based approach: $k$-NN$^{\bigstar}_{\text{ITEMS}}$

Unlike the user-based approach, only the role of rows and columns is swapped here. To determine the rating of the item $i$ by the user $u$, the system looks for neighbors of the item $i$ instead of the user $u$. By calculating the similarity between $i$ and all other items, we can select the $L$ items most similar to $i$, just as we did with users in the user-based method.

Where $\mathcal{L}_u^L(i) \subset I_u$, is the set of items most similar to item $i$ that were rated by user $u$. The similarity between two items, $i$ and $j$, is measured using some of the metrics described in Section 3.3.2.1, which are defined in the subset of ratings of users who rated both items $U_{ij} = U_i \cap U_j$. The $u$ rating prediction for $i$ is obtained as the weighted average of the ratings given by $u$ to the items of $\mathcal{L}_u^L(i)$:

$$\hat{r}_{ui} = \frac{\sum\limits_{j \in \mathcal{L}_u^L(i)} d_{ij} \cdot r_{uj}}{\sum\limits_{j \in \mathcal{L}_u^L(i)} |d_{ij}|} \tag{3.21}$$

Let us suppose now that we want to predict the rating of movie B for user 2 as shown in Figure 3.2. We can exploit the similarity between movies A and B and $r_{2A}$. The complete process for the item-based approach is depicted in Algorithm 8.

**Algorithm 8:** $k$-NN$_{\text{ITEMS}}$

```
Input: Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$.
Output: $\hat{r}_{ui}$, the prediction of $r_{ui}$.
Training phase:
forall $(i, j) \in I^2$ do
    Compute $d_{ij}$.
end forall
Prediction phase:
num $\leftarrow$ 0, denom $\leftarrow$ 0
$I_u^s \leftarrow$ Sorted($I_u$) // Sort by value of similarity with $j$, in decreasing order.
forall $j \in I_u^s[: L]$ do
    I.e for the first L items in $I_u^s$
    num $\leftarrow$ num + $\tilde{r}_{uj} \cdot d_{ij}$
    denum $\leftarrow$ denum + $d_{ij}$
end forall
$\hat{r}_{ui} \leftarrow \frac{\text{num}}{\text{denom}}$
```

### 3.3.2.1 Metrics for quantifying items' similarity

We now define the metrics that are commonly used to compute the similarity between *items*. We can use the metrics described in (3.16) - (3.19) using the column vectors $i$ and $j$ from the rating matrix instead of the row vectors.

**Definition 3.4.** (Cosine similarity). The **cosine similarity** between two items $i$ and $j$ is defined as:

$$d_{ij}^{Cosine} \overset{\text{def}}{=} \frac{\sum\limits_{u \in U_{ij}} r_{ui} \cdot r_{uj}}{\sqrt{\sum\limits_{u \in U_i} r_{ui}^2 \cdot \sum\limits_{u \in U_j} r_{uj}^2}} \tag{3.22}$$

where $U_i$ is the set of all users who rate item  and $U_j$ is the set of all users who rate item $j$. Finally, $u \in U_{ij}$ is the set of users $u$ who rated the items $i$ and $j$.

**Definition 3.5.** (Pearson similarity). The **Pearson similarity** between two items $i$ and $j$ is defined as:

$$d_{ij}^{PCC} \overset{\text{def}}{=} \frac{\sum\limits_{u \in U_{ij}} (r_{ui} - \bar{r}_i) \cdot (r_{uj} - \bar{r}_j)}{\sqrt{\sum\limits_{u \in U_i} (r_{ui} - \bar{r}_i)^2} \cdot \sqrt{\sum\limits_{u \in U_j} (r_{uj} - \bar{r}_j)^2}} \tag{3.23}$$

where $\bar{r}_i$ and $\bar{r}_j$ are the average grade of the items $i$ and $j$, respectively.

**Definition 3.6.** (Pearson similarity centered on the Baseline Predictor). The **Pearson similarity centered on the Baseline Predictor** between two items $i$ and $j$ is defined as:

$$\hat{\beta}_{ij} \overset{\text{def}}{=} \frac{\sum\limits_{u \in U_{ij}} \tilde{r}_{ui} \cdot \tilde{r}_{uj}}{\sqrt{\sum\limits_{u \in U_i} \tilde{r}_{ui}^2} \cdot \sqrt{\sum\limits_{u \in U_j} \tilde{r}_{uj}^2}} = \frac{\sum\limits_{u \in U_{ij}} (r_{ui} - b_{ui}) \cdot (r_{uj} - b_{uj})}{\sqrt{\sum\limits_{u \in U_i} (r_{ui} - b_{ui})^2} \cdot \sqrt{\sum\limits_{u \in U_j} (r_{uj} - b_{uj})^2}} \tag{3.24}$$

The reduced correlation coefficient, as represented in Equation (3.19), is incorporated into Equation (3.24) to calculate the similarity metric **PCCBaseline**, as outlined in Equation (3.25). The inclusion of this coefficient helps mitigate overfitting when there is a limited number of common users. Here, $|U_{ij}|$ represents the count of users who have rated both items $i$ and $j$. The computation of this metric is described in Algorithm 9.

$$d_{ij}^{PCCBaseline} = \frac{|U_{ij}| - 1}{|U_{ij}| - 1 + \xi} \cdot \hat{\beta}_{ij} \tag{3.25}$$

**Algorithm 9:** Computation of the PCCBaseline similarity

```
Input: Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$.
Output: Similarity $d_{ij}^{PCCBaseline}$ for all pairs of items.
Initialization (n is defined as the number of users):
d = empty_array[$n$][$n$]
sum_prod = empty_array[$n$][$n$]
sum_cuadi = empty_array[$n$][$n$]
sum_cuadj = empty_array[$n$][$n$]
forall $u \in U$ do
    forall $i \in I_u$ do
        forall $j \in I_u$ do
            sum_prod($i,j$) += sum_prod($i,j$) + $(r_{ui} - b_{ui}) \cdot (r_{uj} - b_{uj})$
            sum_cuadi($i,j$) += sum_cuadi($i,j$) + $(r_{ui} - b_{ui})^2$
            sum_cuadj($i,j$) += sum_cuadj($i,j$) + $(r_{uj} - b_{uj})^2$
        end forall
    end forall
end forall
forall $i \in I_u$ do
    forall $j \in I_u$ do
        d($i,j$) = $\frac{\text{sum\_prod}(i,j)}{\sqrt{\text{sum\_cuadu}(i,j) \cdot \text{sum\_cuadv}(i,j)}}$
    end forall
end forall
```

### 3.3.2.2 Item-based CF with Baseline Predictor: $k$-NN$^{\bigstar}_{\textbf{items}}$

The algorithm's working principle is similar to Algorithm 7, with a focus on items, instead. The prediction formula is given below:

$$\hat{r}_{ui} = b_{ui} + \frac{\sum\limits_{j \in \mathcal{L}_u^L(i)} d_{ij} \cdot \tilde{r}_{uj}}{\sum\limits_{j \in \mathcal{L}_u^L(i)} |d_{ij}|} = (\bar{r} + b_u + b_i) + \frac{\sum\limits_{j \in \mathcal{L}_u^L(i)} d_{ij} \cdot (r_{uj} - b_{uj})}{\sum\limits_{j \in \mathcal{L}_u^L(i)} |d_{ij}|} \tag{3.26}$$

**Algorithm 10:** $k$-NN$^{\bigstar}_{\text{ITEMS}}$

```
Input: Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$, regularization penalty $\lambda$, number of epochs $n$,
       learning rate $\gamma$ and the number of neighbors $L$.
Output: $\hat{\mathbf{R}}$ an estimation of $\mathbf{R}$.
Training phase:
Initialize $\bar{r} \leftarrow$ global mean of ratings
Initialize $b_u$ and $b_i$ randomly
for epoch $\in [1, n]$ do
    forall $r_{ui} \in \mathcal{D}$: do
        $\text{err}_{ui} = r_{ui} - (\bar{r} + b_u + b_i)$
        $b_u = b_u + \gamma(\text{err}_{ui} - \lambda b_u)$
        $b_i = b_i + \gamma(\text{err}_{ui} - \lambda b_i)$
    end forall
end for
forall $(i, j) \in I^2$ do
    Compute $d_{ij}$ // Run Algorithm 9 to obtain the similarity matrix
end forall
Prediction phase:
$I_u^s \leftarrow \text{Sorted}(I_u)$ // Sort by value of similarity with $j$, in decreasing order
forall $j \in I_u^s[: L]$ do
    I.e. for the first L items in $I_u^s$
    $\text{num} \leftarrow \text{num} + d_{ij} \cdot (r_{uj} - b_{uj})$
    $\text{denom} \leftarrow \text{denom} + d_{ij}$
end forall
$\hat{r}_{ui} \leftarrow b_{ij} + \frac{\text{num}}{\text{denom}}$
```

The proposed algorithm, denoted as Algorithm 10, operates on a rating matrix $\mathbf{R}$ as input. Initially, it trains the Baseline Predictor using Algorithm 4, yielding the matrix $\hat{\mathbf{R}}$ comprising predicted baseline values $b_{uj}$. Subsequently, it computes the normalized rating matrix $\tilde{\mathbf{R}}$ by subtracting $\hat{\mathbf{R}}$ from $\mathbf{R}$ to eliminate biases and establish a consistent reference point for all ratings. Next, it determines the similarity between *items* utilizing Equation

(3.25), resulting in the item-item similarity matrix $\mathbf{D}$. For each item $j$ in the system, it identifies the $L$ nearest neighbors of $i$ based on the similarity matrix $\mathbf{D}$. Consequently, employing Equation (3.26), it calculates the final prediction $\hat{r}_{ui}$ for each user-item pair. To distinguish this algorithm from the generic $k$-NN$_{\text{ITEMS}}$ algorithm presented in Algorithm 8, we refer to it as $k$-NN$^{\bigstar}_{\text{ITEMS}}$. A comprehensive numerical example of the $k$-NN$^{\bigstar}_{\text{ITEMS}}$ algorithm can be found in Appendix C.2.

## 3.4 Model-based collaborative filtering

In this section, we first start by examining and emphasizing the significant distinctions between model-based methods and memory-based methods. By doing so, we will set the stage for our subsequent exploration of model-based methods.

Model-based methods involve building a mathematical or statistical model that represents the underlying relationship between the input variables and the output or target variable. This model is *trained on a dataset to learn the patterns and relationships within the data.* In contrast, memory-based methods, also known as instance-based or lazy learning methods, rely on the observed ratings in the training dataset. These methods do not explicitly build a model but use the training data directly to make predictions or perform computations.

The **learning process** differs between the two approaches: model-based methods typically involve a training phase where the model's parameters or structure is learned from the training data. This process involves optimization techniques such as SGD (see Section 2.2.1.7), Maximum Likelihood Estimation or other optimization algorithms. On the other hand, memory-based methods *do not require an explicit training phase.* Instead, they store the training data in memory and use them directly for prediction or computation. The learning process primarily involves the storage and retrieval of the training data. Another distinction lies in the generalization capability of the methods. Model-based methods aim to generalize from the training data by capturing the underlying patterns and relationships. Once the model is trained, it can make predictions on new, unseen data by applying the learned patterns. In contrast, memory-based methods directly use the observed ratings from the training data for making predictions. These methods do not explicitly generalize beyond the training data, which means they may have limited performance on unseen or out-of-distribution data.

**Scalability** is another factor to consider. Model-based methods can handle large datasets efficiently, as they rely on a compact representation of the learned patterns or relationships. Once the model is trained, making predictions on new data can be computationally efficient. On the other hand, memory-based methods can be memory-intensive, especially when dealing with large training datasets. As they rely on storing the training data, the memory requirements can increase with the size of the dataset.

Furthermore, **interpretability** differs between the two approaches. Model-based methods often provide interpretability, as the learned model can reveal insights into the relationships between input variables and the output. The parameters and structure of the model can be analyzed and understood. In contrast, memory-based methods are generally less interpretable since the predictions are based on direct comparisons or computations using the observed ratings. The decision-making process may be harder to interpret or explain.

Model-based CF algorithms leverage ML to address the aforementioned challenges. These algorithms operate under the assumption of a parameterized model denoted as $f_{\boldsymbol{\Theta}}(u, i)$, where $\boldsymbol{\Theta}$ represents an unknown parameter vector. To make predictions, the model undergoes training using available data, aiming to estimate the parameter vector that minimizes the loss function $S$, as shown in Equation (3.27).

$$\hat{\boldsymbol{\Theta}} = \underset{\boldsymbol{\Theta}}{\operatorname{argmin}}\, S(f_{\boldsymbol{\Theta}}(u, i), r_{ui}) \tag{3.27}$$

The loss function quantifies the fitting error by measuring the difference between the model's prediction $f_{\boldsymbol{\Theta}}(u, i)$ and the actual rating $r_{ui}$ based on historical data ($\mathbf{R}$). It can also include a regularization term to discourage excessive model complexity associated with the parameter vector $\boldsymbol{\Theta}$. By optimizing the loss function, the model strives to achieve accurate predictions while balancing the trade-off between fitting the data and maintaining a manageable level of complexity to prevent overfitting. Once the parameter vector $\hat{\boldsymbol{\Theta}}$ is estimated, the model can predict ratings for any user-item pair $(u, i)$ using Equation (3.28).

$$\hat{r}_{ui} = f_{\hat{\boldsymbol{\Theta}}}(u, i) \tag{3.28}$$

Model-based methods include clustering models [UFA$^+$98], regression-based models [LM05] and many others. A recent class of model-based collaborative filtering is latent factor [Hof04].

### 3.4.1 Matrix factorization

Latent factor models represent a cutting-edge method in the field of RS. These models rely on established dimensionality reduction techniques to address the problem of missing entries in the rating matrix. Specifically, matrix factorization is a sub-type of latent factor models that gained prominence during the Netflix Prize contest. The technique was inspired by singular value decomposition and proved to be more effective than the original item-based collaborative filtering approach utilized by Netflix, which is detailed in Algorithm 8. In fact, matrix factorization resulted in a 4% performance improvement compared to the collaborative item-based filtering approach initially employed by Netflix.

### 3.4.2 Singular value decomposition

Let us consider the case in which the rating matrix is *fully dense*, that is, it does not have unobserved ratings. We can factorize $\mathbf{R}$ using singular value decomposition. Proposition 3.7 illustrates this concept.

**Proposition 3.7.** *Any (fully dense) real-valued matrix $\mathbf{R} \in \mathbb{R}^{m \times n}$ can be decomposed as the product of three lower-ranked matrices:*

$$\mathbf{R} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$$

*where $\mathbf{U}$ is a $m \times m$ matrix with columns containing the $m$ orthonormal eigenvectors of $\mathbf{R}\mathbf{R}^\top$. The matrix $\mathbf{V}$ is a $n \times n$ matrix with columns containing the $n$ orthonormal eigenvectors of $\mathbf{R}^\top\mathbf{R}$. $\mathbf{\Sigma}$ is a $m \times n$ diagonal matrix in which only the diagonal entries are nonzero and contain the square root of the nonzero eigenvalues of $\mathbf{R}^\top\mathbf{R}$. This factorization is called* ***Singular Value Decomposition*** *(SVD) of $\mathbf{R}$.*

We can also roughly factorize the rating matrix $\mathbf{R}$ using **Truncated SVD** of rank $k \ll \min\{m, n\}$:

$$\mathbf{R} \approx \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^\top \tag{3.29}$$

where $\mathbf{U}_k \in \mathbb{R}^{m \times k}$, $\mathbf{\Sigma}_k \in \mathbb{R}^{k \times k}$ and $\mathbf{V}_k^\top \in \mathbb{R}^{n \times k}$. Matrices $\mathbf{U}_k$ and $\mathbf{V}_k$ contain, respectively, the largest $k$ eigenvectors of $\mathbf{R}\mathbf{R}^\top$ and $\mathbf{R}^\top\mathbf{R}$, while $\mathbf{\Sigma}_k$ (diagonal) contains the square roots (not negative) of the $k$ largest eigenvalues of any of the matrices along its diagonal [Agg16].

The diagonal matrix $\mathbf{\Sigma}_k$ can be absorbed into the user factors $\mathbf{U}_k$ or the factors $\mathbf{V}_k$ of items. By convention, user and item factors are defined as follows:

$$\begin{aligned}\mathbf{P} &= \mathbf{U}_k \\ \mathbf{Q} &= \mathbf{\Sigma}_k \mathbf{V}_k\end{aligned}$$

In this way, the factorization of the rating matrix leaves us with $\mathbf{R} = \mathbf{P}\mathbf{Q}^\top$. Thus, the goal of the factoring process is to find the matrices $\mathbf{P}$ and $\mathbf{Q}$ with orthogonal columns.

$$\min S = \|\mathbf{R} - \hat{\mathbf{R}}\|_{\mathrm{F}}^2 = \|\mathbf{R} - \mathbf{P}\mathbf{Q}^\top\|_{\mathrm{F}}^2 \tag{3.30}$$

Subject to:
The columns of $\mathbf{P}$ are mutually orthogonal
The columns of $\mathbf{Q}$ are mutually orthogonal

The major challenge associated with this approach is that the rating matrix is *frequently not fully dense*, which is typical in RS. For instance, the ML 100K and 1M datasets contain merely about 6% and 4% observed ratings, respectively. One approach to address the sparsity issue is to fill in the missing values with some imputed values such as the average rating of the users or items. However, this technique can distort the data and lead to an undefined factorization [Kor08, KY05, SKKR00]. Alternatively, the formulation can be reformulated as an optimization problem in which the squared error of the factorization is optimized *only over the observed ratings of the matrix.* Simon Funk introduced this idea in his blog post [Fun06] for the Netflix Prize.

### 3.4.3 Basic matrix factorization

The basic model of matrix factorization involves decomposing the rating matrix $\mathbf{R} \in \mathbb{R}^{m \times n}$ of rank $k \ll \min\{m, n\}$ into a matrix $\mathbf{P} \in \mathbb{R}^{m \times k}$ and another matrix $\mathbf{Q} \in \mathbb{R}^{n \times k}$, as demonstrated in Equation (3.31).

$$\mathbf{R} \approx \mathbf{P}\mathbf{Q}^{\top} \tag{3.31}$$

Therefore, $\mathbf{P}$ and $\mathbf{Q}$ represent the users and items' latent factor matrices, respectively. The parameter $k$ represents the predetermined number of latent factors. A row of $\mathbf{P}$, denoted as $\mathbf{p}_u \in \mathbb{R}^k$, represents a user, while a column of $\mathbf{Q}$, denoted as $\mathbf{q}_i \in \mathbb{R}^k$, represents an item. Thus, each user $u$ and item $i$ has an associated vector $\mathbf{p}_u$ and $\mathbf{q}_i$, respectively. The latent factors represent the inferred characteristics of the rating patterns, and each value in the vectors $\mathbf{p}_u$ and $\mathbf{q}_i$ indicates the affinity of each user and item to each of the $k$ factors.

Figure 3.3 provides an illustration of a basic matrix factorization model, in which users and items (movies) are projected into a joint latent space. A missing rating (i.e., a shaded cell in the figure) is predicted by computing the inner product of the user's learned factors and the corresponding item.

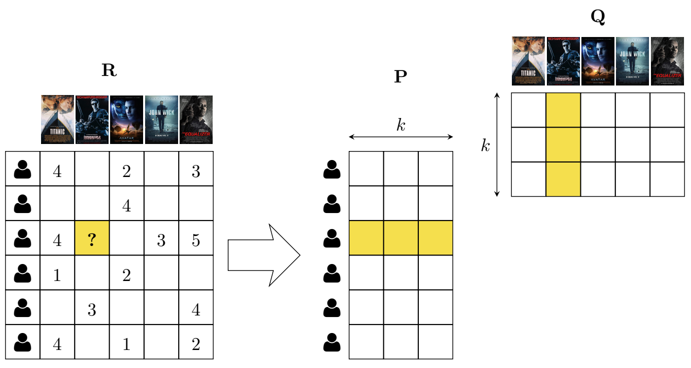


FIGURE 3.3: Decomposition of $\mathbf{R}$ into two lower-ranking matrices $\mathbf{P}$ and $\mathbf{Q}$.

### 3.4.4 Unbiased matrix factorization: SVD$_{\text{UNBIASED}}$

The unconstrained case is the most fundamental form of matrix factorization, where the factor matrices $\mathbf{P}$ and $\mathbf{Q}$ have no restrictions. For a fully dense matrix with observed ratings, we seek the factor matrices $\mathbf{P}$ and $\mathbf{Q}$ that approximately reconstruct the original matrix $\mathbf{R}$. This can be achieved by solving the following optimization problem:

$$\min S = \frac{1}{2}\|\mathbf{R} - \hat{\mathbf{R}}\|_{\mathrm{F}}^2 = \frac{1}{2}\|\mathbf{R} - \mathbf{P}\mathbf{Q}^{\top}\|_{\mathrm{F}}^2 \tag{3.32}$$

Subject to:

No restrictions on $\mathbf{P}$ and $\mathbf{Q}$

In the case of a matrix with missing entries, only a subset of the entries of matrix $\mathbf{R}$ are observed. Consequently, the loss function in Equation (3.32), becomes undefined. It is not possible to compute the Frobenius norm of a matrix that contains missing entries. Therefore, it is necessary to rewrite the loss function *solely based on the observed entries*, allowing the learning of latent factor matrices $\mathbf{P}$ and $\mathbf{Q}$. The advantage of this process is that once the latent factor matrices $\mathbf{P}$ and $\mathbf{Q}$ have been learned, the *entire ratings matrix can be reconstructed efficiently as* $\mathbf{PQ}^\top$*, in a single step.*

To achieve the factorization of the incomplete matrix $\mathbf{R}$ into the approximate product $\mathbf{PQ}^\top$, where $\mathbf{P}$ and $\mathbf{Q}$ are fully specified matrices, it becomes possible to predict all the entries in $\mathbf{R}$. More specifically, the prediction of the $(u, i)$-th entry of matrix $\mathbf{R}$ can be determined as follows:

$$\hat{r}_{ui} = \mathbf{p}_u \cdot \mathbf{q}_i^\top = \sum_{k=1}^{K} p_{uk} \cdot q_{ik} \tag{3.33}$$

Then, the modified loss function $S$, which works with incomplete matrices, *is computed only over the observed ratings in* $\mathcal{D}$ as shown in Equation (3.34).

$$S(\mathbf{P}, \mathbf{Q}) = \sum_{(u,i)\in\mathcal{D}} (r_{ui} - \mathbf{p}_u \cdot \mathbf{q}_i^\top)^2 \tag{3.34}$$

We can avoid overfitting by adding a $\lambda$ parameter to our loss function:

$$S(\mathbf{P}, \mathbf{Q}) = \sum_{(u,i)\in\mathcal{D}} (r_{ui} - \mathbf{p}_u \cdot \mathbf{q}_i^\top)^2 + \lambda(\|\mathbf{p}_u^2\| + \|\mathbf{q}_i^2\|) \tag{3.35}$$

The new parameter $\lambda$ is used to control the magnitudes of vectors $\mathbf{p}_u$ and $\mathbf{q}_i$. In particular, it reduces the size of possible large number values in a vector.

The subsequent phase of the approach involves determining the means to acquire matrices $\mathbf{P}$ and $\mathbf{Q}$. One approach to address this challenge entails initializing the two matrices with random values and evaluating the dissimilarity between their product and $\mathbf{R}$. Subsequently, an iterative process is employed to minimize this dissimilarity. This numerical approximation is the SGD technique, mentioned in Section 2.2.1.7, which strives to identify a local minimum for the dissimilarity. Specifically, the dissimilarity corresponds to the error between the actual rating and the predicted rating, and it can be computed for each user-item pair using Equation (3.36).

$$\text{err}_{ui} = (r_{ui} - \hat{r}_{ui})^2 = (r_{ui} - \sum_{k=1}^{K} p_{uk} \cdot q_{ik})^2 \tag{3.36}$$

### 3.4.4.1 Optimization through Stochastic Gradient Descent

For the set of parameters $\mathbf{\Theta} = (\mathbf{P}, \mathbf{Q})$ of the model we obtain the optimal values $\hat{\mathbf{\Theta}} = (\mathbf{P}^*, \mathbf{Q}^*)$ such that:

$$\hat{\mathbf{\Theta}} = \min_{\mathbf{\Theta}} S(\mathbf{P}, \mathbf{Q}) \tag{3.37}$$

$$\hat{\mathbf{\Theta}} = \min_{(\mathbf{P},\mathbf{Q})} \frac{1}{2} \sum_{(u,i)\in D} \left[r_{ui} - \left(\mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]^2 + \frac{\lambda}{2}\left(\|\mathbf{p}_u\|^2 + \|\mathbf{q}_i\|^2\right) \tag{3.38}$$

To minimize the loss function given in Equation (3.38), the partial derivatives of $S(\mathbf{P}, \mathbf{Q})$ with respect to $\mathbf{p}_u$ and $\mathbf{q}_i$ are calculated. To find $\frac{\partial S(\mathbf{P},\mathbf{Q})}{\partial \mathbf{p}_u}$, we differentiate each term with respect to $\mathbf{p}_u$. Differentiating the first term:

$$\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{1}{2}\left[r_{ui} - \left(\mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]^2\right] = -\left[r_{ui} - \left(\mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]\mathbf{q}_i$$

Differentiating the second term[1]:

$$\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{\lambda}{2}\|\mathbf{p}_u\|^2\right] = \lambda \mathbf{p}_u \tag{3.39}$$

Now, summing up the derivatives of each term, we have:

$$\frac{\partial S(\mathbf{P}, \mathbf{Q})}{\partial \mathbf{p}_u} = -\left[r_{ui} - \left(\mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]\mathbf{q}_i + \lambda \mathbf{p}_u = -\text{err}_{ui} \cdot \mathbf{q}_i + \lambda \mathbf{p}_u \tag{3.40}$$

To find $\frac{\partial S(\mathbf{P},\mathbf{Q})}{\partial \mathbf{q}_i}$, we differentiate each term with respect to $\mathbf{q}_i$. Differentiating the first term:

[1]Refer to Appendix B.5 for a detailed calculation of Equations (3.39) and (3.41).

$$\frac{\partial}{\partial \mathbf{q}_i}\left[\frac{1}{2}\left(\mathbf{p}_u \cdot \mathbf{q}_i^\top\right)^2\right] = -\left[r_{ui} - \left(\mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]\mathbf{p}_u$$

Differentiating the second term:

$$\frac{\partial}{\partial \mathbf{q}_i}\left[\frac{\lambda}{2}\|\mathbf{q}_i\|^2\right] = \lambda \mathbf{q}_i \tag{3.41}$$

Now, summing up the derivatives of each term, we have:

$$\frac{\partial S(\mathbf{P}, \mathbf{Q})}{\partial \mathbf{q}_i} = -\left[r_{ui} - \left(\mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]\mathbf{p}_u + \lambda \mathbf{q}_i = -\mathrm{err}_{ui} \cdot \mathbf{p}_u + \lambda \mathbf{q}_i \tag{3.42}$$

As described in Section 2.2.1.7, in order to minimize a loss function, we can adapt the general update equation $\mathbf{\Theta}_{k+1} \leftarrow \mathbf{\Theta}_k - \gamma_k \nabla \delta_{i_k}(\mathbf{\Theta}_k)$ for each parameter $\Theta$ belonging to $\mathbf{\Theta}$:

$$\Theta \leftarrow \Theta - \gamma \frac{\partial S(f_{\mathbf{\Theta}}(u, i), r_{ui})}{\partial \Theta}$$

By computing the gradient of the loss function with respect to the parameters $\mathbf{\Theta}$, we can determine the direction of steepest ascent. To apply this, we perform separate updates for each parameter:

$$\mathbf{p}_u \leftarrow \mathbf{p}_u + \gamma(\mathrm{err}_{ui} \cdot \mathbf{q}_i - \lambda \mathbf{p}_u) \tag{3.43}$$

$$\mathbf{q}_i \leftarrow \mathbf{q}_i + \gamma(\mathrm{err}_{ui} \cdot \mathbf{p}_u - \lambda \mathbf{q}_i) \tag{3.44}$$

In these equations, $\gamma$ represents the learning rate, $\mathrm{err}_{ui}$ is the prediction error for the user-item pair $(u, i)$ and $\lambda$ is the regularization parameter. These updates are performed iteratively during the training process to minimize the cost function and improve the accuracy of the model.

To initiate the rating prediction process, we begin by randomly initializing the matrices $\mathbf{P}$ and $\mathbf{Q}$. These matrices are utilized to compute the predicted ratings matrix $\hat{\mathbf{R}}$ through dot product computation. The error between the predicted ratings and the actual ratings is then calculated employing Equation (3.36). Subsequently, we iterate through each rating $r_{ui}$ in the training set, predicting $\hat{r}_{ui}$ and calculating the corresponding error $\mathrm{err}_{ui}$. The user and item feature vectors $\mathbf{p}_u$ and $\mathbf{q}_i$ are updated based on the prediction error, employing a learning rate $\gamma$. This iterative process of prediction and updating continues until a satisfactory approximation of the original ratings matrix $\mathbf{R}$ is attained. Upon completion of the training phase, the model enables the prediction of ratings for any user-item pair, even in the absence of original ratings (missing data positions). This prediction is accomplished using Equation (3.33). As a result, the model offers a comprehensive depiction of how any user would rate any item, based on their past ratings and those of similar users. Algorithm 11 provides a detailed description of this process.

**Algorithm 11:** SVD$_{\text{UNBIASED}}$

```
Input: Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$, regularization penalty $\lambda$, number of epochs $n$,
       learning rate $\gamma$ and the number of latent factors $K$.
Output: $\mathbf{P}^*$, $\mathbf{Q}^*$, the user and item feature matrices, respectively and the optimal
        number of epochs $n^*$.
Construct a training set $\mathcal{D}_{\text{TRAIN}}$
Construct a validation set $\mathcal{D}_{\text{VAL}}$
Randomly initialize $\mathbf{P}$ and $\mathbf{Q}$.
$n \leftarrow 0$
loop until the terminal condition is met. One epoch:
    $n \leftarrow n + 1$
    Randomly shuffle observed ratings in $\mathcal{D}_{\text{TRAIN}}$
    foreach $(u, i) \in \mathcal{D}_{\text{TRAIN}}$ in shuffled order do
        $\hat{r}_{ui} = \sum_{k=1}^{K} q_{ik} \cdot p_{uk}$
        $\text{err}_{ui} = r_{ui} - \hat{r}_{ui}$
        /* Update variables according to Equations (3.40) and (3.42)    */
        foreach $k \in \{1, \ldots, K\}$ do
            $p_{uk}^{+} = p_{uk} + \gamma(\text{err}_{ui} \cdot q_{ik} - \lambda p_{uk})$
            $q_{ik}^{+} = q_{ik} + \gamma(\text{err}_{ui} \cdot p_{uk} - \lambda q_{ik})$
            $p_{uk} = p_{uk}^{+}$
            $q_{ik} = q_{ik}^{+}$
        end foreach
    end foreach
    Compute the RMSE on $\mathcal{D}_{\text{VAL}}$ using Equation (2.12)
    if the RMSE on $\mathcal{D}_{\text{VAL}}$ was better than in any previous epoch:
        $\mathbf{P}^* = \mathbf{P}$, $\mathbf{Q}^* = \mathbf{Q}$, $n^* = n$
    end
    Terminal condition: RMSE on $\mathcal{D}_{\text{VAL}}$ does not decrease.
end
```

### 3.4.5 Biased matrix factorization: SVD$_{\text{BIASED}}$

Matrix factorization has been widely used in model-based RS, and researchers have proposed several adaptations of this technique. One significant adaptation, introduced by [Pat07], involves an optimized version of Simon Funk's algorithm. This adaptation incorporates the Baseline Predictor $b_{ui}$ into the prediction function $\hat{r}_{ui}$. The prediction function combines the global mean rating, user bias and item bias with the dot product of the user and item latent factor vectors:

$$\hat{r}_{ui} = \bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top \tag{3.45}$$

Subsequently, our loss function is defined as:

$$S(b_u, b_i, \mathbf{P}, \mathbf{Q}) = \frac{1}{2} \sum_{(u,i)\in D} \left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]^2 + \frac{\lambda}{2}\left(b_u^2 + b_i^2 + \|\mathbf{p}_u\|^2 + \|\mathbf{q}_i\|^2\right) \tag{3.46}$$

The error between the actual rating and the predicted rating can be computed for each user-item pair using Equation (3.47).

$$\text{err}_{ui} = (r_{ui} - \hat{r}_{ui})^2 = [r_{ui} - (\bar{r} + b_u + b_i + \sum_{k=1}^{K} p_{uk} \cdot q_{ik})]^2 \tag{3.47}$$

#### 3.4.5.1 Optimization through Stochastic Gradient Descent

To determine the optimal values of the model's parameters $\mathbf{\Theta} = (b_u, b_i, \mathbf{P}, \mathbf{Q})$, we aim to find the corresponding values $\hat{\mathbf{\Theta}} = (b_u^*, b_i^*, \mathbf{P}^*, \mathbf{Q}^*)$. This can be achieved by solving the following minimization problem:

$$\hat{\mathbf{\Theta}} = \min_{(b_u, b_i, \mathbf{P}, \mathbf{Q})} \frac{1}{2} \sum_{(u,i)\in D} \left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]^2 + \frac{\lambda}{2}\left(b_u^2 + b_i^2 + \|\mathbf{p}_u\|^2 + \|\mathbf{q}_i\|^2\right) \tag{3.48}$$

To minimize the loss function given in Equation (3.48), the partial derivatives of $S(b_u, b_i, \mathbf{P}, \mathbf{Q})$ with respect to $b_u, b_i, \mathbf{p}_u$ and $\mathbf{q}_i$ are calculated. To find $\frac{\partial S(b_u,b_i,\mathbf{P},\mathbf{Q})}{\partial b_u}$, we differentiate each term separately. Differentiating the first term:

$$\frac{\partial}{\partial b_u}\left[\frac{1}{2}\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]^2\right] = -\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]$$

Differentiating the second term:

$$\frac{\partial}{\partial b_u}\left[\frac{\lambda}{2} b_u^2\right] = \lambda b_u$$

Now, summing up the derivatives of each term, we have:

$$\frac{\partial S(b_u, b_i, \mathbf{P}, \mathbf{Q})}{\partial b_u} = -\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right] + \lambda b_u = -\mathrm{err}_{ui} + \lambda b_u \tag{3.49}$$

Similarly, to find $\frac{\partial S(b_u,b_i,\mathbf{P},\mathbf{Q})}{\partial b_i}$, we differentiate each term with respect to $b_i$. Differentiating the first term:

$$\frac{\partial}{\partial b_i}\left[\frac{1}{2}\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]^2\right] = -\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]$$

Differentiating the second term:

$$\frac{\partial}{\partial b_i}\left[\frac{\lambda}{2} b_i^2\right] = \lambda b_i$$

Now, summing up the derivatives of each term, we have:

$$\frac{\partial S(b_u, b_i, \mathbf{P}, \mathbf{Q})}{\partial b_i} = -\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right] + \lambda b_i = -\mathrm{err}_{ui} + \lambda b_i \tag{3.50}$$

To find $\frac{\partial S(b_u,b_i,\mathbf{P},\mathbf{Q})}{\partial \mathbf{q}_i}$, we differentiate each term with respect to $\mathbf{q}_i$. Differentiating the first term:

$$\frac{\partial}{\partial \mathbf{q}_i}\left[\frac{1}{2}\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]^2\right] = -\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]\mathbf{p}_u$$

Differentiating the second term:

$$\frac{\partial}{\partial \mathbf{q}_i}\left[\frac{\lambda}{2}\|\mathbf{q}_i\|^2\right] = \lambda \mathbf{q}_i$$

Now, summing up the derivatives of each term, we have:

$$\frac{\partial S(b_u, b_i, \mathbf{P}, \mathbf{Q})}{\partial \mathbf{q}_i} = -\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]\mathbf{p}_u + \lambda \mathbf{q}_i = -\text{err}_{ui} \cdot \mathbf{p}_u + \lambda \mathbf{q}_i \quad (3.51)$$

To find $\frac{\partial S(b_u,b_i,\mathbf{P},\mathbf{Q})}{\partial \mathbf{p}_u}$, we differentiate each term with respect to $\mathbf{p}_u$. Differentiating the first term:

$$\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{1}{2}\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]^2\right] = -\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]\mathbf{q}_i$$

Differentiating the second term:

$$\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{\lambda}{2}\|\mathbf{p}_u\|^2\right] = \lambda \mathbf{p}_u$$

Now, summing up the derivatives of each term, we have:

$$\frac{\partial S(b_u, b_i, \mathbf{P}, \mathbf{Q})}{\partial \mathbf{p}_u} = -\left[r_{ui} - \left(\bar{r} + b_u + b_i + \mathbf{p}_u \cdot \mathbf{q}_i^\top\right)\right]\mathbf{q}_i + \lambda \mathbf{p}_u = -\text{err}_{ui} \cdot \mathbf{q}_i + \lambda \mathbf{p}_u \quad (3.52)$$

To ascertain the direction of steepest ascent, we compute the gradient $\frac{\partial S(\mathbf{\Theta})}{\partial \Theta}$ of the loss function with respect to the parameters $\mathbf{\Theta}$. To implement this, we execute individual updates for each parameter:

$$b_u \leftarrow b_u + \gamma(\text{err}_{ui} - \lambda b_u) \quad (3.53)$$

$$b_i \leftarrow b_i + \gamma(\text{err}_{ui} - \lambda b_i) \quad (3.54)$$

$$\mathbf{q}_i \leftarrow \mathbf{q}_i + \gamma(\text{err}_{ui} \cdot \mathbf{p}_u - \lambda \mathbf{q}_i) \quad (3.55)$$

$$\mathbf{p}_u \leftarrow \mathbf{p}_u + \gamma(\text{err}_{ui} \cdot \mathbf{q}_i - \lambda \mathbf{p}_u) \quad (3.56)$$

See Algorithm 12 for the training algorithm.

**Algorithm 12:** SVD$_{\text{BIASED}}$

**Input:** Matrix $\mathbf{R}$, set of ratings $\mathcal{D}$, regularization penalty $\lambda$, number of epochs $n$, learning rate $\gamma$ and the number of latent factors $K$.

**Output:** $\mathbf{P}^*$, $\mathbf{Q}^*$, $\mathbf{b}_u^*$, $\mathbf{b}_i^*$, the user and item feature matrices and biases, respectively and the optimal number of epochs $n^*$.

```
Construct a training set $\mathcal{D}_{\text{TRAIN}}$
Construct a validation set $\mathcal{D}_{\text{VAL}}$
Randomly initialize $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{b}_u$, $\mathbf{b}_i$
Initialize $\bar{r} \leftarrow$ global mean of ratings
$n \leftarrow 0$
loop until the terminal condition is met. One epoch:
    $n \leftarrow n + 1$
    Randomly shuffle observed ratings in $\mathcal{D}_{\text{TRAIN}}$
    foreach $(u, i) \in \mathcal{D}_{\text{TRAIN}}$ in shuffled order do
        $\hat{r}_{ui} = \bar{r} + b_u + b_i + \sum_{k=1}^{K} q_{ik} \cdot p_{uk}$
        $\text{err}_{ui} = r_{ui} - \hat{r}_{ui}$
        /* Update variables according to Equations (3.49), (3.50), (3.51) and (3.52)   */
        foreach $k \in \{1, \ldots, K\}$ do
            $p_{uk}^{+} = p_{uk} + \gamma(\text{err}_{ui} \cdot q_{ik} - \lambda p_{uk})$
            $q_{ik}^{+} = q_{ik} + \gamma(\text{err}_{ui} \cdot p_{uk} - \lambda q_{ik})$
            $p_{uk} = p_{uk}^{+}$
            $q_{ik} = q_{ik}^{+}$
        end foreach
        $b_u = b_u + \gamma(\text{err}_{ui} - \lambda b_u)$
        $b_i = b_i + \gamma(\text{err}_{ui} - \lambda b_i)$
    end foreach
    Compute the RMSE on $\mathcal{D}_{\text{VAL}}$ using Equation (2.12)
    if the RMSE on $\mathcal{D}_{\text{VAL}}$ was better than in any previous epoch:
        $\mathbf{P}^* = \mathbf{P}$, $\mathbf{Q}^* = \mathbf{Q}$, $\mathbf{b}_u^* = \mathbf{b}_u$, $\mathbf{b}_i^* = \mathbf{b}_i$, $n^* = n$
    end
    Terminal condition: RMSE on $\mathcal{D}_{\text{VAL}}$ does not decrease.
end
```

## 3.5 Evaluation of recommendation algorithms

Evaluation of a RS can be done in three primary configurations. The first involves conducting an **in-person assessment** with multiple users, where they interact directly with the system by providing ratings or comments. This approach provides the most precise evaluation, as it directly captures the explicit needs of the users. However, it can be costly and time-consuming for users.

The second configuration, which we will use in this work, is **offline evaluation**. Here, we have a dataset of past interactions, such as all the ratings that users have given to items, stored in a matrix. We use this data to train predictive models that simulate user behavior and compute different metrics. This process is carried out offline, before users use the system. The models are periodically updated to incorporate information from new users and items, as well as the ratings that users give to them.

The third configuration involves **online evaluation**. In this case, recommendations are made to users by verifying the models trained in the offline process. This part is carried out when users request a recommendation, online. The process differs for new and old users, as new users have not rated any item, and their preferences are unknown to the system. The choice of evaluation configuration depends on factors such as precision, cost and availability of user data. Each approach has its advantages and disadvantages and the most appropriate approach should be selected based on the specific requirements of the recommendation system and the available resources. The evaluation process is an essential component of any RS, as it helps to ensure that the system is effective, efficient and meets the needs of its users [GSB$^+$16].

In the context of RS, we often split the dataset of all ratings $\mathcal{D}$ into $k$[2] non-overlapping subsets. In each round of the $k$-fold cross validation process, one fold is used for validation, denoted as $\mathcal{D}_{\text{VAL}}$, while the remaining $k-1$ folds are merged to form a training set $\mathcal{D}_{\text{TRAIN}}$. This process is repeated $k$ times, with each fold being used once for validation. As shown in Figure 3.4, which illustrates the 5-fold process, the training set is used to *fit* a model using a learning algorithm $\mathcal{A}$ with *fixed* hyperparameters $\boldsymbol{\lambda}$. Next, the model is evaluated using the validation set. The procedure is repeated $k$ times, resulting in $k$ models that are trained on distinct yet partly overlapping training sets and evaluated on non-overlapping validation sets. Finally, to estimate the performance of the model, the performance estimates from each of the $k$ validation sets are averaged. This provides an unbiased estimate of the model's performance on new data.

[2]In this context, the symbol $k$ is used to represent the number of folds.

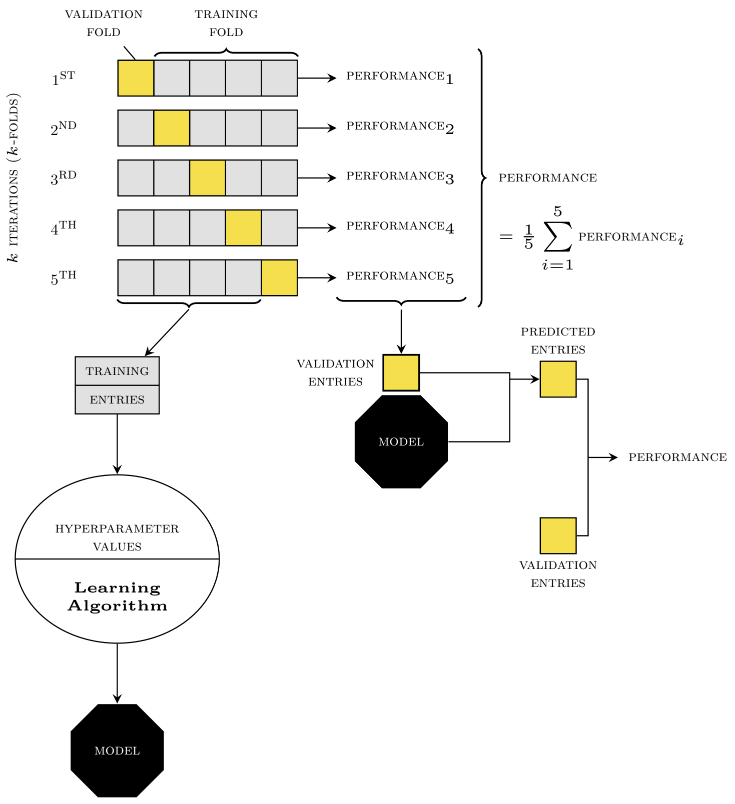


FIGURE 3.4: 5-folds CV procedure for model evaluation (Reference:[Ras20]).

## 3.5.1 Metrics for prediction accuracy

When our primary objective involves predicting ratings for unrated items, it is essential to assess the accuracy of these predictions. This quantifiable measure evaluates the proximity between the predicted ratings for recommended items, which are computed using a training subset of a dataset and the actual ratings of the items within the remaining validation subset of the same dataset. To further illustrate its significance, we revisit this metric, previously introduced as Equation (2.12):

$$\mathrm{RMSE}(f \mid \mathcal{D}) = \sqrt{\frac{1}{|\mathcal{D}|} \sum_{r_{ui} \in \mathcal{D}} (r_{ui} - \hat{r}_{ui})^2}$$

This metric fails to sufficiently capture the user interaction within a RS [JRTZ16]. Arguably, the key aspect of a RS lies in its ability to effectively rank diverse items for a specific user based on their preferences. In order to more accurately depict the user-system interaction, alternative precision-oriented metrics are occasionally employed to offer a more insightful perspective. In the following section, we will explore these metrics in detail.

### 3.5.2 Metrics for classification accuracy

Classification accuracy metrics evaluate the recommendation capabilities of a RS, contrasting with rating prediction metrics that assess the accuracy of predicted ratings. These metrics compare the system's recommended item list to the user's true preferences, which are defined differently based on the available rating data. Implicit rating systems, which utilize user actions like clicks as preference indicators, consider an item relevant if the user clicks on it. Conversely, explicit rating systems, where users provide ratings, deem an item relevant if the user's rating exceeds a predefined threshold. In this context, each rating is classified as either relevant or irrelevant, with the aim of exclusively recommending relevant items and avoiding irrelevant ones. Once the relevant item set is determined and recommendations are generated, the subsequent step involves computing a confusion matrix for each user. The matrix, presented as a two-by-two table, quantifies the recommended and non-recommended relevant items, as well as the recommended and non-recommended irrelevant items. This confusion matrix serves as a valuable tool for evaluating RS performance and identifying areas for improvement. Through analysis of the matrix, various classification metrics such as Precision, Recall and F1-score can be calculated. These metrics offer insights into the accuracy and completeness of the recommendations, and they are extensively employed for RS evaluation. Furthermore, they can be customized to suit the specific requirements of a given application.

#### 3.5.2.1 Confusion matrix

| | Relevant | Irrelevant |
|---|---|---|
| Recommended | TP | FP |
| Not Recommended | FN | TN |

FIGURE 3.5: Confusion matrix for RS.

- Precision: $\frac{tp}{tp+fp}$ - percentage of items that are relevant.
- Recall: $\frac{tp}{tp+fn}$ - percentage of relevant items that are recommended.
- False positive rate: $\frac{fp}{fp+tn}$ - percentage of irrelevant items that are recommended.

- Specificity: $\frac{tp}{tp+fn}$ - percentage of irrelevant items that are not recommended.

Since the algorithm recommends $k$[3] items, we denote by $\mathcal{B}_u(k)$ the set of items recommended for the user $u$. Next, $\mathcal{C}_u$ as the set of relevant items among all the items for the user $u$ and, by $\mathcal{D}_u$, the set of users that have interactions in the set $\mathcal{D}_{\text{VAL}}$. **Precision** is defined as the fraction of recommended items that are relevant to users or the fraction of recommended relevant items in the list. **Recall** is defined as the fraction of recommended items that are relevant over all relevant items. In other words, it measures what fraction of the relevant items have been retrieved in the set of recommendations:

$$\text{Precision@}k \stackrel{\text{def}}{=} \frac{1}{|\mathcal{D}_u|} \sum_{u \in \mathcal{D}_u} \text{Precision}_u\text{@}k = \frac{1}{|\mathcal{D}_u|} \sum_{u \in \mathcal{D}_u} \frac{|\mathcal{B}_u(k) \cap \mathcal{C}_u|}{|\mathcal{B}_u(k)|} \tag{3.57}$$

$$\text{Recall@}k \stackrel{\text{def}}{=} \frac{1}{|\mathcal{D}_u|} \sum_{u \in \mathcal{D}_u} \text{Recall}_u\text{@}k = \frac{1}{|\mathcal{D}_u|} \sum_{u \in \mathcal{D}_u} \frac{|\mathcal{B}_u(k) \cap \mathcal{C}_u|}{|\mathcal{C}_u|} \tag{3.58}$$

Let us consider a user $u$ with a set of relevant items consisting of 5 items: $\mathcal{C}(u) = \{i_1, i_3, i_4, i_6, i_8\}$. If the list is of size $k = 3$, we recommend $\mathcal{B}(3) = \{i_1, i_2, i_3\}$ to this user, then the Precision score is:

$$\frac{|\{i_1, i_2, i_3\} \cap \{i_1, i_3, i_4, i_9, i_6\}}{|\{i_1, i_2, i_3\}|} = \frac{2}{3}$$

And Recall is:

$$\frac{|\{i_1, i_3, i_4, i_9, i_6\} \cap \{i_1, i_2, i_3\}|}{|\{i_1, i_3, i_4, i_9, i_6\}} = \frac{2}{5}$$

Commonly, precision is expressed as precision in $k$ where $k$ is the length of the list of recommended items, for example, Precision@1 $= 1$ would indicate that an item was recommended, and the item was considered a true recommendation, that is, relevant. Then, Precision@2 $= 0.5$ would indicate that two items were recommended and one of them was considered a true positive, etc. Precision and Recall are always defined in the range $[0, 1]$.

[3]In this context, the symbol $k$ is used to represent the number of items or recommendations that are considered or evaluated for the calculation of the respective metric. Within model-based methods, the symbol $k$ is employed to represent the number of latent factors considered.

Precision and Recall are typically reported and analyzed together, as they tend to have an inverse relationship. Specifically, Precision refers to the proportion of recommended items that are actually relevant to a user, while Recall is the proportion of relevant items that are correctly recommended. These metrics are obtained by averaging the values across all users in the system. An algorithm can optimize Precision by recommending only a few items, but this would likely result in a low Recall, as many relevant items would not be recommended. Conversely, an algorithm can achieve a high Recall by recommending many items, but this would likely lead to a low Precision, as many irrelevant items would also be recommended. Therefore, an ideal algorithm aims to strike a balance between these two properties, recommending predominantly good items while also capturing almost all relevant items [KEK18]. To facilitate finding this balance, researchers often look at the F-Score, which is the harmonic mean between Precision and Recall.

$$\mathrm{F@}k \overset{\mathsf{def}}{=} \frac{1}{|\mathcal{D}_u|} \sum_{u \in \mathcal{D}_u} 2 \cdot \frac{\mathrm{Precision}_u@k \cdot \mathrm{Recall}_u@k}{\mathrm{Precision}_u@k + \mathrm{Recall}_u@k} \tag{3.59}$$

The F-Score takes into account both, Precision and Recall, with greater weight given to whichever metric has a lower value. As such, it is a useful summary measure for evaluating the overall performance of a RS. The higher these three metrics (Precision, Recall, and F-Score), the better the RS.

### 3.5.3 Metrics for ranking accuracy

In RS the user generally receives a predicted, ordered and classified list of recommendations that contains the top-$k$ items. In the best case, the list should have the most preferred items at the top. Therefore, recommendations can be studied as a *ranking problem* [CKT10]. One drawback that arises with the Precision metric is that while it rewards the algorithm for recommending relevant items, it does not take into account where they are on the recommendation.

In general, we want the relevant items to appear first in the list. To evaluate this, we can use the average mean precision metric, which is the mean of the average precision over each user. By taking the mean average precision we give more importance to the first items in the list than to the last ones, since the first item is used in all the Precision@$k$ calculations, while the last is only used in one. Average precision takes the average of the Precision@$k$ in each of the relevant items in the recommendation.

## Mean average precision

The Mean Average Precision at $K$ (MAP@$K$) is a crucial metric for evaluating the ranking quality in RS. It measures the mean value across $Q$ profiles within the test set. Each profile represents a specific user or query for which recommendations are generated. $Q$ is used as a variable to denote the total number of profiles in the dataset. MAP@$K$ quantifies the precision at each rank, ranging from 1 to $K$, considering the fraction of correctly predicted items among the top $k$ recommendations. Its sensitivity to the order of correct results makes it a more dependable measure of ranking performance. To compute the Average Precision (AP@$K$) at position $k$ in the ordered list of recommendations, the following equation is used:

$$\text{AP@}K = \frac{\sum_{k=1}^{K} (\text{Precision@}k \times \text{rel}(k))}{\#\text{ relevant in } K} \tag{3.60}$$

Here, Precision@$k$ is the precision calculated at position $k$ of the ordered list, and rel@$k$ is a binary function with a value of 1 if the item at position $k$ is relevant. The MAP@$K$ for a set of users receiving recommendations is the average of the individual average precision (AP@$k$) values for each user's recommendation. It is represented as:

$$\text{MAP@}K = \frac{1}{Q} \sum_{q=1}^{Q} \left( \frac{1}{K} \sum_{k=1}^{K} \text{precision}_q(k) \right) \tag{3.61}$$

For each user in the recommendation set, a classification of items is generated based on their interactions. The MAP@$k$ is computed by considering different cutoff points $k$, and its value always falls within the range $[0, 1]$. This comprehensive metric captures the overall performance of the RS, providing valuable insights into its effectiveness in generating relevant recommendations for various users or queries.

We now provide a the step-by-step procedure to compute MAP@$K$.

- Set a range threshold $k$
- Compute the relevant percentage in the top-$k$
- Ignore items classified below $k$
- Query 1: ■□■□□■

  - Precision@1: $\frac{1}{1}$
  - Precision@3: $\frac{2}{3}$
  - Precision@6: $\frac{3}{6}$
  - AP@6 $= \frac{1}{3} \times (\frac{1}{1} + \frac{2}{3} + \frac{3}{6}) = 0.72$

- Query 2: ■■□■□□
  - Precision@1: $\frac{1}{1}$
  - Precision@2: $\frac{2}{2}$
  - Precision@4: $\frac{3}{4}$
  - AP@4 $= \frac{1}{3} \times (\frac{1}{1} + \frac{2}{2} + \frac{3}{4}) = 0.917$

In this way:

$$\text{MAP@}K = \frac{1}{2}(0.72 + 917) = 0.8185 \tag{3.62}$$

For a detailed step-by-step procedure to compute this metric, please refer to Annex C.4.

# Chapter 4

# Hyperparameter Optimization

The chapter is organized as follows. First, we provide a definition of the hyperparameter optimization problem in Section 4.1. In Section 4.4, we describe manual, grid, and random search techniques. Finally, Section 4.5 covers Bayesian optimization in detail.

## 4.1 Introduction

When faced with a ML problem, such as rating prediction, practitioners need to make various decisions, including selecting a suitable algorithm and its corresponding hyperparameters. This task of algorithm selection aims to identify the algorithm or a group of algorithms that are more likely to achieve superior performance on specific datasets and evaluation measures. These algorithms typically have two types of parameters: ordinary parameters and hyperparameters, which need to be manually set before initiating the model training process. Existing automated methods for adjusting hyperparameters are valuable when there are numerous hyperparameters, and it is challenging to adjust them manually due to a lack of understanding of their effects. This problem is exacerbated when hyperparameters are not independent and must be optimized together. In this section, we present diverse methods for autonomously identifying the optimal combination of algorithms and configurations. Our objective is to provide a comprehensive overview of contemporary hyperparameter optimization techniques, encompassing traditional grid search, random search, and advanced Bayesian optimization (BO). This comprehensive approach establishes the foundation for our experiments in Chapter 5.

## 4.2 Parameters vs. hyperparameters

Algorithm $\mathcal{A}$ is characterized by two main components: a vector $\boldsymbol{\Theta}$ of *model parameters* and a vector $\boldsymbol{\lambda}$, of *hyperparameters*. The model parameters, represented by $\boldsymbol{\Theta}$, are learned directly from data during the training phase. Meanwhile, the hyperparameters, denoted by $\boldsymbol{\lambda}$, influence how the algorithm learns the values for $\boldsymbol{\Theta}$. These hyperparameters can be manually specified or optimized during the validation phase. To illustrate this concept, we will examine the case of matrix factorization, as depicted in Algorithm 11. In this context, the model parameters $\boldsymbol{\Theta}$ correspond to the latent factors that capture user preferences and item characteristics. By factorizing the rating matrix $\mathbf{R}$, the algorithm *learns* the latent factors that best represent the underlying patterns in the data. The hyperparameters $\boldsymbol{\lambda}$ in this case can include the *number* of latent factors to be considered, the regularization term $\lambda$[1], and the learning rate $\gamma$. These hyperparameters *affect* how the matrix factorization algorithm learns the latent factors and influences the accuracy and generalization of the resulting RS. During the training phase, the goal is to estimate the optimal value of $\boldsymbol{\Theta}$, denoted as $\hat{\boldsymbol{\Theta}}$, which minimizes the given loss function defined in Equation (4.1).

$$\hat{\boldsymbol{\Theta}} = \min_{(\mathbf{P},\mathbf{Q})} \frac{1}{2} \sum_{(u,i)\in D} \left[ r_{ui} - \left( \mathbf{p}_u \cdot \mathbf{q}_i^\top \right) \right]^2 + \frac{\lambda}{2} \left( \|\mathbf{p}_u\|^2 + \|\mathbf{q}_i\|^2 \right) \tag{4.1}$$

Once the model is trained, it needs to be validated using a separate validation dataset or through techniques like cross-validation (see Algorithm 15). The corresponding loss function on the validation dataset provides an evaluation of the model's performance on unseen data. It is important to note that during the validation phase, the estimated $\hat{\boldsymbol{\Theta}}$ remains unchanged for each fold or validation set. It is worth mentioning that the hyperparameters $\boldsymbol{\lambda}$ are not estimated during the training phase but rather need to be specified *before* starting the training process. The choice of appropriate hyperparameter values greatly impacts the performance and effectiveness of the RS. In some cases, the loss function in matrix factorization can be formulated analytically, allowing for the estimation of $\hat{\boldsymbol{\Theta}}$ using gradient-based optimization methods, as discussed in Section 2.2.1.7. However, the optimization of the black box loss function, which depends on the hyperparameters $\boldsymbol{\lambda}$, may require derivative-free global optimization approaches, such as Bayesian optimization, to find the optimal hyperparameter values that maximize the performance on the validation dataset [AC19].

[1]In this context, the symbol $\lambda$ is specifically used to denote the regularization term, whereas $\boldsymbol{\lambda}$ is employed to represent the hyperparameters.

## 4.3 Hyperparameter optimization: An overview

In the context of hyperparameter optimization, where we have $n$ hyperparameters denoted as $\boldsymbol{\lambda}_1, \dots, \boldsymbol{\lambda}_n$, each with a domain range of 1 to $n$, the hyperparameter space can be defined as a subset of the Cartesian product of these domains, denoted as $\boldsymbol{\Lambda_1} \times \cdots \times \boldsymbol{\Lambda}_n$. It is important to note that this subset relation exists because certain settings of one hyperparameter can render other hyperparameters inactive. For instance, in the case of an artificial neural network (ANN), the parameters related to the specifics of the third layer become irrelevant if the network depth is set to one or two. Similarly, in the context of a support vector machine (SVM) with a polynomial kernel, the hyperparameters associated with the polynomial kernel become irrelevant if we decide to use a different kernel [AC19]. To formally express this relationship, we define a hyperparameter $\boldsymbol{\lambda}_o$ as being conditional on another hyperparameter $\boldsymbol{\lambda}_c$. In other words, $\boldsymbol{\lambda}_o$ is active only if hyperparameter $\boldsymbol{\lambda}_c$ takes values from a specific set $\mathcal{V}_o(c) \supset c$. In this case, we refer to $\boldsymbol{\lambda}_c$ as the parent of $\boldsymbol{\lambda}_o$.

Let us formally define the hyperparameter optimization task. A typical learning algorithm $\mathcal{A}$ incorporates adjustable hyperparameters $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$, which govern its behavior. By employing this learning algorithm, we can generate a model using the training data $\mathcal{D}_{\text{TRAIN}}$ as shown in Equation (4.2).

$$f_{\boldsymbol{\lambda}} = \mathcal{A}(\boldsymbol{\lambda}, \mathcal{D}_{\text{TRAIN}}) \tag{4.2}$$

Hereafter, we denote by $f_{\boldsymbol{\lambda}}$ *a model trained using hyperparameters* $\boldsymbol{\lambda}$, where $f_{\boldsymbol{\lambda}} : \mathbb{R}^d \to \mathcal{Y}$. The process of hyperparameter optimization aims to discover the *optimal* hyperparameters for the learning algorithm $\mathcal{A}$. By establishing the aforementioned $S : \boldsymbol{\Lambda} \to \mathbb{R}^+$ loss function, that assigns a numerical score to each possible configuration $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$, the objective of hyperparameter optimization is to identify the optimal configuration $\boldsymbol{\lambda}^\star$ that minimizes $R(f)$:

$$\boldsymbol{\lambda}^\star = \underset{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}}{\operatorname{argmin}} \, R(f) \tag{4.3}$$

$$\boldsymbol{\lambda}^\star = \underset{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}}{\operatorname{argmin}} \underset{(\mathbf{x},y)\sim\mathcal{P}_{\mathcal{T}}}{\mathbb{E}} S(f_{\boldsymbol{\lambda}}(u,i), r_{ui}) \tag{4.4}$$

Since the distribution $\mathcal{P}_{\mathcal{T}}$ is unknown, it is not feasible to find an exact solution for Equation (4.4). Thus, it is necessary to rely on empirical risk to estimate the true risk (generalization error). In this context, the objective function can be defined as the empirical risk error on a validation dataset:

$$\hat{R}_{\mathcal{D}_{\text{VAL}}}(f_{\boldsymbol{\lambda}} \mid S) = \frac{1}{|\mathcal{D}_{\text{VAL}}|} \sum_{(\mathbf{x},y)\in\mathcal{D}_{\text{VAL}}} S(f_{\boldsymbol{\lambda}}(u,i), r_{ui}) \tag{4.5}$$

In our case, $S$ represents the RMSE calculated through $k$-fold cross-validation. Then, the optimization problem can be expressed as follows:

$$\boldsymbol{\lambda}^{\star} = \underset{\boldsymbol{\lambda}\in\boldsymbol{\Lambda}}{\operatorname{argmin}}\, \hat{R}_{\mathcal{D}_{\text{VAL}}}(f_{\boldsymbol{\lambda}}, \mid S) \tag{4.6}$$

When dealing with a fixed dataset $D \sim \mathcal{D}$, the expectation can be approximated by averaging the function $S$ over various training-validation splits of $D$. In the case of extremely large datasets, a single training-validation split may be adequate. Consequently, the optimization process aims to minimize the validation loss $S$ across the hyperparameter space $\boldsymbol{\Lambda}$, denoted as $\hat{R}_{\mathcal{D}_{\text{VAL}}}(f_{\boldsymbol{\lambda}} \mid S)$, where $\boldsymbol{\lambda}$ represents the hyperparameters. For model-based methods employing a parameterized model $f_{\boldsymbol{\Theta}}(u,i)$ with an unknown parameter vector $\boldsymbol{\Theta}$, Equation (4.6) can be reformulated as follows:

$$\boldsymbol{\lambda}^{\star} = \underset{\boldsymbol{\lambda}\in\boldsymbol{\Lambda}}{\operatorname{argmin}}\, \hat{R}_{\mathcal{D}_{\text{VAL}}}(f_{\boldsymbol{\Theta}}^{\boldsymbol{\lambda}}, \mid S) \tag{4.7}$$

## 4.4 Standard techniques

Hyperparameter optimization methods can be broadly categorized into two categories: manual and automatic search. In manual search, sets of hyperparameters are tested by hand, which depends on the intuition and experience of expert users to identify hyperparameters that will have the greatest impact on the results. However, manual search requires a high level of expertise and practical experience, making it challenging to handle a large number of hyperparameters and a wide range of values. As the number of hyperparameters and the range of values increases, it becomes increasingly challenging for humans to handle high-dimensional data. The primary issue with hyperparameter optimization is that evaluating the objective function to find the best performance is computationally expensive. In automatic search methods, different hyperparameters are tested and evaluated iteratively, and the results are used to determine the next set of hyperparameters to test. This process can be automated and can save a considerable amount of time and effort. However, evaluating the objective function for each set of hyperparameters can be time-consuming, especially for complex models like deep neural networks. With a large number of hyperparameters, the optimization process becomes even more challenging. Therefore, there is a need for efficient and effective methods for hyperparameter optimization that can automate the process and reduce the computational cost.

### 4.4.1 Grid search

Manual search can be time-consuming and impractical, especially when dealing with complex models that require the tuning of multiple hyperparameters. To address this issue, automatic search algorithms such as *grid search* have been proposed. This approach involves an exhaustive search of all possible combinations of discrete hyperparameter values in the training set, with subsequent evaluation of performance against a predefined metric in a validation set. The best performing combination of hyperparameters is then returned. While grid search has been effective in optimizing hyperparameters for many ML models, its use in the context of RS has been limited due to the large number of hyperparameters involved. This can lead to a combinatorial explosion of possible configurations, making it computationally expensive to evaluate each one. Therefore, researchers have explored alternative approaches such as Bayesian optimization and random search that are more efficient and effective for finding optimal hyperparameters in the context of RS.

To illustrate the grid search procedure, let $n$ represent the number of hyperparameters. The grid search technique encompasses the following steps:

- For each hyperparameter $z_i (i \in \{1, 2, \dots, n\})$ define a list of all the values that can be assigned to $z_i$, let $l_i$ be the list.
- For each $n$-tuple $(v_1, \dots, v_n)$ where $\forall i \in \{1, 2, \dots, n\}$, $v_i \in l_i$:
  - Fit the model by assigning $v_i$ to $z_i$ for each $i \in \{1, 2, \dots, n\}$ and evaluate its performance.
- Choose the hyperparameter values that throw the best performance.

The number of times the model is fitted and tested is $|\prod_{i=1}^{n} l_i|$. As a result, it suffers from the Hughes effect [OMT+08] because the number of joint values (tuples) grows exponentially with the number of hyperparameters.

### 4.4.2 Random search

One major limitation of grid search is that its search order is heavily dependent on the implementation and it always selects the same limited set of values for each hyperparameter. To address this issue, the *random search* technique has been introduced. It extracts the value of each hyperparameter from a uniform distribution, allowing a much wider range of explored values. In terms of implementation cost, random search only requires the ability to uniformly extract from a range or list, making it computationally similar to grid search. The gain in performance from random search often offsets any increased implementation cost, as demonstrated by [BB12]. Let us now see an example of the random optimization of hyperparameters:

- For $k$ times:
  - Fit the model by randomly assigning values to the hyperparameters and then evaluate its performance.
- Choose the hyperparameter values that lead to the best performance.

We cannot assume that the assigned values are the most appropriate. However, this technique can be faster than grid search.

## 4.5 Bayesian optimization

Neither grid search nor random search leverage the information obtained from previous steps during the search for the optimal solution. To address this issue, Bayesian optimization [JSW98] can be employed. To illustrate this method let us consider a pharmaceutical company that is developing a new drug for a specific medical condition. The design space in this context comprises various parameters, including the chemical composition, dosage form, administration route and other factors that can impact the drug's efficacy and safety. Each proposed formulation can be synthesized and subjected to rigorous testing to assess its performance. The optimization problem in this scenario involves finding the optimal combination of parameters that minimizes the desired outcome, such as minimizing side effects or maximizing therapeutic effectiveness. This can be expressed as:

$$\mathbf{x}^\star = \operatorname*{argmin}_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}) \tag{4.8}$$

Here, $\mathbf{x}$ represents a vector that captures the parameters defining the drug formulation, and $f(\mathbf{x})$ denotes an objective function that quantifies the desired outcome based on the chosen parameters. Bayesian optimization is as a method to tackle optimization problems with specific challenges. In particular, it addresses problems characterized by the following properties:

- Black-box: The objective function $f$ can only be evaluated point-wise, without access to its derivatives. Although differentiability may be assumed, derivative information is not available.
- The evaluation of $f(\mathbf{x})$: For any given $\mathbf{x}$, is resource-intensive, often requiring time-consuming experiments or simulations to determine the objective function's value for a given set of parameters.
- Noisy evaluations: When evaluating the objective function $f$, the observed values may be subject to noise or uncertainty, requiring careful handling and statistical analysis.
- Global and non-convex: The goal is to find the global minimum of a non-convex function within a bounded domain $\mathcal{X}$.

By addressing these challenging properties, Bayesian optimization provides a powerful framework to navigate the complex parameter space and efficiently search for optimal drug formulations that meet specific therapeutic goals while considering limitations such as expensive and noisy evaluations.

### 4.5.1 Problem formalization

In typical scenarios, our objective is to determine the minimum of an unknown function $f(\mathbf{x})$, with $f : \mathcal{X} \to \mathbb{R}$:

$$\mathbf{x}^\star = \operatorname*{argmin}_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}) \tag{4.9}$$

In the specific context discussed in Section 4.3, $\mathbf{x}$ takes the form of $\boldsymbol{\lambda}$ i.e., $\mathbf{x} = \boldsymbol{\lambda}$, representing the hyperparameters targeted for optimization in Equation (4.6). By employing the Bayesian optimization framework, we can effectively tackle optimization problems and adapt the approach based on the specific scenario at hand, i.e., hyperparameter optimization.

Bayesian optimization is an iterative approach that aims to solve the aforementioned problem by leveraging all previously observed values of the unknown function $f(\boldsymbol{\lambda})$. This is achieved by constructing a *surrogate* model of $f(\boldsymbol{\lambda})$ that guides the selection of the next evaluation point $\boldsymbol{\lambda}'$, while considering both exploration and exploitation criteria. The fundamental principle of Bayesian optimization lies in effectively utilizing all available information to inform the optimization process at each iteration. To accomplish this, a history of observed points, denoted as $\mathcal{D} = \{\boldsymbol{\lambda}_i, f(\boldsymbol{\lambda}_i)\}_{i=1}^{n}$, is accumulated. This historical information plays a crucial role in guiding the exploration and optimization of the target function. In contrast, gradient descent can be viewed as an exploitative strategy that relies solely on local information, such as the current position $\boldsymbol{\lambda}_n$ and the local curvature.

### 4.5.2 Surrogate model

A surrogate model is a regression model using an algorithm $\mathcal{A}'$ (different from $\mathcal{A}$ whose hyperparameters need to be optimized) to approximate the objective function $S$. The data points used for training are already evaluated configurations. The surrogate model provides a posterior probability distribution that describes the potential values of $S(\boldsymbol{\lambda})$ in a candidate $\boldsymbol{\lambda}$ configuration. The idea here is that every time we look at $S$ at a new point $\boldsymbol{\lambda}$, we update this posterior distribution. Various models have been proposed as a surrogate model. A popular option is Gaussian Processes (GP).

#### 4.5.2.1 Gaussian Processes

A GP is a supervised learning model used primarily for regression problems. It is a distribution over functions, namely, from a set of data points the GP offers possible functions that fit these points, weighted by their probability [Ber19]. The shape and properties of the possible functions are defined by a covariance function. When predicting the value of an invisible point, the GP returns a normal distribution; the variance being an estimate of the uncertainty of the model at this point. The multi-point prediction will result in a joint Gaussian distribution. The GP is completely specified by a mean function $\mu(\boldsymbol{\lambda}) : \boldsymbol{\Lambda} \to \mathbb{R}$ and a defined positive covariance function also called the kernel, $k(\boldsymbol{\lambda}, \boldsymbol{\lambda}') : \boldsymbol{\Lambda}^2 \to \mathbb{R}$:

$$S(\boldsymbol{\lambda}) \sim \mathcal{GP}(\mu(\boldsymbol{\lambda});\ k(\boldsymbol{\lambda}, \boldsymbol{\lambda}')) \tag{4.10}$$

The most common choice for the GP kernel is the Square Exponential Kernel. The algorithm starts with an initial set of $k$ configurations $\{\boldsymbol{\lambda}_i\}_{i=1}^{k}$ and their associated function values $\{y_i\}_{i=1}^{k}$, with $y_i = S(\boldsymbol{\lambda}_i)$. In each iteration $n \in \{k+1, \ldots, N\}$ we update the GP model using Bayes' rule to obtain a posterior distribution conditional on the current training set $\mathcal{D}_n = \{(\boldsymbol{\lambda}, y_i)\}_{i=1}^{n}$ containing the configurations and observations evaluated in the past. For any potentially unexamined configuration $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ the posterior mean $\mu_n(\boldsymbol{\lambda})$ and the posterior variance $\sigma_n^2(\boldsymbol{\lambda})$ of the GP, conditioned by $\mathcal{D}_k$, are known in closed form:

$$\mu_n(\boldsymbol{\lambda}) = \mathbf{k}(\boldsymbol{\lambda})^\top (\mathbf{K} + \tau^2 \mathbf{I})^{-1} \mathbf{y} \tag{4.11}$$

$$\sigma_n^2(\boldsymbol{\lambda}) = k(\boldsymbol{\lambda}, \boldsymbol{\lambda}) - \mathbf{k}(\boldsymbol{\lambda})^\top (\mathbf{K} + \tau^2 \mathbf{I})^{-1} \mathbf{k}(\boldsymbol{\lambda}) \tag{4.12}$$

where $K$ is the $n \times n$ matrix whose entries are $K_{i,j} = k(\boldsymbol{\lambda}_i, \boldsymbol{\lambda}_j)$, $\mathbf{k}(\boldsymbol{\lambda})$ is the vector $n \times 1$ of the covariance terms between $\boldsymbol{\lambda}$ and $\{\boldsymbol{\lambda}_i\}_{i=1}^{n}$, $\mathbf{y}$ is the vector $n \times 1$ whose $i^{\text{th}}$ entry is $y_i$, and $\tau^2$ is the noise variance.

When a new point is selected and evaluated, $\boldsymbol{\lambda}_{n+1}$ provides a new observation $y_{n+1} = S(\boldsymbol{\lambda}_{n+1})$, so we can add the new pair $\{(\boldsymbol{\lambda}_{n+1}, y_{n+1})\}$ to the actual training set $\mathcal{D}_n$, getting a new training set for the next iteration $\mathcal{D}_{n+1} = \mathcal{D}_n \cup (\boldsymbol{\lambda}_{n+1}, y_{n+1})$

$$\boldsymbol{\lambda}_{n+1} = \underset{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}}{\operatorname{argmax}}\, a_n(\boldsymbol{\lambda}, \mathcal{D}_n) \tag{4.13}$$

where $a_n$ is the acquisition function to maximize.

#### 4.5.2.2 Covariance of the Gaussian Process

As mentioned above, the covariance function is one of the components that defines a GP. This describes the properties of the model, such as smoothness, noise, periodicity, etc. It is determined by a function called kernel $k(\boldsymbol{\lambda}, \boldsymbol{\lambda}')$ that measures the similarity between two points in the input space and a noise variation that is taken as normally distributed with the variance $\sigma_n^2$ and is independent for each variable. The kernel function is somewhat more complex. Different types of kernel have been designed. We must pay attention when choosing the right one. The Matérn kernels [RW05], are a widely used option in the literature for hyperparameter optimization. They are parametric functions parameterized by a smoothness parameter $\nu > 0$. A special case is the **square exponential kernel** (RBF) with $\nu \to \infty$ which is given by:

$$k_{\text{RBF}}(\boldsymbol{\lambda}, \boldsymbol{\lambda}') = \sigma_f^2 \exp\left(-\frac{1}{2l^2} \left\|\boldsymbol{\lambda} - \boldsymbol{\lambda}'\right\|^2\right) \tag{4.14}$$

With this kernel, the influence of a point on the value of another point decays exponentially with its relative distance. This implies that the Gaussian friction quickly returns to its previous one in areas without observed points. The parameter "$l$" represents a characteristic length scale that determines the proximity between points and their mutual influence. In simpler terms, it quantifies the extent to which points affect each other. The parameter $\sigma_f^2 = \nu$ is a signal variance. It tells us how much the function changes. If it is large the function has high frequency. Popular choices for the smoothness parameter are $\nu = 1.5$ and $\nu = 2.5$.

### 4.5.3 Acquisition function

In Bayesian optimization, the goal is to find the set of hyperparameters that will lead to the best performance of a model. The GP is a useful tool in this context because it provides an estimate of the performance of a model for each set of hyperparameters, as well as the uncertainty of the estimate. However, the question remains of how to choose which model to train. One approach is to choose the model that is slightly better than the current best model, where the GP generates little uncertainty. Another approach is to choose a model with high uncertainty but potentially high performance. This creates a trade-off between exploration and exploitation, where *exploitation focuses on improving the current best model*, while *exploration focuses on searching for potentially better models.* To determine which model to train, an acquisition function, denoted by "$a$", is used. Most acquisition functions focus on models that return the (Gaussian) distribution over $f$, which represents the function that maps the hyperparameters to the performance of the model. Three commonly used acquisition functions are the *Probability of Improvement* (PI), *Expected Improvement* (EI), and *Lower Confidence Bound* (LCB).

The PI function selects the hyperparameters that have the highest probability of improving the current best model. It is defined as the probability that the performance of a new model will be better than the current best model. The EI function selects the hyperparameters that have the highest expected improvement over the current best model. It is defined as the expected value of the improvement of a new model over the current best model. The LCB function selects the hyperparameters that have the lowest predicted value of the function $f$, plus a term that increases with the uncertainty of the estimate. This encourages exploration of new hyperparameters that could potentially lead to better performance. Overall, the choice of acquisition function depends on the trade-off between exploration and exploitation that is desired for a given problem. The PI function is more exploitative, while the EI and LCB functions are more exploratory. The optimal choice of acquisition function will depend on the specific problem and the goals of the optimization process. Below, we formally introduce these acquisition functions.

#### 4.5.3.1 Probability of improvement

The Probability of Improvement [Kus64], measures the probability that the value of a function at a point $\boldsymbol{\lambda}$ is less than a threshold $\tau$:

$$a_{\text{PI}}(\boldsymbol{\lambda}, \mathcal{D}_n) = \mathbb{E}_f \mathbb{1}(f < \tau) = p_{\mathcal{D}_n}(f < \tau) \tag{4.15}$$

where $\mathbb{1}$ is an indicator function of $(f < \tau)$. The distribution $p_{\mathcal{D}_n}(f < \tau)$ is given by the surrogate model that comes from the posterior distribution of $f$ at a point $\boldsymbol{\lambda}$.

A common distribution is the Gaussian for which PI has:

$$a_{\text{PI}}(\boldsymbol{\lambda}, \mathcal{D}_n) = \Phi\left(\frac{\tau - \mu(\boldsymbol{\lambda})}{\sigma(\boldsymbol{\lambda})}\right) \tag{4.16}$$

where $\Phi$ is the cumulative distribution function of the standard Gaussian distribution. $\boldsymbol{\lambda}$ represents a given set of hyperparameters and $\tau$ is the minimum error encountered so far. The functions $\mu(\boldsymbol{\lambda})$ and $\sigma(\boldsymbol{\lambda})$ are the mean and variance of the posterior predictive distribution, respectively.

#### 4.5.3.2 Expected improvement

Expected Improvement [SSW$^+$16], is the most prominent option for hyperparameter optimization and it is also the acquisition function that we will use in our work. Formally, the improvement for a $\boldsymbol{\lambda}$ hyperparameter setting is defined as:

$$I(\boldsymbol{\lambda}) = \max\{S^{\min} - Y, 0\} \tag{4.17}$$

where $S^{\min}$ is currently the best function value and $Y$ is a random variable that models our knowledge of the value of the $S$ function for the hyperparameter setting $\boldsymbol{\lambda}$, which depends on $\mathcal{D}_n$. The hyperparameter setting with the greatest expected improvement:

$$\mathrm{E}\left[I(\boldsymbol{\lambda})\right] = \mathrm{E}\left[\max\left\{S^{\min} - Y, 0\right\} \mid \mathcal{D}_n\right] \tag{4.18}$$

is chosen for the next evaluation. Supposing that $Y \sim \mathcal{N}(\mu(\mathcal{D}_{n+1}(\boldsymbol{\lambda}), \sigma^2(\mathcal{D}_n(\boldsymbol{\lambda})))$, the expected improvement can be formulated in closed form as:

$$a_{\mathrm{EI}}(\boldsymbol{\lambda}, \mathcal{D}_n) = \mathrm{E}\left[I(\boldsymbol{\lambda})\right] = \begin{cases} \sigma(\mathcal{D}_{n+1}(\boldsymbol{\lambda}))(Z \cdot \Phi(Z) + \phi(Z)), & \text{if } \sigma(\mathcal{D}_{n+1}(\boldsymbol{\lambda})) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{4.19}$$

where:

$$Z = \frac{S^{\min} - \mu(\mathcal{D}_{n+1}(\boldsymbol{\lambda}))}{\sigma(\mathcal{D}_{n+1}(\boldsymbol{\lambda}))} \tag{4.20}$$

where $\phi(\cdot)$ and $\Phi(\cdot)$ denote the standard normal density and the distribution function, and $\mu(\mathcal{D}_{n+1}(\boldsymbol{\lambda}))$ and $\sigma(\mathcal{D}_{n+1}(\boldsymbol{\lambda}))$ are the expected value and standard deviation of the prediction $\mathcal{D}_{n+1}(\boldsymbol{\lambda})$.

#### 4.5.3.3 Lower confidence bound

Lower Confidence Bound (or upper confidence bound for maximization problems) assumes the function has the lower bound value [SKKS10]. Therefore, in the face of uncertainty, it is optimistic and assumes the best of cases. The value is calculated as:

$$a_{\mathrm{LCB}}(\boldsymbol{\lambda}, \mathcal{D}_n) = \mu(\boldsymbol{\lambda}) - \kappa\sigma(\boldsymbol{\lambda}) \tag{4.21}$$

where the parameter $\kappa \geq 0$ is set by the user. As with the other functions described, $\kappa$ should be adjusted to balance exploitation and exploration. Algorithm 13 describes the steps of a generic Bayesian optimization.

### 4.5.4 Bayesian optimization algorithm

**Algorithm 13:** BO of a learning algorithm's hyperparameters

```
Input: $\mathcal{D}_{\text{TRAIN}}$ and $\mathcal{D}_{\text{VAL}}$, learning algorithm $\mathcal{A}$, loss function $S$ and the number of
       evaluations $N$.
Output: Returns the best model and hyperparameters based on the observations
        made up to the current iteration $n$ in the loop $\text{argmin}_{(f,\boldsymbol{\lambda})\in\mathcal{D}_n} \hat{R}_f$
$\mathcal{H}_0 \leftarrow \varnothing$
for $n = 1, \dots, N$ do
    $\mu_n(\boldsymbol{\lambda}), \sigma_n^2(\boldsymbol{\lambda}) \leftarrow \mathcal{GP}(\mathcal{D}_{n-1})$ // Fit GP, get mean and variance functions
    $\boldsymbol{\lambda}_n \leftarrow \underset{\boldsymbol{\lambda}}{\text{argmax}}\, a(\boldsymbol{\lambda} \mid \mu_n(\boldsymbol{\lambda}), \sigma_n^2(\boldsymbol{\lambda}))$ // Choose next hyperparameters
    $f_{\boldsymbol{\lambda}_n} \leftarrow \mathcal{A}(\boldsymbol{\lambda}_n, \mathcal{D}_{\text{TRAIN}})$ // Train the model
    $\hat{R}_{\boldsymbol{\lambda}_n} \leftarrow \hat{R}_{\mathcal{D}_{\text{VAL}}}(f_{\boldsymbol{\lambda}} \mid S)$ // Compute validation loss
    $\mathcal{D}_n \leftarrow \mathcal{D}_{n-1} \cup (\boldsymbol{\lambda}_n, S_{\boldsymbol{\lambda}_n})$ // Update observations
end for
return $\text{argmin}_{(f,\boldsymbol{\lambda})\in\mathcal{D}_n} \hat{R}_f$
```

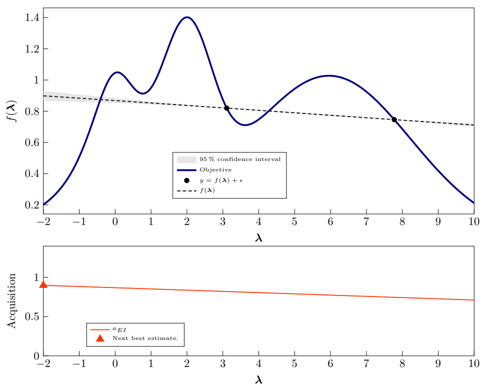
Algorithm 13 shows a step-by-step procedure for Bayesian optimization of hyperparameters. In iteration $n$ we approximate $f_{\boldsymbol{\lambda}_n}$ using our GP surrogate model based on the observation history $\mathcal{D}_n$, i.e., the set of all the hyperparameter settings and performances that have been evaluated so far. The surrogate model is an approximation of $f_{\boldsymbol{\lambda}_n}$ with the property that it can be quickly evaluated. Based on the GP predictions and the corresponding uncertainties about them, the acquisition function finds a trade-off between exploitation and exploration and determines the hyperparameter setting to test next. This setting is then evaluated and the new observation is added to the observation history. After $N$ evaluations, the best performing hyperparameter settings are returned i.e., the model and hyperparameters with the lowest validation loss up to the current iteration $n$ in the loop. We can also train and validate the model with a cross-validation loop (see Algorithm 16). This helps limit overfitting in the hyperparameters selection.

We now provide Algorithm 14, specifically tailored to the context of RS. The algorithm takes three inputs: the hyperparameter space denoted as $\boldsymbol{\Lambda}$, the target score function $S(f_{\boldsymbol{\Theta}}^{\boldsymbol{\lambda}}(u,i), r_{ui})$ that evaluates the performance of a hyperparameter configuration, and the maximum number of evaluations denoted as $N$. The algorithm starts by selecting an initial hyperparameter configuration $\boldsymbol{\lambda}_0$ from the hyperparameter space $\boldsymbol{\Lambda}$ and evaluates its initial score $y_0 = S(f_{\boldsymbol{\Theta}}^{\boldsymbol{\lambda}_0}(u,i), r_{ui})$. Several variables are initialized, including $\boldsymbol{\lambda}^\star$ set to $\boldsymbol{\lambda}_0$, $y^\star$ set to $y_0$, $\mathcal{D}_0$ initialized with $(\boldsymbol{\lambda}_0, y_0)$, and $\epsilon^\star$ initialized with $y_0$. The main optimization loop then iterates from 1 to $N$, selecting a new hyperparameter configuration $\boldsymbol{\lambda}_n$ based on an acquisition function $a_n(\boldsymbol{\lambda}, \mathcal{D}_n)$. Then, the score function $S(f_{\boldsymbol{\Theta}}^{\boldsymbol{\lambda}_n}(u,i), r_{ui})$ is evaluated with $\boldsymbol{\lambda}_n$ to obtain a new score $y_n$, which is added to the dataset $\mathcal{D}_n$. The surrogate model, representing the relationship between hyperparameters and scores, is updated using $\mathcal{D}_n$. If $y_n$ is lower than $\epsilon^\star$, then $\boldsymbol{\lambda}_n$ becomes the new best configuration $\boldsymbol{\lambda}^\star$, and $y_n$ becomes the new best score $y^\star$. After completing all iterations, the algorithm returns the best hyperparameter configuration $\boldsymbol{\lambda}^\star$ and its corresponding best score $y^\star$.

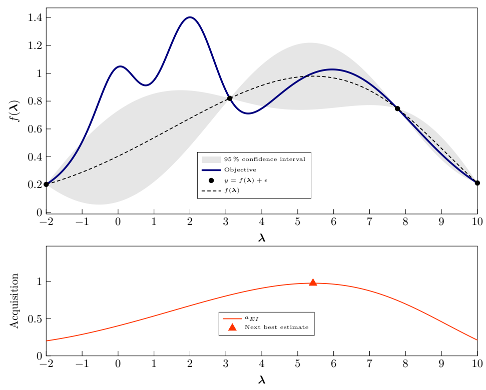
**Algorithm 14:** BO of a recommendation algorithm's hyperparameters

```
Input: Hyperparameter space $\boldsymbol{\Lambda}$, loss function $S(f_{\boldsymbol{\Theta}}^{\boldsymbol{\lambda}}(u,i), r_{ui})$, number of evaluations $N$.
Output: $\boldsymbol{\lambda}^\star$ and $y^\star$.
Select an initial configuration: $\boldsymbol{\lambda}_0 \in \boldsymbol{\Lambda}$
Evaluate the initial score $y_0 = S(f_{\boldsymbol{\Theta}}^{\boldsymbol{\lambda}_0}(u,i), r_{ui})$
$\boldsymbol{\lambda}^\star \leftarrow \boldsymbol{\lambda}_0$
$y^\star \leftarrow S(f_{\boldsymbol{\Theta}}^{\boldsymbol{\lambda}_0}(u,i), r_{ui})$
$\mathcal{D}_0 \leftarrow \{\boldsymbol{\lambda}_0, y_0\}$
$\epsilon^\star \leftarrow y_0$ // Initialize the threshold $\epsilon^\star$ with the initial score
for $n = 1, \dots, N$ do
    Select a new hyperparameter configuration $\boldsymbol{\lambda}_n \in \boldsymbol{\Lambda}$ by optimizing $a_n$:
    $\boldsymbol{\lambda}_n = \operatorname*{argmax}_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} a_n(\boldsymbol{\lambda}, \mathcal{D}_n)$
    Evaluate $S$ in $\boldsymbol{\lambda}_n$ to obtain a new numeric score $y_n = S(f_{\boldsymbol{\Theta}}^{\boldsymbol{\lambda}_n}(u,i), r_{ui})$
    Increase the data $\mathcal{D}_n = \mathcal{D}_{n-1} \cup (\boldsymbol{\lambda}_n, y_n)$
    Update the surrogate model
    if $y_n < \epsilon^\star$ then
        $\boldsymbol{\lambda}^\star \leftarrow \boldsymbol{\lambda}_n$
        $y^\star \leftarrow y_n$
    end if
end for
```

In summary, Bayesian optimization operates by leveraging a set of explored combinations and their corresponding models' performance. Through fitting a Gaussian Process to this set, the algorithm is able to make predictions about the performance of untested combinations. These predictions are then utilized by an acquisition function to determine the most promising combination for further testing. The selected combination is evaluated, incorporated into the set of explored combinations, and the process continues until resource constraints are reached. This iterative approach allows for efficient exploration of the hyperparameter space while maximizing the utilization of available resources. Illustrations in both Figure 4.1 and 4.2 depict Bayesian optimization in seven iterations, utilizing Gaussian Processes as a surrogate model and Expected Improvement as the acquisition function. At the top of the figure, the objective function to be minimized and its posterior model are displayed. Below, the acquisition function for maximizing, obtained from the posterior distribution, is shown. The dashed line represents the true function, while the black diamonds represent the observed samples of the function. The blue line and the envelope indicate the posterior mean and variance functions of the GP. The primary objective is to minimize an objective function $a$, with an input hyperparameter $\boldsymbol{\lambda}$.



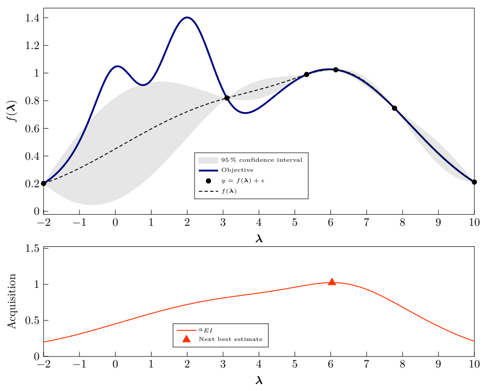
(A) $n = 2$ evaluations.



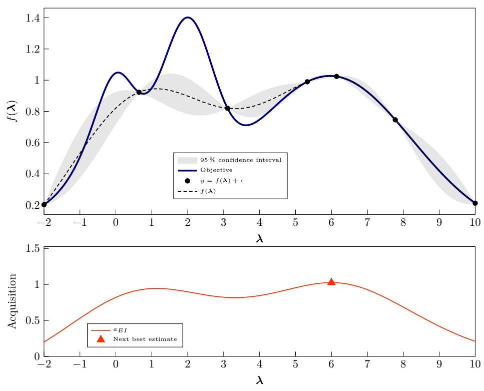
(B) $n = 4$ evaluations.

FIGURE 4.1: Example of a GP with an EI acquisition function.

The algorithm develops a surrogate model that approximates the objective function based on observations, with the graph showing the predicted distribution over $a$ through the mean value and the 95% confidence interval, employing a probabilistic model of the objective function. The acquisition function assigns a utility value for each point for evaluation, balancing exploitation, given by the mean value, and exploration, given by the uncertainty of the prediction. The point with the maximum utility is evaluated. In the subsequent iteration, the surrogate model is reconstructed to adapt to a new observation. We observe that after a few iterations, the algorithm can approach the true maximum closely.



(A) $n = 6$ evaluations.



(B) $n = 7$ evaluations.

FIGURE 4.2: Example of a GP with an EI acquisition function.

# Chapter 5

# Experiments and Discussion

This chapter is structured as follows: Section 5.1 describes the hardware and software employed in our experiments and outlines how the datasets were constructed. Next, Section 5.3 presents an exploratory analysis of the two datasets, followed by Section 5.4 which details the methods used for their division and the techniques employed for training, validation and testing. In Section 5.6.1, we examine the impact of each hyperparameter on the prediction algorithms. In Section 5.6.2, we fine-tune the hyperparameters of the proposed algorithms using the techniques described in Chapter 4. We run multiple experiments, report the results and analyze them. Lastly, in Section 5.6.3, we evaluate the performance of the algorithms under different training proportions and provide commentary on the results.

## 5.1 System configuration

Tests were run on a 4-core Intel Xeon CPU, clocked at 2.8 GHz, equipped with 32GB of RAM and running on the Ubuntu operating system. All algorithms have been implemented in Python 3.6. Our experiments were carried out using Predictive Accuracy, an open source Python recommendation engine that we developed for the occasion. The main characteristics of our library are the following:

- Easy handling of datasets. Not only can the built-in Movielens datasets be used, but custom datasets can be used as well.

- The algorithms mentioned in this work are ready to be used and do not require any prior adjustment.
- The parameters of the proposed algorithms are easily modifiable.

To implement Bayesian optimization (4.5) with PI (4.16), EI (4.19), LBC (4.21) and Gaussian Processes, we used the *Scikit-Optimize* package [PVG$^+$12].

According to [Bjo19], an essential element of the scientific method is *reproducibility.* This means that the findings of a research study should be replicable by anyone who has access to the original data, mathematical derivations or programming code. In other words, for a research work to contribute to the advancement of the field, it must undergo scrutiny and replication by other researchers. To encourage a greater level of research reproducibility, we have included the simulation code alongside this work. The interested reader can access it through the following link:

SIMULATION CODE

## 5.2 Datasets

The MovieLens dataset [HK15], managed by GroupLens Research at the University of Minnesota, comprises movie ratings obtained through the MovieLens project. Ratings were obtained exclusively via the MovieLens website, excluding users with less than 20 ratings. To evaluate the proposal presented in this work, we selected the ML 100K and ML 1M datasets, which respectively include 100000 ratings from 943 users for 1682 movies and 1000000 ratings from 6040 users for 3952 movies (Table 5.1).

| | $\|U\|$ | $\|I\|$ | $\|\mathbf{R}\|$ | **Density** | **Rating** |
|---|---|---|---|---|---|
| **ML 100K** | 943 | 1682 | 100000 | 6.30 % | 1:5 (5) |
| **ML 1M** | 6040 | 3952 | 1000000 | 4.47% | 1:5 (5) |

TABLE 5.1: Dataset statistics.

The dataset contains ratings that follow the format below:

```
(user_id, movie_id, rating, timestamp)
```

In this format, `user_id` and `movie_id` serve as unique identifiers for users and movies, respectively. The `rating` ranges from 1 (worst) to 5 (best), represented as an integer. The `timestamp` is a UNIX timestamp represented as an integer indicating the time of the rating in seconds since 1-1-1970.

### 5.2.1 Description of the variables involved

- Quantitative variables
  - `user_id`: discrete. It ranges from 1 to 943 for ML 100K, and between 1 and 6040 for ML 1M.
  - `movie_id`: discrete. It ranges between 1 and 1682 for ML 100K, and between 1 and 3952 for ML 1M.
  - `timestamp`: discrete. Date and time the rating was given, represented in seconds.
- System prediction
  - `rating`: discrete. Between 1 and 5 stars.

Since the ratings are drawn from the set $\{1, 2, 3, 4, 5\}$, it is tempting to use a categorical rating model. However, this does not reflect the fact that **ratings are ordered**.

## 5.3 Exploratory data analysis

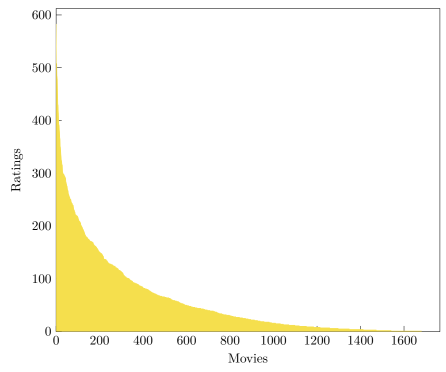
In this section, we present an analysis of the Movielens dataset 100K and 1M, providing a comprehensive overview of the dataset, the distribution of user ratings and the long-tail phenomenon.

### 5.3.1 Long tail

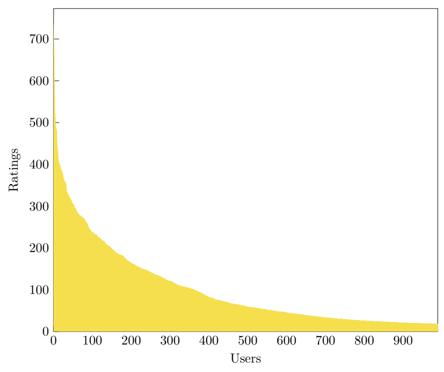
Let

$$f_i = \sum_{u=1}^{n} \mathbb{1}_{u,i \in \mathcal{D}}$$

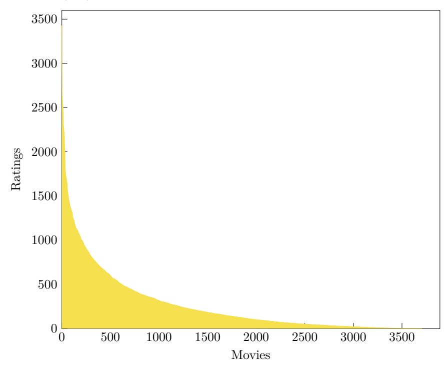
represent the frequency of item $i$ in $\mathcal{D}$. This frequency is equivalent to the number of users who have rated item $i$ or the number of entries in the $i$-th column of matrix $\mathbf{R}$. Popularity of an item is measured by its frequency. In most datasets, items follow a *long tail* distribution, where only a few items are highly popular, while the majority have few ratings. This phenomenon aligns with the Pareto principle, commonly known as the 80/20 rule. To illustrate this, let us say there is a website where people can rate movies on a scale from 1 to 5 stars. Let us imagine that there are 100 different movies on the website, ranging from old classics to new releases. When we look at the data, we can see that typically only a small number of movies, maybe around 20, receive a lot of ratings from users.



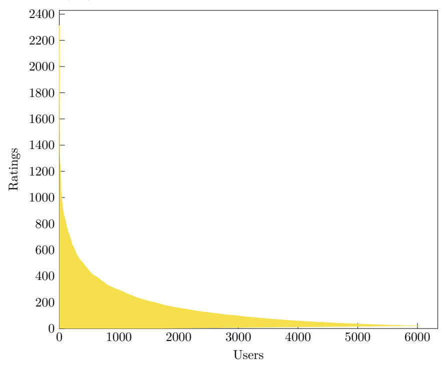
(A) Item ratings: ML 100K.



(B) User ratings: ML 100K



(C) Item ratings: ML 1M.



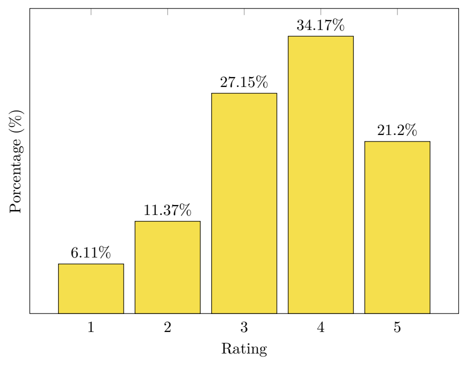
(D) User ratings: ML 1M

FIGURE 5.1: Popularity of items in decreasing order.

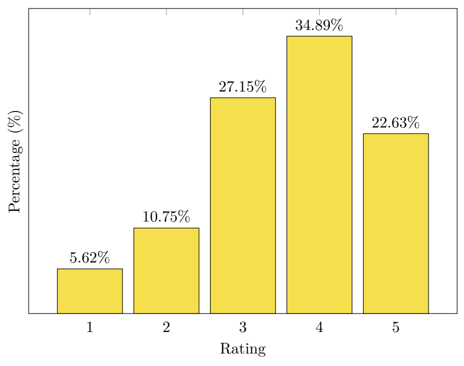
These popular movies account for 80% of all the ratings on the website. On the other hand, the remaining 80 movies receive fewer ratings overall, representing only 20% of the total ratings (see Figure 5.1) In other words, some movies are rated much more frequently than others, and this can make it challenging to recommend movies that are not as popular. This is an important consideration for websites that offer movie recommendations, as they need to ensure they are not just recommending the most popular movies, but also providing options that users may not have considered before. RS face a significant challenge of compensating for this imbalance by generating reliable predictions for long-tail items to avoid recommending only the popular items and increase novelty [And04].

### 5.3.2 Distribution of ratings

The presented histogram, as depicted in Figure 5.2, reveals that the predominant ratings for both datasets fall within the range of 3 to 5 stars. This observation suggests that users are more inclined to assign ratings of 3 and 4 stars, which hold a greater impact on the final predictive model derived from the entire set of training ratings. Notably, negative ratings (1 or 2 stars) account for a mere 17.48% of the overall ratings, whereas the remaining 82.52% tend to be relatively positive, as can be seen in both Figure 5.2a and Figure 5.2b. Additionally, the 4-star rating is the most frequent rating given, with left-skewed distributions. These outcomes hold consistently across both datasets.



(A) ML 100K



(B) ML 1M

FIGURE 5.2: Ratings as a function of percentage.

## 5.4 Dataset splitting

Datasets are created so that each of the subsets is separate from the other two, and the three subsets combined contain all the ratings. Then, $\mathcal{D}$ is divided into three disjoint sets $(\mathcal{D}_{\text{TRAIN}} \cap \mathcal{D}_{\text{VAL}}) \cup (\mathcal{D}_{\text{TRAIN}} \cap \mathcal{D}_{\text{TEST}}) \cup (\mathcal{D}_{\text{VAL}} \cap \mathcal{D}_{\text{TEST}}) = \varnothing$, the training set $\mathcal{D}_{\text{TRAIN}}$, the validation set $\mathcal{D}_{\text{VAL}}$, and the test set $\mathcal{D}_{\text{TEST}}$. Such that $\mathcal{D}_{\text{TRAIN}} \cup \mathcal{D}_{\text{VAL}} \cup \mathcal{D}_{\text{TEST}} = \mathcal{D}$. We also denote $(\mathcal{D}_{\text{TRAIN}} \cap \mathcal{D}_{\text{VAL}}) = \varnothing$ as $\mathcal{D}_{\text{TRAINVAL}}$.

## 5.5 Experiments

The section follows a structured format, beginning with Experiment 1 (Section 5.5.1): Hyperparameter impact analysis in RS, where the focus is on analyzing the influence of hyperparameters on the performance of ML models applied to RS. Experiment 2 (Section 5.5.2): Comparative performance of hyperparameter optimization techniques provides a detailed comparison between random search and Bayesian optimization techniques in identifying optimal hyperparameters for RS models. Experiment 3 (Section 5.5.3): Investigating the effects of data sparsity on recommendation algorithms delves into studying the impact of varying levels of sparsity in the rating matrix on algorithm performance and accuracy.

### 5.5.1 Hyperparameter impact analysis in RS

Here, we aim to analyze the impact of hyperparameters on the performance of ML algorithms applied to RS. The objective of this experiment is to investigate various hyperparameters and their effects on model accuracy and convergence speed. By conducting a comprehensive analysis, we aim to identify the optimal hyperparameters that enhance the overall performance of the RS. The findings from this experiment provide valuable insights into the significance of hyperparameters and their impact on the effectiveness of ML models in the context of RS.

The experiment utilizes a 5-fold cross-validation method, as presented in Algorithm 15, to estimate the validation error. The average validation error across $k$ trials gives an estimation of the validation error. During trial $i$, the $i$-th subset of data is used as the validation set, and the remaining data is utilized as the training set. It is noteworthy that the errors will yield a biased estimate of the true generalization error. Nonetheless, this should not be a concern, as the experiment's objective is to evaluate the impact of hyperparameters. In the subsequent experiment, we will concentrate on obtaining an unbiased estimation of the generalization errors.

### 5.5.2 Comparative performance of hyperparameter optimization techniques

In this experiment, we compare the performance of two distinct hyperparameter optimization techniques: random search and Bayesian optimization. Our objective is to evaluate

**Algorithm 15:** $k$-fold cross-validation

```
Input: Model $f$, a set of ratings $\mathcal{D}$, the number of folds $k$.
Output: Learnt model $f^\star$, average training error $E_{\text{TRAIN}}$, average validation error $E_{\text{VAL}}$.
Randomly shuffle the ratings in $\mathcal{D}$
Divide $\mathcal{D}$ into $k$ folds $\mathcal{D}_1, \dots, \mathcal{D}_k$
for fold index $i = 1, \dots, k$ do
    Use $i$-th fold as the validation set $\mathcal{D}_{\text{VAL}} = \mathcal{D}_i$
    Use the rest as the training set $\mathcal{D}_{\text{TRAIN}} = \mathcal{D} \setminus \mathcal{D}_i$
    Learn function $f^\star$ via ERM on the training set:
        $$f_i^\star = \operatorname*{argmin}_{f \in \mathcal{F}} \hat{R}_{\mathcal{D}_{\text{TRAIN}}}(f) \quad (5.1)$$
    Compute the training error:
        $$E^i_{\text{TRAIN}} = \hat{R}_{\mathcal{D}_{\text{TRAIN}}}(f^\star) = \sqrt{\frac{1}{|\mathcal{D}_{\text{TRAIN}}|} \sum_{r_{ui} \in \mathcal{D}_{\text{TRAIN}}} (r_{ui} - \hat{r}_{ui})^2} \quad (5.2)$$
    Compute the validation error:
        $$E^i_{\text{VAL}} = \hat{R}_{\mathcal{D}_{\text{VAL}}}(f^\star) = \sqrt{\frac{1}{|\mathcal{D}_{\text{VAL}}|} \sum_{r_{ui} \in \mathcal{D}_{\text{VAL}}} (r_{ui} - \hat{r}_{ui})^2} \quad (5.3)$$
end for
Compute average training and validation errors:
    $$E_{\text{TRAIN}} = \frac{1}{k} \sum_{i=1}^{k} E^i_{\text{TRAIN}}, \quad E_{\text{VAL}} = \frac{1}{k} \sum_{i=1}^{k} E^i_{\text{VAL}} \quad (5.4)$$
Pick a learnt model $f^\star = f_i^\star$ for some $i \in \{1, ..., k\}$
```

the efficiency and effectiveness of each approach in identifying optimal hyperparameters for RS algorithms. Through a thorough analysis, we gain valuable insights into the strengths and weaknesses of both techniques. By examining the factors influencing their effectiveness, this experiment provides a comprehensive comparison of random search and Bayesian optimization.

The experiment utilizes Algorithm 16, which offers a straightforward and efficient method for selecting the best candidate model among a set of models, $f_1, f_2, \dots, f_M$. The process involves learning and validating a model $f_i^\star$ for each model $f_i$ using Algorithm 15. Specifically, for each model $f_i$, we employ the ERM (Equation (2.14)) approach to learn the model $f_i^\star$ and determine its validation error, $E^i_{\text{VAL}}$, as outlined in Equations (5.2) and (5.3), respectively. We then select the model $f_{\hat{i}}^\star$ with the smallest validation error, $E^{\hat{i}}_{\text{VAL}} = \min_{i=1,\dots,M} E^i_{\text{VAL}}$, from the model set $\hat{\mathcal{F}}_{\hat{i}}$ that achieved the lowest validation error.

**Algorithm 16:** $k$-Fold cross validation for model selection

```
Input: List of candidates models $f_1, \dots, f_M$, a set of ratings $\mathcal{D}$, number of folds $k$.
Output: Model $f^\star$, training error $E^{\hat{i}}_{\text{TRAIN}}$, validation error $E^{\hat{i}}_{\text{VAL}}$, test error $E_{\text{TEST}}$.
Randomly shuffle the ratings in $\mathcal{D}$
Construct a training set and a validation set $\mathcal{D}_{\text{TRAINVAL}}$
Construct a test set $\mathcal{D}_{\text{TEST}}$
for model index $i = 1, \dots, M$ do
    Run Algorithm 15 using $f = f_i$, dataset $\mathcal{D} = \mathcal{D}_{\text{TRAINVAL}}$ and $k$ folds
    Algorithm 15 delivers model $f^\star$ and validation error $E_{\text{VAL}}$
    Store learnt model $f^\star_i = f^\star$ and validation error $E^i_{\text{VAL}} = E_{\text{VAL}}$
end for
Pick model $f_{\hat{i}}$ with minimum validation error $E^{\hat{i}}_{\text{VAL}} = \min_{i=1,\dots,M} E^i_{\text{VAL}}$
Define optimal model $f^\star = f^\star_{\hat{i}}$
Compute the test error:
```

$$E_{\text{TEST}} = \hat{R}_{\mathcal{D}_{\text{TEST}}}(f^\star) = \sqrt{\frac{1}{|\mathcal{D}_{\text{TEST}}|} \sum_{r_{ui} \in \mathcal{D}_{\text{TEST}}} (r_{ui} - \hat{r}_{ui})^2} \tag{5.5}$$

Algorithm 16 follows a similar workflow to ERM, which involves selecting the best model from a pool of candidate models. However, in Algorithm 16, we choose the optimal model space from a collection of candidate model spaces. We measure the quality of each model space using its validation error, which we compute by training the model $f^\star \in \mathcal{F}$ via ERM on the training set, and then measuring its average loss on the validation set.

For each configuration, we train the model on the sub-training set and evaluate it on the validation set using 5-fold cross-validation, in this phase we experiment with various hyperparameter configurations using Bayesian optimization (see Section 4.5). To accurately estimate the performance of the selected model $f^\star$, it is essential to test it on an independent set of ratings that were not employed for training (Equation (5.2)) or validation (Equation (5.3)). We construct this test set and calculate the test error, which is the average loss of the final model on the test set, as shown in Equation (5.5). For a detailed illustration of the proposed procedure, see Appendix D.1.

### 5.5.3 Investigating the effects of data sparsity on recommendation algorithms

Here, we focus on investigating the effects of data sparsity on recommendation algorithms. Our objective is to explore how varying levels of sparsity in the rating matrix impact algorithm performance and accuracy. By studying the behavior of RS under different sparsity conditions, we provide valuable insights into the robustness and adaptability of these algorithms in real-world scenarios. The findings from this experiment contribute to a better understanding of the challenges posed by data sparsity and offer guidance for developing more effective recommendation algorithms.

**Algorithm 17:** Hold-out validation

```
Input: Model $f$, a set of ratings $\mathcal{D}$.
Output: Learnt model $f^\star$, training error $E_{\text{TRAIN}}$, validation error $E_{\text{VAL}}$.
Randomly shuffle the ratings in $\mathcal{D}$
Create the training set $\mathcal{D}_{\text{TRAIN}}$
Create the validation set $\mathcal{D}_{\text{VAL}}$
Learn model $f^\star$ via ERM on the training set:
    $f^\star = \operatorname*{argmin}_{f \in \mathcal{F}} \hat{R}_{\mathcal{D}_{\text{TRAIN}}}(f)$    (5.6)
Compute the training error:
    $E_{\text{TRAIN}} = \hat{R}_{\mathcal{D}_{\text{TRAIN}}}(f^\star) = \sqrt{\frac{1}{|\mathcal{D}_{\text{TRAIN}}|} \sum_{r_{ui} \in \mathcal{D}_{\text{TRAIN}}} (r_{ui} - \hat{r}_{ui})^2}$    (5.7)
Compute the validation error:
    $E_{\text{VAL}} = \hat{R}_{\mathcal{D}_{\text{VAL}}}(f^\star) = \sqrt{\frac{1}{|\mathcal{D}_{\text{VAL}}|} \sum_{r_{ui} \in \mathcal{D}_{\text{VAL}}} (r_{ui} - \hat{r}_{ui})^2}$    (5.8)
```

This experiment employs the classical hold-out approach, as presented in Algorithm 17. The dataset is initially split into two subsets: the training set (80% of the data) and the validation set (20% of the data). We choose a suitable learning algorithm and train it to create a model. Next, we use the validation set to predict ratings, providing an estimation of the model's performance on unseen data. Furthermore, we provide a comprehensive explanation of how the training set is utilized.

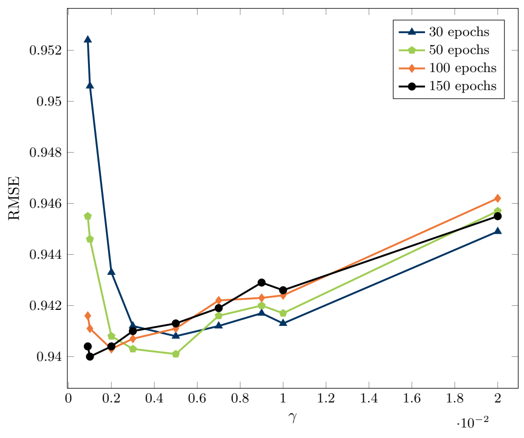
## 5.6 Results

This section presents the findings of Experiments 1 (5.5.1), 2 (5.5.2), and 3 (5.5.3), which are detailed in Sections 5.6.1, 5.6.2, and 5.6.3, respectively.

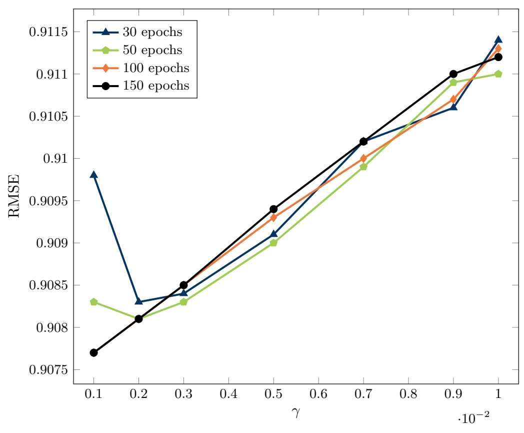
### 5.6.1 Experiment 1

#### 5.6.1.1 Baseline Predictor

To obtain biases $b_u$ and $b_i$, we used Algorithm 4. The hyperparameters we needed to adjust were: learning rate $\gamma$, regularization rate $\lambda$ and the number of epochs $n$. The learning rate $\gamma$ needed constant readjustment to avoid a value that was too high or too low. If it was too high, it could delay convergence, and if it was too low, the iteration would likely gradually enter a divergence that surrounded the minimum instead of the expected direction.



(A) Validation error ML 100K.



(B) Validation error ML 1M.

FIGURE 5.3: RMSE under different values of $\Gamma$ and $n$.

The value of $\gamma$ was closely related to the number of epochs. For a very low value of $\gamma$, a higher number of epochs $n$ was required, as the algorithm learned at a much slower pace. However, the number of epochs should not be too high as this would cause the model to overfit the training data, which could worsen performance on the validation set. We kept the regularization rate $\lambda$ fixed at 0.01 and varied $\gamma$ and $n$. Since the learning rate and the required number of epochs can influence each other, they must be varied simultaneously to optimize both values. In Figure 5.10a, we see that high values of the learning rate lead to overfitting, especially as the number of epochs increases. The error also increases rapidly when $\gamma$ is set to low values and the number of epochs decreases. The minimum seems to occur around $\gamma = 0.005$ and $n = 50$ for ML 100K and $\gamma = 0.001$ and $n = 100$ for ML 1M. We considered 100 training epochs instead of 150 for ML 1M because 50 extra epochs consume more time and computational resources to obtain an insignificant improvement.

#### 5.6.1.2 $k$-NN$^{\bigstar}_{\text{USERS}}$

To enhance the performance of the $k$-nearest neighbors method, we incorporated the Baseline Predictor, $b_{ui}$. Without any interaction between two users, the prediction function would output a rating prediction of $r_{ui} = 0$. However, including the Baseline Predictor in our prediction function allowed us to obtain a rating prediction *even when there is no interaction between users*. This improved the algorithm's performance significantly.

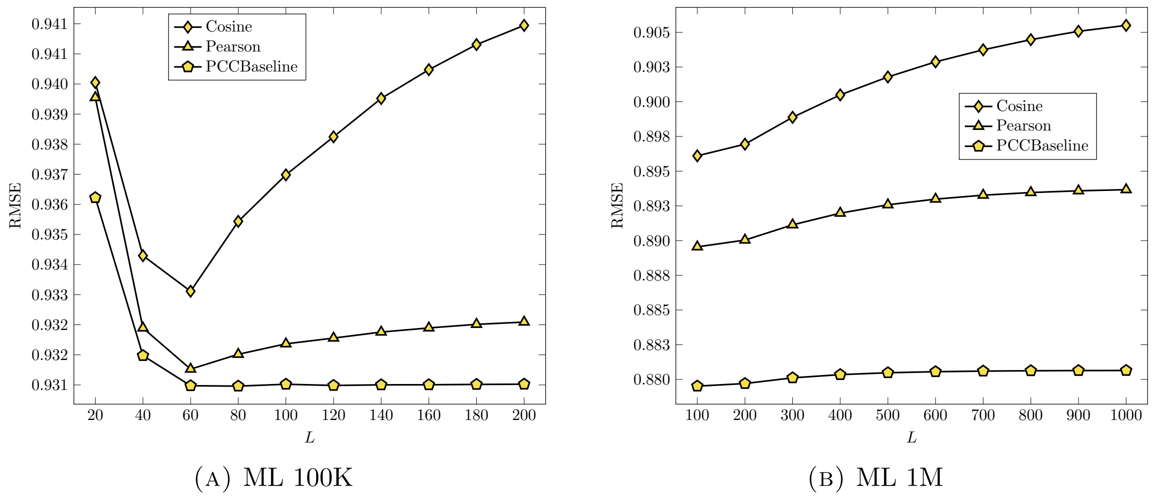


FIGURE 5.4: Comparison of **rmse** with different similarity metrics and sizes of $L$.

To minimize Equation (3.3) and obtain the biases of users and movies, we fixed the hyperparameter values at $\lambda = 0.02$, $n = 50$ and $\gamma = 0.005$ for both datasets. Next, we computed the similarity between users using the metrics described in Section 3.3.1.2, and selected the $L$ nearest neighbors for each user. We varied the value of $L$ from 20 to 200 for ML 100K, and from 100 to 1000 for ML 1M during the experiment. Finally, we predicted the target user's rating for an unrated item by combining the ratings of the selected neighbors using Equation (3.20). Figures 5.4a and 5.4b illustrate that using the Pearson similarity based on the Baseline Predictor (Equation (3.19)) results in the lowest **rmse** and thus achieves the highest level of prediction accuracy. It turns out that subtracting $b_{ui}$ from the user provides better predictions compared to other metrics. Figure 5.4a indicates that for cosine similarity, the error is minimized to 0.9370 when $L = 60$ but increases as $L$ exceeds 60. In contrast, for Pearson similarity, the error initially increases but reaches its minimum of 0.9327 at $L = 60$ and hits a plateau when $L > 160$. Additionally, centered Pearson similarity on the Baseline Predictor demonstrates an initial decrease, then stabilizes and attains a minimum error of 0.9306 when $L = 60$ and $\xi = 1$. Upon analyzing the curves, it is apparent that cosine similarity yields the lowest accuracy because it only considers the angle between vectors, disregarding the correlation between the ratings and the average ratings. For instance, considering three ratings $r_1$, $r_2$, and $r_3$, represented by the vectors $(5, 5, 5)$, $(1, 1, 1)$, and $(4, 5, 5)$, respectively, it is evident that $r_1$ and $r_3$ are more similar, although according to the similarity metric, $r_1$ and $r_2$ are more similar.

As demonstrated in Figure 5.4b, cosine similarity produces a minimum error of 0.8961 at $L = 100$. For $100 < L < 900$, the error increases as $L$ increases, eventually stabilizing for $L \geq 1000$. This phenomenon occurs because many users do not possess a high degree of similarity within their neighborhood set $\mathcal{L}_u$ when $L$ is large, resulting in a higher error in the predicted rating. In contrast, Pearson similarity initially decreases the error before increasing it, attaining a minimum of 0.8895 for $L = 100$. As $L$ rises between $100 < L < 700$, the error increases before stabilizing for $L > 700$. When Pearson similarity is centered on the Baseline Predictor, it first decreases to a minimum error of 0.8795 at $L = 100$ and $\xi = 1$, then increases between $100 < L < 400$ and hits a plateau for $L > 500$.

The proposed algorithm was evaluated on the ML 100K dataset with varying degrees of sparsity to determine its effectiveness in handling sparse data. Pearson similarity centered on the Baseline Predictor was used as the metric, with a fixed regularization rate of $\lambda = 0.02$, learning rate of $\gamma = 0.005$, number of epochs of $n = 20$ and $\xi = 1$. The number $L$ of neighbors was varied from 100 to 800 in increments of 100, as recommended by [KB11] for the Netflix dataset, which has been shown to be effective for our experiments.

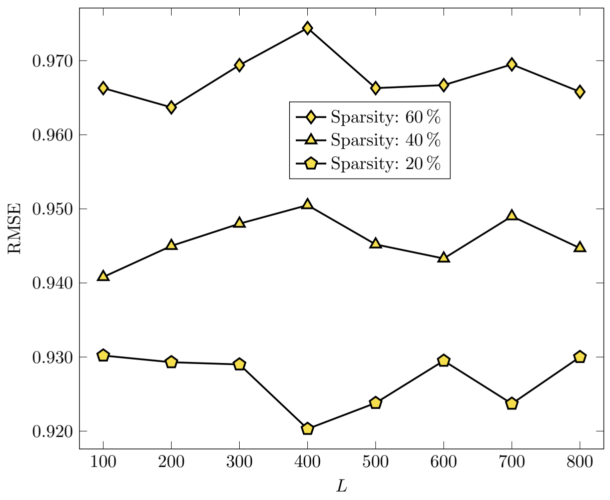
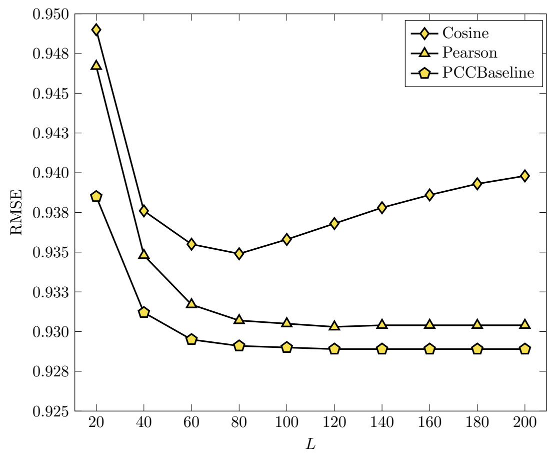


FIGURE 5.5: Evolution of precision (**rmse**) for ML 100K according to matrix density.

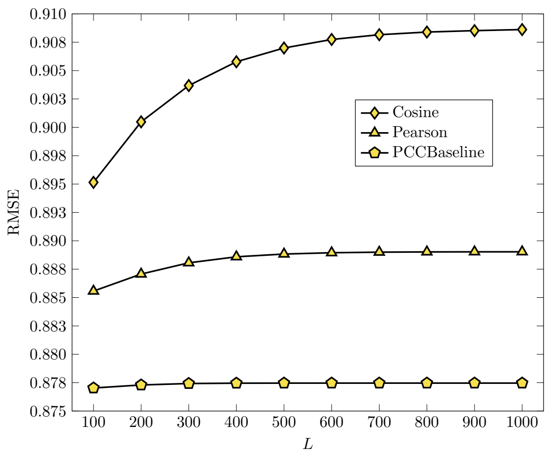
The relationship between the sparsity degree of ratings and the error value is depicted in Figure 5.5. As the degree of sparsity increases, the error value also increases gradually. The figure showcases three sparsity degree levels: 60%, 40%, and 20%, which correspond to data densities of 40%, 60%, and 80%, respectively. A higher number of ratings in the rating matrix results in a lower sparsity degree. This reduction in sparsity provides more rating information, leading to more accurate similarity calculations and reducing the discrepancy between predicted and actual ratings. On the other hand, as the sparsity degree of ratings increases, the time required to reach the minimum error value also increases gradually. This is because a higher degree of sparsity necessitates more abundant rating data for the calculation of user similarity, which in turn leads to an expanded neighborhood set $\mathcal{L}_u$.

#### 5.6.1.3 $k$-NN$^{\star}_{\text{ITEMS}}$

The proposed algorithm incorporates $b_{ij}$ into the $k$-nearest neighbors method to enhance its performance. In order to minimize Equation (3.3), we utilize the identical hyperparameter values as those employed in the $k$-NN$^{\star}_{\text{USERS}}$ algorithm discussed in Section 5.6.1.2. Next, we employ three similarity metrics mentioned in Section 3.3.2.1 to compute the similarity between movies, selecting $L$ nearest neighbors for each item. We explore the range of $L$ values from 20 to 200 for ML 100K and 100 to 1000 for ML 1M. Ultimately, we estimate the target user rating for an unrated item by using Equation (3.26) to combine the ratings of the selected neighbors. As illustrated in Figure 5.6a, the error is minimized at 0.9351 when $L = 80$ using cosine similarity. The error subsequently increases with the increase in $L$ for $60 < L < 100$. With Pearson similarity, the error initially increases but then decreases, reaching a minimum of 0.9303 at $L = 120$. When $L > 120$, the error hits a plateau. For centered Pearson similarity based on the Baseline Predictor, the error initially decreases and then hits a plateau, reaching a minimum of 0.9289 at $L = 120$ and $\xi = 1$.



(A) ML 100K



(B) ML 1M

FIGURE 5.6: Comparison of **rmse** with different similarity metrics and sizes of $L$.

The curve in Figure 5.6b depicts the error trend for three similarity metrics: cosine, Pearson, and centered Pearson similarity based on the Baseline Predictor. For cosine similarity, the error reaches its minimum value of 0.8951 when $L = 100$, and then increases for $100 < L < 700$ due to a larger number of movies with low similarity in the neighbor set $\mathcal{L}_i$ at larger values of $L$, leading to higher errors in predicted ratings. Finally, the error tends to stabilize for $L > 700$. In contrast, the error for Pearson similarity first decreases and then increases, with a minimum of 0.8855 for $L = 100$, followed by an increase for $100 < L < 500$, and hits a plateau for $L > 500$. Centered Pearson similarity based on the Baseline Predictor follows a similar pattern, with a minimum error of 0.8770 for $L = 100$ and $\xi = 1$, followed by stabilization. We also evaluated the performance of these metrics on the ML 100K dataset, with varying levels of sparsity, using the same evaluation protocol as described in Section 5.6.1.2. In this evaluation, we focused on items.

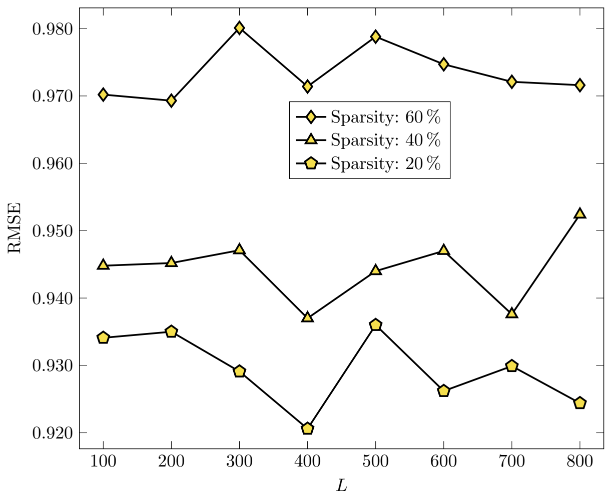
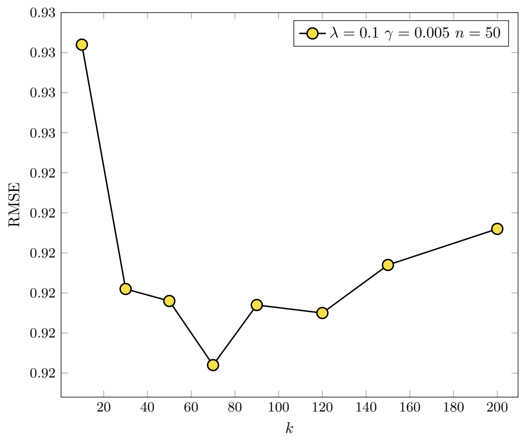


FIGURE 5.7: Evolution of precision (**rmse**) for ML 100K according to matrix density.

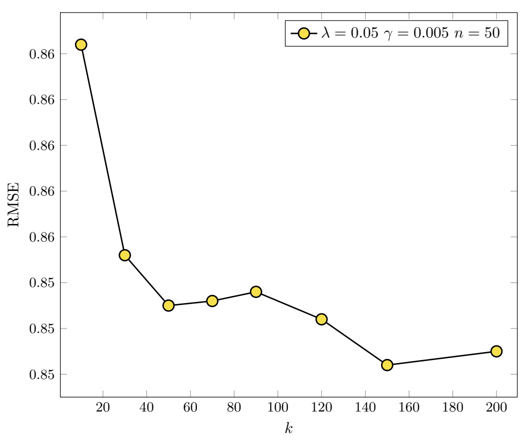
The findings depicted in Figure 5.7 reveal a noticeable upward trend in error as the degree of sparsity of ratings in the matrix increases. Notably, the algorithm exhibits similar behavior to the user-based approach; however, the error rate obtained for all three degrees of sparsity is comparatively lower in this method. It is worth noting that as the degree of sparsity increases, less rating information is available to compute movie similarity, resulting in a smaller neighborhood $\mathcal{L}_i$.

#### 5.6.1.4 SVD$_{\text{UNBIASED}}$

In this algorithm, we investigate the impact of the number of latent factors in $\mathbf{P}$ and $\mathbf{Q}$ on calculation precision. SVD$_{\text{UNBIASED}}$ employs four parameters, specifically the learning rate $\gamma$, the number of latent factors $k$, the regularization rate $\lambda$ and the number of epochs $n$, as presented in Algorithm 11. We set $\gamma = 0.005$, $n = 50$ for both datasets, and $\lambda = 0.01$ for ML 100K and $\lambda = 0.05$ for ML 1M. We varied the number of latent factors $k$ within the range of 10 to 200 with intervals of 20, i.e., $k \in \{10, 30, 50, 70, 90, 110, 130, 150, 170, 190\}$ and studied their effect on the calculation precision.



(A) Validation error ML 100K.

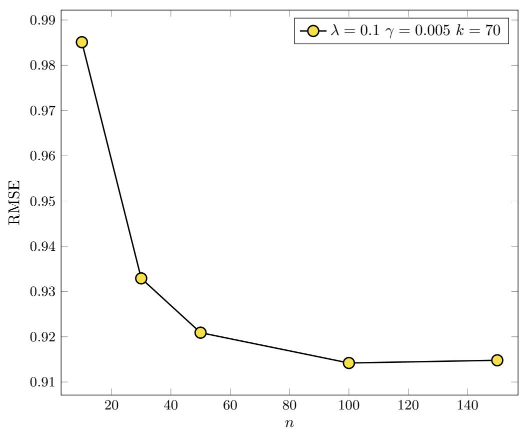


(B) Validation error ML 1M.

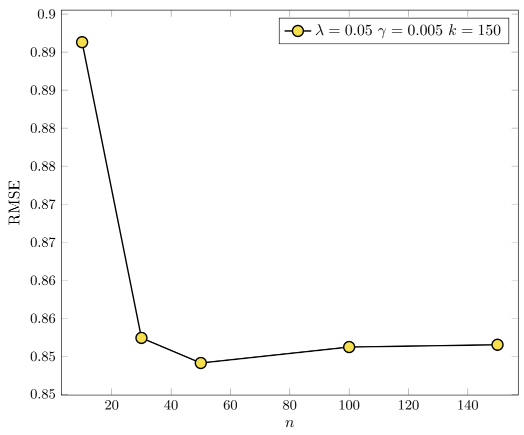
FIGURE 5.8: Trend of **rmse** under different values of $K$.

The experimental results for ML 100K and ML 1M are presented in Figure 5.8a and Figure 5.8b, respectively. With $k = 70$ latent factors, ML 100K achieved an **rmse** of 0.9192, while ML 1M achieved an **rmse** of 0.8532 with $k = 150$ latent factors. The recommendation precision slightly decreases as $k$ increases in both datasets. The increase in **rmse** could be due to the addition of the regularization term $\lambda$, which helps prevent overfitting. However, to improve the model's precision, $\lambda$ must be appropriately adjusted as the number of latent factors increases. We will later discuss how to obtain the optimal value of $\lambda$. Additionally, we analyzed the correlation between algorithm precision and the number of epochs. Let us recall that the number of epochs represents the total number of iterations required to reach the local minimum of the loss function $S$ (Equation(3.36)) using the SGD technique. We tested different values of $n \in \{10, 30, 50, 100, 150\}$ while fixing $\gamma = 0.005$, $k = 70$ and $\lambda = 0.01$ for ML 100K and $\gamma = 0.005$, $k = 150$ and $\lambda = 0.05$ for ML 1M.

After examining Figures 5.9a and 5.9b, it becomes apparent that increasing the value of $n$ results in a decrease in the **rmse** value, followed by some fluctuation, ultimately reaching gradual stability. For the ML 100K dataset, we obtained an **rmse** of 0.9142 with $n = 100$ epochs, while for the ML 1M dataset, the **rmse** was 0.8541 with $n = 50$ epochs, as shown



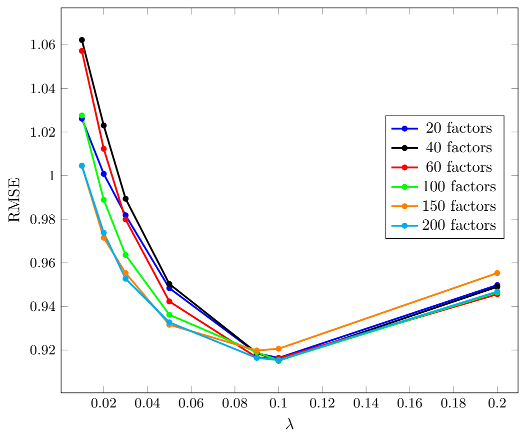
(A) Validation error ML 100K.



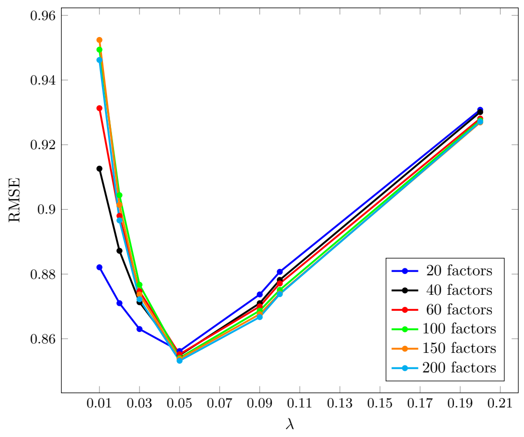
(B) Validation error ML 1M.

FIGURE 5.9: RMSE for different numbers of epochs.

in Figure 5.8b. It is important to note that increasing the number of epochs beyond the local minimum point can lead to a decline in model performance.



(A) Validation error ML 100K.



(B) Validation error ML 1M.

FIGURE 5.10: RMSE under different values of $\Lambda$ and $K$.

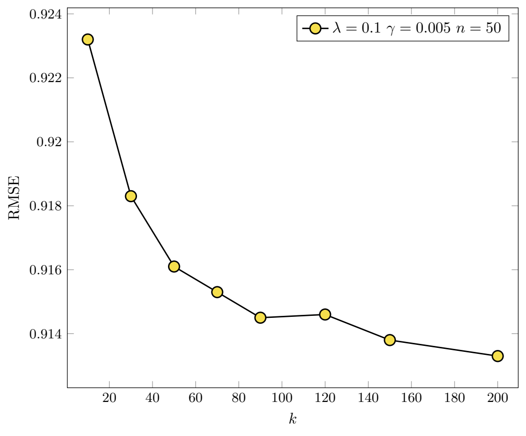
To address the issue of data sparsity and overfitting caused by data noise, we evaluated the optimal value of the regularization rate $\lambda$. We set the learning rate $\gamma$ to 0.005 for both datasets and we varied the number of latent factors $k$, in matrices $\mathbf{P}$ and $\mathbf{Q}$, to $k \in \{20, 40, 60, 100, 150, 200\}$. We tested the regularization parameter $\lambda$ for each value of $k$ using $\lambda \in \{0.01, 0.02, 0.03, 0.05, 0.09, 0.1, 0.2\}$.

The results demonstrate a non-linear relationship between $\lambda$ and the accuracy of the RS, as evidenced in Figure 5.10a. Initially, increasing $\lambda$ improves accuracy, but further increments lead to a decline. Consequently, selecting an appropriate $\lambda$ value becomes crucial for attaining optimal accuracy. The experiment, utilizing a fixed learning rate at

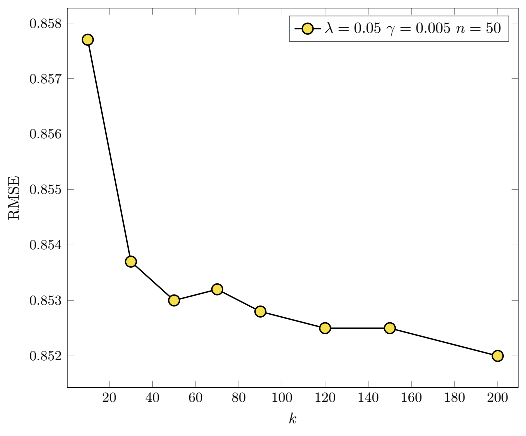
$\gamma = 0.005$ and 100 epochs, highlights the importance of avoiding excessively small or large $\lambda$ values. For example, in the ML 100K dataset, the minimum RMSE of 0.9149 occurred at $\lambda = 0.1$ and $k = 100$. Conversely, in the ML 1M dataset, the minimum RMSE of 0.8532 was observed at $\lambda = 0.05$, $k = 200$, $n = 50$, and $\gamma = 0.005$, as depicted in Figure 5.10b.

#### 5.6.1.5 SVD$_{\text{BIASED}}$

We extend the approach presented in Section 5.6.1.4 by including the global average, user bias and movie bias in the model, represented by $\bar{r}$, $b_u$, and $b_i$, respectively.



(A) Validation error ML 100K.



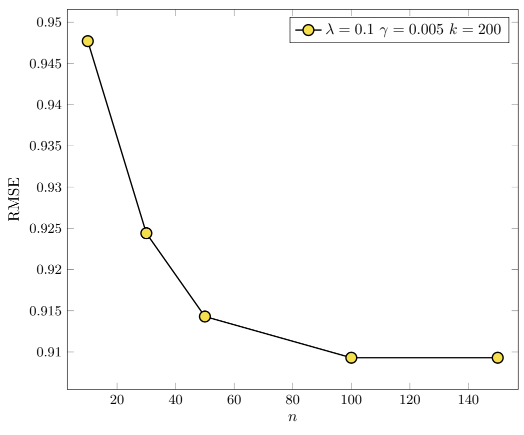
(B) Validation error ML 1M.

FIGURE 5.11: Trend of RMSE under different values of $K$.

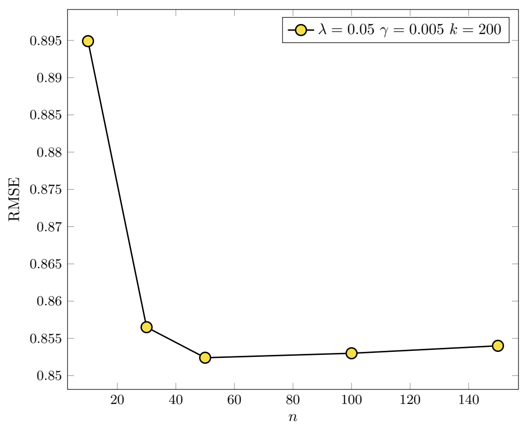
Initially, we assess the impact of latent factors by varying their number $k$ over a range of values: $k \in \{10, 30, 50, 70, 90, 120, 150, 200\}$. The ML 100K dataset employs $\lambda = 0.01$, while the ML 1M dataset uses $\lambda = 0.05$. Both datasets share a fixed learning rate, $\gamma = 0.005$, and consist of 50 epochs. In the ML 100K dataset, the optimal **rmse** value of 0.9134 emerges when $k = 200$, as illustrated in Figure 5.11a.

Similarly, the ML 1M dataset achieves its best **rmse** value of 0.8520 with $k = 200$, as shown in Figure 5.11b. Subsequently, we explore the algorithm's behavior with different epoch numbers. For both datasets, we maintain $\gamma = 0.005$ and $k = 200$, while setting $\lambda = 0.1$ for ML 100K and $\lambda = 0.05$ for ML 1M. We conduct experiments with $n \in \{10, 30, 50, 100, 150\}$ epochs.

We achieved an **rmse** of 0.9093 for ML 100K with $n = 100$ epochs. Figure 5.12a shows that the error remains stable beyond epoch 100. Similarly, for ML 1M, we obtained an



(A) Validation error ML 100K.



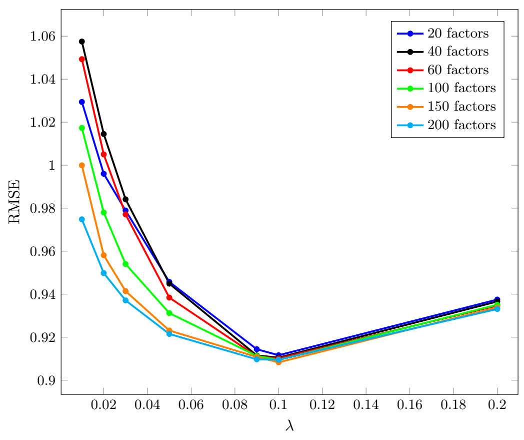
(B) Validation error ML 1M.

FIGURE 5.12: RMSE for different number of epochs.

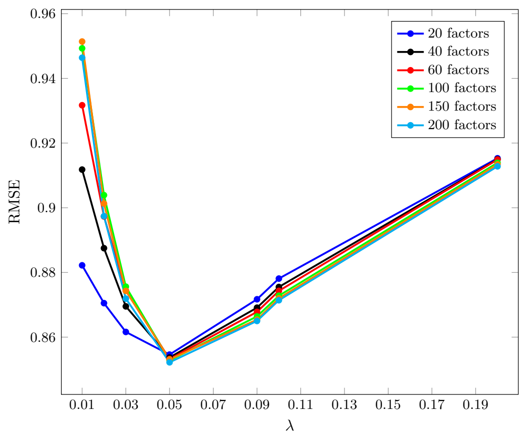
**rmse** of 0.8525 with $n = 50$ epochs. Figure 5.12b illustrates that the model starts to overfit after 50 epochs.

Furthermore, we explored the impact of the regularization rate $\lambda$ on $b_u$, $b_i$, $\mathbf{p}_u$ and $\mathbf{q}_i$ while keeping the learning rate $\gamma$ fixed at 0.005 for all parameters. We conducted experiments with $n = 50$ epochs and considered different numbers of latent factors $k$ in the $\mathbf{P}$ and $\mathbf{Q}$ matrices, specifically $k \in \{20, 30, 50, 100, 150, 200\}$.

We evaluated $\lambda$ for each $k$ value across the range $\lambda \in \{0.01, 0.02, 0.03, 0.05, 0.09, 0.1, 0.2\}$. Figure 5.13 demonstrates a similar behavior to that observed in Algorithm 11 when employing the hyperparameter values selected in Section 5.6.1.4.



(A) Validation error ML 100K.



(B) Validation error ML 1M.

FIGURE 5.13: RMSE under different values of $\Lambda$ and $K$.

### 5.6.2 Experiment 2

#### 5.6.2.1 Performance comparison

In this experiment's initial phase, we compared the effectiveness of random search and Bayesian optimization (Algorithm 14) techniques. Our hyperparameter space comprised 300 models, trained on the ML 100K dataset, with the optimization algorithms set to default values, except for the number of evaluations and the acquisition function. We set the number of evaluations to 50.

We implemented Bayesian optimization by utilizing the *gp_minimize* function from the *Scikit-Optimize* package. This function employs a Gaussian Process as a surrogate model, specifically utilizing a Matern kernel with a parameter of $\nu = 2.5$. Within the *gp_minimize* function, we selected the *gp_hedge* parameter to enable probabilistic selection of one of the three acquisition functions at each iteration. The inclusion of *gp_hedge* introduces a randomized approach, assigning probabilities to the three acquisition functions: PI (4.16), EI (4.19), and LBC (4.21). These probabilities determine the likelihood of selecting a specific acquisition function during each iteration. Initially, equal probabilities are assigned to all acquisition functions. As the optimization progresses, the performance of the acquisition functions is evaluated based on their past performance and their ability to guide the search towards promising regions within the search space. The probabilities assigned to the acquisition functions are dynamically updated using *"multi-armed bandit"* algorithms, which strive to strike a balance between exploration (trying different acquisition functions) and exploitation (focusing on the function that has exhibited superior performance). By adaptively adjusting the probabilities, the algorithm allocates more evaluations to acquisition functions that have demonstrated better performance, while still allowing for a degree of exploration. This iterative process enhances the efficiency of the optimization by directing the search towards more promising regions. The integration of the *gp_hedge* parameter within the *gp_minimize* function facilitates Bayesian optimization by probabilistically selecting acquisition functions, dynamically updating probabilities based on performance, and effectively balancing exploration and exploitation to guide the search towards optimal solutions.

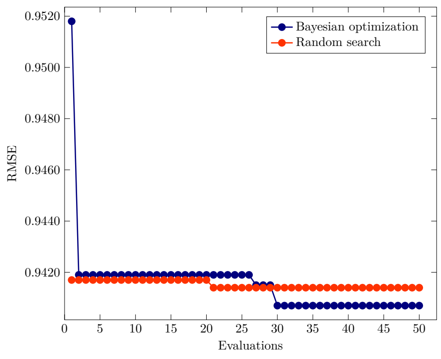
The training process involved three hyperparameters for the Baseline Predictor, resulting in a total of 5733 potential configurations[1]. The memory-based methods used four hyperparameters resulting in a total of 178281 potential configurations, while the model-based methods employed four hyperparameters with a total of 350811 potential configurations.

[1]Using the formula $b - a + 1$ with $[a, b] = \{x \in \mathbb{R} | \ a \leq x \leq b\}$, we obtain $71 \times 9 \times 9 = 5733$

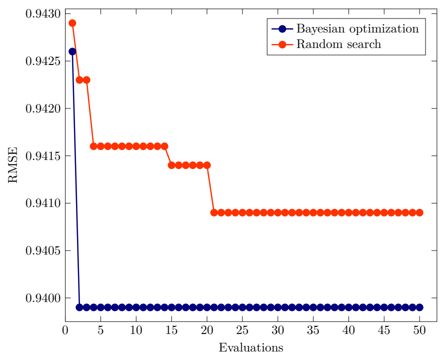
| **Hyperparameter** | **Range of values** |
|---|---|
| Number of epochs $n$ | $[30;\ 100]$ |
| Number of neighbors $L$ | $[20;\ 50]$ |
| Number of factors $k$ | $[40;\ 100]$ |
| Learning rate $\gamma$ | $[10^{-3};\ 9 \cdot 10^{-3}]$ |
| Regularization penalty $\lambda$ | $[10^{-2};\ 9 \cdot 10^{-2}]$ |

Table 5.2: Hyperparameter space explored by the two methods.

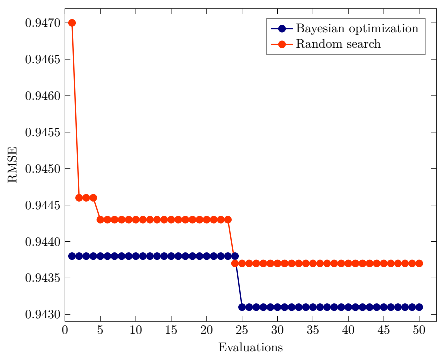
The range of each hyperparameter is provided in Table 5.2. For memory-based methods, the centered Pearson similarity metric was employed on the Baseline Predictor, with $\xi = 0$, as illustrated in Equations (3.19) and (3.25). Four metrics are reported in this experiment: RMSE, Precision, F-Score and MAP. Precision, F-Score and MAP are expressed as percentages. As outlined in Section 3.5.2, higher values for classification metrics reflect better performance, whereas lower values for RMSE are preferred. RMSE is the only metric that exclusively relies on the prediction algorithm $\mathcal{A}$. Conversely, all other metrics are dependent on the recommended items and are determined by a recommendation threshold, namely, an item is recommended to a user if $r_{ui} \geq 4$.



(A) Baseline Predictor.



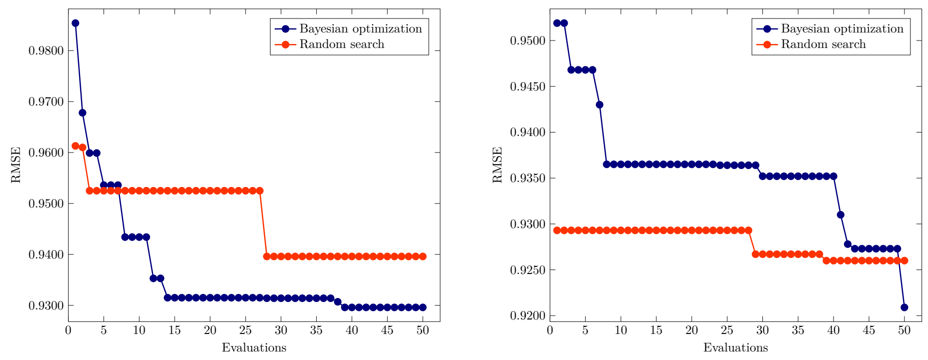
(B) $k$-NN$^{\bigstar}_{\text{USERS}}$



(C) $k$-NN$^{\bigstar}_{\text{ITEMS}}$

Figure 5.14: Test error of memory-based and model-based methods with respect to the number of trained models in ML 100K.

Figures 5.14a, 5.14b, 5.14c, 5.15a, and 5.15b depict the error (RMSE) as a function of the number of evaluations. The random search method required an average of 25 evaluations to find the best model, whereas Bayesian optimization consistently outperformed it in every objective. Specifically, for the 5.14a model, it took 30 evaluations for Bayesian optimization to identify the best model, while for 5.14b and 5.14c, it only required 2 and 25 evaluations, respectively. For the 5.15a and 5.15b models, it took 39 and 50 evaluations, respectively.



(A) $\text{SVD}_{\text{UNBIASED}}$ (B) $\text{SVD}_{\text{BIASED}}$

FIGURE 5.15: Test error of memory-based and model-based methods with respect to the number of trained models in ML 100K.

Remarkably, even the worst execution of Bayesian optimization surpassed the average performance of random search. The curves reveal that Bayesian optimization yields slightly superior results and converges much faster than random search. Nevertheless, given the stochastic nature of these methods, more evaluations are needed to derive definitive conclusions from this evaluation metric. Notably, random search proves to be remarkably effective, although Bayesian optimization outperformed it in every algorithm and metric (RMSE), thereby justifying its substantially higher implementation cost.

| **Dataset** | **Algorithm** | **Best hyperparameters** |
|---|---|---|
| ML 100K | Baseline Predictor | $n = 50$, $\gamma = 0.003862$, $\lambda = 0.010161$ |
| | $k$-$\text{NN}^{\bigstar}_{\text{USERS}}$ | $n = 39$,$\gamma = 0.001229$, $\lambda = 0.034972$, $L = 39$ |
| | $k$-$\text{NN}^{\bigstar}_{\text{ITEMS}}$ | $n = 50$, $\gamma = 0.009$, $\lambda = 0.01$, $L = 100$ |
| | $\text{SVD}_{\text{UNBIASED}}$ | $n = 50$, $k = 43$, $\gamma_{pu} = 0.004815$, $\gamma_{qi} = 0.006103$, $\lambda_{pu} = 0.09$, $\lambda_{qi} = 0.09$ |
| | $\text{SVD}_{\text{BIASED}}$ | $n = 50$, $k = 10$, $\gamma_{pu} = 0.009$, $\gamma_{qi} = 0.009$, $\gamma_{bu} = 0.009$, $\gamma_{bi} = 0.009$, $\lambda_{pu} = 0.09$, $\lambda_{qi} = 0.09$, $\lambda_{bu} = 0.01$, $\lambda_{bi} = 0.01$ |

TABLE 5.3: Optimal values of hyperparameters for each model in ML 100K.

Expanding the hyperparameter space would provide insights into how Bayesian optimization behaves in larger spaces. Currently, there is no theory on how to establish good hyperparameter spaces. However, understanding the relationship between models, tasks and model performance, would be practically useful for designing hyperparameter optimization methods. Table 5.3 displays the best hyperparameter values obtained for ML 100K using Bayesian optimization, while Table 5.4 displays the test error.

| **Algorithm** | **RMSE** | **% Improvement** | **F-Score** |
|---|---|---|---|
| Baseline Predictor | 0.9399 | - | 50 |
| $k$-NN$^{\star}_{\text{USERS}}$ | 0.9381 | 0.19 % | 54 |
| $k$-NN$^{\star}_{\text{ITEMS}}$ | 0.9333 | 0.7 % | 53 |
| SVD$_{\text{UNBIASED}}$ | 0.9159 | 2.56 % | 46 |
| SVD$_{\text{BIASED}}$ | 0.9083 | 3.39 % | 63 |

TABLE 5.4: Unbiased test error in $\mathcal{D}_{\text{TEST}}$ for ML 100K.

We evaluated the MAP performance of each recommendation algorithm across varying values of $k$, as presented in Table A.1. The results revealed that matrix factorization algorithms significantly outperformed the other methods.

#### 5.6.2.2 Improved hyperparameter optimization using Bayesian optimization

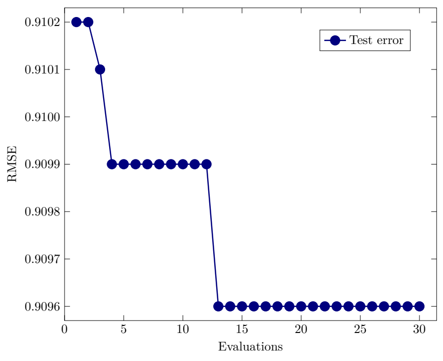
In this part, we derived optimal hyperparameter values for the ML 1M dataset using Bayesian optimization, a more effective method than random search as previously demonstrated in the experiment. The similarity metric employed is consistent with the previous section, and the number of evaluations has been limited to 30 to increase efficiency.

| **Dataset** | **Algorithm** | **Best hyperparameters** |
|---|---|---|
| ML 1M | Baseline Predictor | $n = 39$, $\gamma = 0.002845$, $\lambda = 0.013167$ |
| | $k$-NN$^{\star}_{\text{USERS}}$ | $n = 22$,$\gamma = 0.002116$, $\lambda = 0.019989$, $L = 80$ |
| | $k$-NN$^{\star}_{\text{ITEMS}}$ | $n = 33$, $\gamma = 0.004111$, $\lambda = 0.01$, $L = 98$ |
| | SVD$_{\text{UNBIASED}}$ | $n = 50$, $k = 40$, $\gamma_{pu} = 0.004815$, $\gamma_{qi} = 0.006103$, $\lambda_{pu} = 0.024944$, $\lambda_{qi} = 0.09$ |
| | SVD$_{\text{BIASED}}$ | $n = 50$, $k = 40$, $\gamma_{pu} = 0.004475$, $\gamma_{qi} = 0.009$, $\gamma_{bu} = 0.019$, $\gamma_{bi} = 0.001$, $\lambda_{pu} = 0.046606$, $\lambda_{qi} = 0.09$, $\lambda_{bu} = 0.01$, $\lambda_{bi} = 0.01$ |

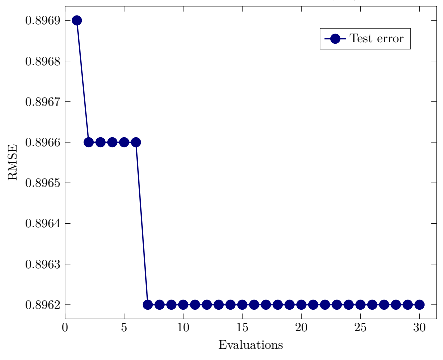
TABLE 5.5: Optimal values of hyperparameters for each model in ML 1M.

Table 5.5 displays the values of the best hyperparameters obtained for ML 1M through Bayesian optimization. Subsequently, Tables 5.6 and A.2 demonstrate the test error and MAP performance, respectively.

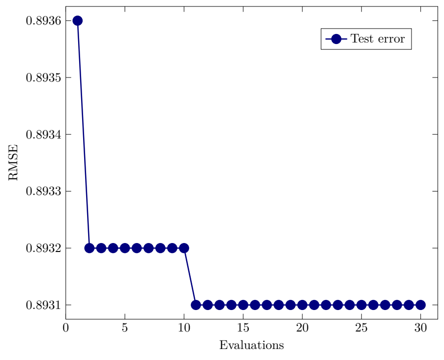
In Figures 5.16a, 5.16b, 5.16c, 5.16d, and 5.16e, we can see that the error has significantly decreased in comparison to the ML 100K dataset. This indicates that by maintaining the same complexity of the model and transitioning from a small dataset to a larger one, the error decreases.



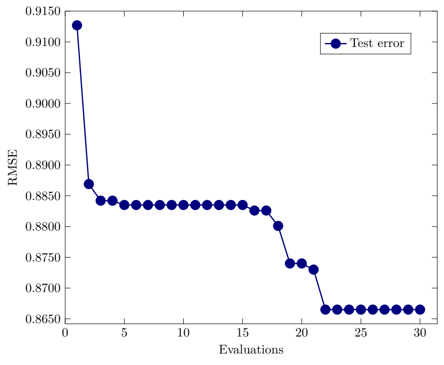
(A) Baseline Predictor.



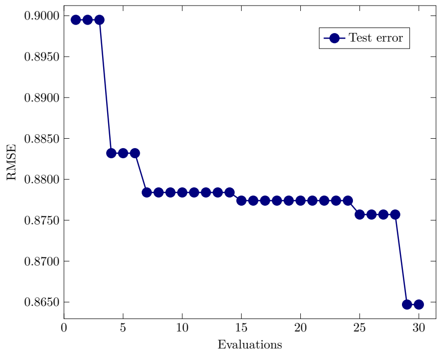
(B) $k$-NN$^{\star}_{\text{USERS}}$



(C) $k$-NN$^{\star}_{\text{ITEMS}}$



(D) SVD$_{\text{UNBIASED}}$



(E) SVD$_{\text{BIASED}}$

FIGURE 5.16: Test error of memory-based and model-based methods with respect to the number of models trained on ML 1M.

#### 5.6.2.3 RMSE analysis

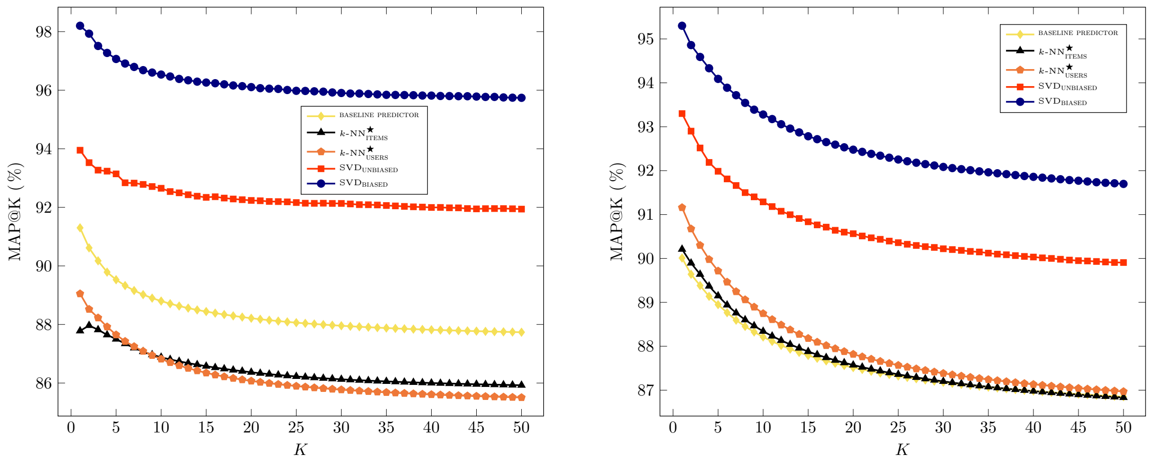
The SVD$_{\text{BIASED}}$ algorithm outperforms all other evaluated algorithms. It achieves an RMSE of 0.9083 for the ML 100K dataset and 0.8538 for the ML 1M dataset, as shown in

| Algorithm | RMSE | % Improvement | F-Score |
|---|---|---|---|
| Baseline Predictor | 0.9095 | - | 54 |
| $k$-NN$^{\bigstar}_{\text{USERS}}$ | 0.8828 | 2.94 % | 57 |
| $k$-NN$^{\bigstar}_{\text{ITEMS}}$ | 0.8798 | 3.28 % | 58 |
| SVD$_{\text{UNBIASED}}$ | 0.8557 | 6.02 % | 56 |
| SVD$_{\text{BIASED}}$ | 0.8538 | 6.24 % | 62 |

TABLE 5.6: Unbiased test error in $\mathcal{D}_{\text{TEST}}$ for ML 1M.

Tables 5.4 and 5.6, respectively. It is important to note that the algorithm's approach differs from other nearest neighbor methods. Matrix factorization algorithms model data structures at a higher level of abstraction to capture user tastes and preferences, while $k$-NN techniques focus on more basic data features. Despite these differences, matrix factorization algorithms are still an important competing models in performance comparisons. It is worth noting the significant difference in improvement percentages achieved between the two datasets, which may be attributed to their size.

#### 5.6.2.4 MAP and F-Score



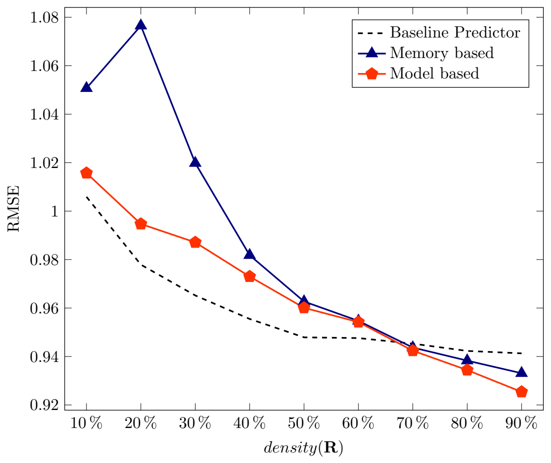
(A) ML 100K (B) ML 1M

FIGURE 5.17: MAP@$K$ performance of each recommendation algorithm.

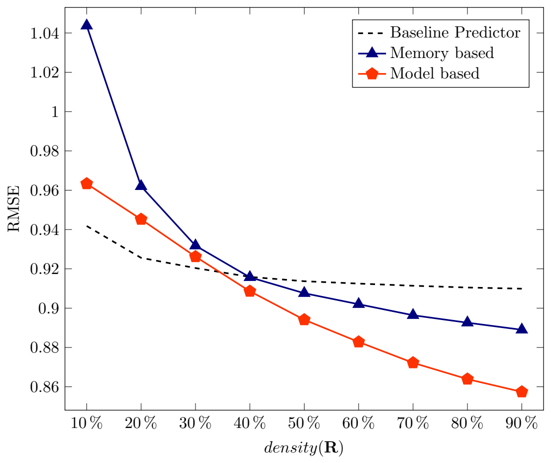
In terms of F-Score, the SVD$_{\text{BIASED}}$ algorithm outperforms all other evaluated algorithms, indicating its potential for achieving a good trade-off between accuracy and scalability. Although model-based methods achieve slightly better F-Scores than memory-based methods in both datasets, the results suggest that the SVD$_{\text{BIASED}}$ algorithm is the most promising candidate for achieving the best overall performance. Notably, the majority of users rated only a few movies in both datasets, with a minimum of 20 ratings per user. This information is crucial for interpreting the average MAP precision values reported in Tables A.1 and A.2. Specifically, for $K < 20$, the MAP@$K$ values are more reliable in reflecting the algorithms' true behavior. On the other hand, evaluating the behavior for values where $K > 20$, as depicted in Figures 5.17a and 5.17b, is more complex. However, the model-based methods exhibit superior MAP precision compared to the memory-based methods in both datasets.

### 5.6.3 Experiment 3

The five algorithms were implemented successfully on the ML 100K and ML 1M datasets. By examining Figures 5.18a and 5.18b, we analyzed the accuracy of these algorithms while varying the percentage of data used for training purposes. It can be observed that all algorithms demonstrate enhanced performance with an increasing percentage of data allocated for training. This behavior aligns with our expectations, as a higher proportion of data in the training set results in an augmented density of the rating matrix, thus providing the algorithm with a greater quantity of information for accurate prediction calculations.



(A) ML 100K



(B) ML 1M

FIGURE 5.18: Evolution of precision for ML 100K and ML 1M according to matrix density.

In our observations, we note that model-based methods exhibit superior performance compared to memory-based methods across all levels of sparsity, while displaying reduced sensitivity to variations in the sparsity level within the rating matrix. However, for low-density conditions, the Baseline Predictor is the most robust.

Memory-based methods exhibit a noteworthy pattern: as density increases, results demonstrate rapid improvement. These algorithms excel in scenarios characterized by relatively high density, yet their accuracy significantly declines when confronted with sparsity. This decline can be attributed to their reliance on a limited subset of available information, specifically, the interaction between items and users. Consequently, as matrix density decreases, the computation of these interactions becomes increasingly challenging, leading to a notable deterioration in overall performance. Furthermore, the memory-based methods encounter a prominent issue known as the "cold start" problem, elaborated upon in Section 2.3.3. Reducing the size of the training set is associated with a substantial decrease in prediction accuracy.

In contrast, model-based methods demonstrate a greater resilience to density variations, displaying a more gradual decline in accuracy. This characteristic can be attributed to the construction phase of the model. During the construction phase of a model-based method, certain details are inherently omitted or simplified. This omission of details can lead to a decrease in accuracy compared to memory-based methods, which directly utilize the available information. However, this trade-off allows model-based methods to achieve scalability and computational efficiency, making them more suitable for handling large datasets and complex recommendation scenarios. In other words, *model-based methods sacrifice some level of accuracy by abstracting or omitting certain specifics in order to create a more efficient and scalable algorithm.* By doing so, these methods can process and analyze vast amounts of data more effectively, which is particularly valuable in scenarios with dense datasets where memory-based approaches may become computationally expensive or impractical. Therefore, the choice between model-based and memory-based methods depends on the specific requirements of the recommendation system, striking a balance between accuracy and efficiency.

# Chapter 6

# Conclusions and Future Work

This chapter provides a summary of the key findings from the previous chapters, followed by a discussion of potential future directions for research.

## 6.1 Summary

In Chapter 2, we presented a comprehensive introduction to the domains of Artificial Intelligence and Machine Learning. Our study focused specifically on recommender systems, with an emphasis on collaborative filtering. Chapter 3 delved into two primary approaches to collaborative filtering: nearest neighbor-based techniques and matrix factorization. In Chapter 4, we highlighted the significance of hyperparameter optimization and conducted a thorough review of state-of-the-art methods, with a particular focus on Bayesian optimization. Lastly, Chapter 5 provided a detailed exposition of our experimental results and a comprehensive analysis of their implications.

All experiments were conducted using two well-established datasets, specifically ML 100K and ML 1M. The evaluation of various algorithms involved the application of metrics such as RMSE, Precision, F-Score and MAP. The first experiment aimed to analyze the influence of hyperparameters on the algorithms. It was demonstrated that the learning rate plays a pivotal role in mitigating overfitting and enhancing prediction accuracy. Furthermore, the use of Pearson similarity centered on the Baseline Predictor yielded superior outcomes when compared to alternative similarity metrics. Employing this metric enabled the consideration of a reduced number of neighbors, consequently decreasing both the "learning" and prediction times associated with memory-based approaches. Notably, our findings aligned with prior research [YWZ+16], highlighting the overall superiority of the item-based approach over the user-based approach, particularly in datasets characterized by a higher number of users than items.

In the second experiment, we conducted a comparative analysis between two prominent techniques for hyperparameter optimization: random search and Bayesian optimization. Our results unequivocally demonstrated the superiority of Bayesian optimization in terms of both efficiency and effectiveness. We showcased the application of Bayesian optimization in effectively fine-tuning the hyperparameters of recommender systems, emphasizing the incorporation of three distinct acquisition functions. It is imperative to emphasize that meticulous hyperparameter tuning assumes critical importance in collaborative filtering, given the need to optimize a computationally demanding loss function, ultimately enabling the attainment of optimal performance.

In the third experiment, our focus was on analyzing the influence of data sparsity on the performance of collaborative filtering algorithms. By comparing the Baseline Predictor with two collaborative filtering algorithms on identical datasets, we uncovered noteworthy insights. Notably, the memory-based approach exhibited increased sensitivity to data sparsity, whereas the model-based approach demonstrated superior robustness and overall accuracy. Remarkably, the Baseline Predictor outperformed both approaches when dealing with exceedingly sparse datasets. This highlights the significance of dataset density as a pivotal factor that significantly impacts the scalability and accuracy of collaborative filtering algorithms.

Our comprehensive findings indicate the pronounced superiority of the model-based approach, specifically the $\text{SVD}_{\text{BIASED}}$ algorithm, over the memory-based approach in relation to accuracy and scalability. This superiority is particularly evident when dealing with larger and sparser datasets. Nevertheless, further rigorous testing is imperative to validate this conclusion across a broader spectrum of datasets with diverse characteristics. Subsequent research endeavors could concentrate on the development of more efficient techniques for effectively managing substantial and sparse datasets. Additionally, evaluating the performance of collaborative filtering algorithms across various domains and applications would contribute valuable insights to the field.

# Appendix A

# MAP Performance

## A.1 MAP performance for ML 100K and 1M datasets

| **Algorithm** | $K$=1 | $K$=2 | $K$=3 | $K$=4 | $K$=5 | $K$=6 | $K$=7 | $K$=8 | $K$=9 | $K$=10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Baseline Predictor | 89.69 | 89.56 | 89.41 | 89.26 | 89.13 | 89.02 | 88.92 | 88.84 | 88.76 | 88.69 |
| $k$-NN$^{\bigstar}_{\text{USERS}}$ | 90.13 | 89.52 | 89.00 | 88.90 | 88.71 | 88.58 | 88.47 | 88.39 | 88.31 | 88.24 |
| $k$-NN$^{\bigstar}_{\text{ITEMS}}$ | 89.37 | 88.81 | 88.50 | 88.31 | 88.14 | 88.01 | 87.89 | 87.78 | 87.68 | 87.59 |
| SVD$_{\text{UNBIASED}}$ | 95.33 | 94.615 | 94.14 | 93.98 | 93.70 | 93.52 | 93.33 | 93.25 | 93.16 | 93.06 |
| SVD$_{\text{BIASED}}$ | 97.56 | 97.45 | 97.41 | 97.185 | 97.05 | 96.84 | 96.73 | 96.59 | 96.50 | 96.47 |

Table A.1: Values of MAP@$K$ for different $K$ in $\mathcal{D}_{\text{TEST}}$ for ML 100K.

| **Algorithm** | $K$=1 | $K$=2 | $K$=3 | $K$=4 | $K$=5 | $K$=6 | $K$=7 | $K$=8 | $K$=9 | $K$=10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Baseline Predictor | 89.70 | 89.41 | 89.08 | 88.85 | 88.60 | 88.46 | 88.30 | 88.15 | 88.02 | 87.91 |
| $k$-NN$^{\bigstar}_{\text{USERS}}$ | 90.41 | 90.01 | 89.65 | 89.38 | 89.13 | 88.93 | 88.75 | 88.59 | 88.46 | 88.34 |
| $k$-NN$^{\bigstar}_{\text{ITEMS}}$ | 91.86 | 91.38 | 90.95 | 90.55 | 90.20 | 89.92 | 89.68 | 89.48 | 89.30 | 89.15 |
| SVD$_{\text{UNBIASED}}$ | 93.99 | 93.29 | 92.86 | 92.56 | 92.24 | 92.02 | 91.83 | 91.67 | 91.51 | 91.37 |
| SVD$_{\text{BIASED}}$ | 95.86 | 95.335 | 94.92 | 94.64 | 94.39 | 94.19 | 93.98 | 93.81 | 93.67 | 93.53 |

Table A.2: Values of MAP@$K$ for different $K$ in $\mathcal{D}_{\text{TEST}}$ for ML 1M.

# Appendix B

# Definitions and Technical Details

In order to enhance the readability of Chapter 2 and considering space limitations, we have decided to include additional material in a separate appendix. While this appendix is not essential for understanding the subsequent sections of this work, it serves two purposes. Firstly, it presents detailed examples on linear and logistic regression in Sections B.1 and B.2, respectively. Secondly, it provides details on the source of prediction errors, namely the bias and variance trade-off, in Sections B.3 and B.4, respectively. Lastly, it provides a step-by-step calculation of Equations (3.39) and (3.41) in Section B.5.

## B.1 Linear regression

To illustrate Definition 2.1 further, we will provide an example of a straightforward ML algorithm, namely linear regression. Essentially, the purpose of linear regression is to develop a system capable of taking a vector $\mathbf{x} \in \mathbb{R}^n$ as input and predicting a scalar $y \in \mathbb{R}$ as its output. The term "linear" indicates that the output of this method is a linear function of the input. In other words, we can express the output, denoted by $\hat{y} = h_{\mathbf{w}}(\mathbf{x})$, as the product of a weight vector $\mathbf{w}$ and the input vector $\mathbf{x}$ as shown in Equation (B.1).

$$\hat{y} = \mathbf{w}^\top \mathbf{x} \tag{B.1}$$

where $\mathbf{w} \in \mathbb{R}^n$ is a vector of parameters. Equation (B.1) represents the prediction that our model makes for the value of $y$ based on the input $\mathbf{x}$. By solving for the weight vector

$\mathbf{w}$, we can develop a linear regression model that optimally predicts the output $y$ for any given input $\mathbf{w}$. Specifically, the coefficient $w_i$ represents the value that is multiplied by the feature $x_i$ before being summed with the contributions from all other features. To effectively perform our task $T$, we define it as the prediction of $y$ from $\mathbf{x}$ using the output $\hat{y}$. Measuring the model's performance $P$ can be accomplished by computing its mean squared error (MSE) on the validation set:

$$\text{MSE}_{\text{VAL}} = \frac{1}{n}\|(\hat{\mathbf{y}}^{(\text{VAL})} - \mathbf{y}^{(\text{VAL})})\|_2^2 \tag{B.2}$$

To create a ML algorithm, the initial step involves designing an algorithm that improves the weights, represented by $\mathbf{w}$, to decrease the MSE of the validation set $\text{MSE}_{\text{VAL}}$. This can be accomplished by enabling the algorithm *to learn and acquire experience by observing a training set*, indicated by $(\mathbf{X}^{(\text{TRAIN})}, \mathbf{y}^{(\text{TRAIN})})$. To attain this objective efficiently, a recommended approach is to minimize the MSE of the training set $\text{MSE}_{\text{TRAIN}}$. Specifically, we can obtain the point at which the gradient of $\text{MSE}_{\text{TRAIN}}$ equals 0 to minimize the $\text{MSE}_{\text{TRAIN}}$.

$$\begin{aligned}
&\nabla_{\mathbf{w}}\text{MSE}_{\text{TRAIN}} = 0\\
&\Rightarrow \nabla_{\mathbf{w}}\frac{1}{n}\|\hat{\mathbf{y}}^{(\text{TRAIN})} - \mathbf{y}^{(\text{TRAIN})}\|_2^2 = 0\\
&\Rightarrow \frac{1}{n}\nabla_{\mathbf{w}}\|\mathbf{X}^{(\text{TRAIN})}\mathbf{w} - \mathbf{y}^{(\text{TRAIN})}\|_2^2 = 0\\
&\Rightarrow \nabla_{\mathbf{w}}\left(\mathbf{X}^{(\text{TRAIN})}\mathbf{w} - \mathbf{y}^{(\text{TRAIN})}\right)^\top\left(\mathbf{X}^{(\text{TRAIN})}\mathbf{w} - \mathbf{y}^{(\text{TRAIN})}\right) = 0\\
&\Rightarrow \nabla_{\mathbf{w}}\left(\mathbf{w}^\top\mathbf{X}^{(\text{TRAIN})\top}\mathbf{X}^{(\text{TRAIN})}\mathbf{w} - 2\mathbf{w}^\top\mathbf{X}^{(\text{TRAIN})\top}\mathbf{y}^{(\text{TRAIN})} + \mathbf{y}^{(\text{TRAIN})\top}\mathbf{y}^{(\text{TRAIN})}\right) = 0\\
&\Rightarrow 2\mathbf{X}^{(\text{TRAIN})\top}\mathbf{X}^{(\text{TRAIN})}\mathbf{w} - 2\mathbf{X}^{(\text{TRAIN})\top}\mathbf{y}^{(\text{TRAIN})} = 0\\
&\Rightarrow \hat{\mathbf{w}} = \left(\mathbf{X}^{(\text{TRAIN})\top}\mathbf{X}^{(\text{TRAIN})}\right)^{-1}\mathbf{X}^{(\text{TRAIN})\top}\mathbf{y}^{(\text{TRAIN})}
\end{aligned} \tag{B.3}$$

Equation (B.3) is usually called the **normal equation** of the OLS (Ordinary Least Squares). It turns out that *solving Equation (B.3) provides a straightforward learning algorithm.* Our prior approach assumed an implicit intercept value of zero, thereby constraining the regression line to pass through the origin. However, this may not always be an appropriate model, and a more general approach would be to allow the model to *learn the intercept value as well.* Consequently, the updated model can be formulated as shown in Equation (B.4).

**Algorithm 18:** Linear regression

```
Input: Training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ containing feature vectors $\mathbf{x}_i$ and labels $y_i$.
Output: Optimal parameter vector $\hat{\mathbf{w}}$.
Construct the matrix $\mathbf{X}$ and the vector $\mathbf{y}$ from the training set, where each $\mathbf{x}$
 includes $x_0 = 1$ bias coordinate, as follows:
```

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & \dots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \dots & x_{m,n} \end{bmatrix}_{m \times n} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}_{m \times 1}$$

```
 if $\mathbf{X}^\top\mathbf{X}$ is invertible then
    Compute the pseudo-inverse $\mathbf{X}^\dagger$ of the matrix $\mathbf{X}$:
        $\mathbf{X}^\dagger = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$    (B.5)
     return Optimal parameter vector $\hat{\mathbf{w}} = \mathbf{X}^\dagger\mathbf{y}$
end if
else
    return No solution exists
end if
```

$$\hat{y} = h_{\hat{\mathbf{w}}}(\mathbf{x}) = \mathbf{w}^\top\mathbf{x} + b \qquad \text{(B.4)}$$

Compared to the SGD optimization algorithm employed in logistic regression (B.2), Algorithm 18 does not demonstrate the conventional attributes of *learning* in the same manner. This is because the vector $\hat{\mathbf{w}}$ is obtained through an analytic solution involving matrix inversion and multiplications, as opposed to iterative learning steps typically associated with the learning process. However, if the vector $\hat{\mathbf{w}}$ demonstrates a satisfactory validation error, we can consider the learning process successful. Linear regression stands out as a unique case wherein we possess an analytic formula for learning that can be easily evaluated. This attribute contributes significantly to the widespread adoption and popularity of this technique.

## B.2 Logistic regression

The task of data classification involves determining the class membership $y_0$ of an unknown data item $x_0$ based on a given dataset $\mathcal{D}$ consisting of data items $\mathbf{x}_i$ with known class memberships $y_i$. For the purpose of this discussion, we will focus on dichotomous classification problems where the class labels $y$ are either 0 or 1. The $\mathbf{x}_i$ typically represent $m$-dimensional vectors, with their components referred to as covariates, independent variables (in statistics), or input variables (in machine learning). In most problem domains, there is no specific functional relationship $y = h(\mathbf{x})$ between $y$ and $\mathbf{x}$. Instead, the relationship between $\mathbf{x}$ and $y$ needs to be described more generally using a probability distribution $p(\mathbf{x}, y)$. Under this framework, it is assumed that the dataset $\mathcal{D}$ consists of independent samples drawn from this probability distribution.

According to statistical decision theory, the optimal decision for class membership is to select the class label $y$ that maximizes the posterior distribution $p(y \mid \mathbf{x})$. There are two distinct approaches to data classification. The first approach focuses solely on a dichotomous distinction between the two classes and assigns class labels (0 or 1) to unknown data items. The second approach aims to *model* $p(y \mid \mathbf{x})$, providing not only a class label for each data item but also a *probability of class membership*. Support vector machines are prominent examples of the first approach. On the other hand, logistic regression, artificial neural networks, $k$-nearest neighbors and decision trees belong to the second approach, although they differ significantly in how they approximate $p(y \mid \mathbf{x})$ from the given data [DOM02]. A logistic regression loss measure is most appropriate in situations where the desired output variable "$y$", of the learning algorithm, is binary-valued (e.g., $y \in \{0, 1\}$). Below, we formally introduce binary logistic regression.

**Definition B.1.** (Binary logistic regression). The objective is to approximate the hypothesis $\hat{y} \approx h_{\hat{\mathbf{w}}}(\mathbf{x}) = y \in \{0, 1\}$. To achieve this, we adopt the following form for $h_{\hat{\mathbf{w}}}(\mathbf{x})$:

$$\begin{aligned} h_{\hat{\mathbf{w}}}(\mathbf{x}) &= \hat{y}(\mathbf{x}, \mathbf{w}) = \psi(\mathbf{x}, \mathbf{w}) \\ &= p(\mathbf{x}, \mathbf{w}) = \frac{1}{1 + \exp(-\hat{y}(\mathbf{x}, \mathbf{w}))} \end{aligned} \tag{B.6}$$

The logistic function $\psi$ squashes inputs from $[-\infty, \infty]$ into the range $[0, 1]$. Equation (B.6) defines the probability, denoted as $p$, that the response variable $\hat{y}$ (which takes binary values of 0 or 1) is equal to 1, given an input pattern $\mathbf{x}$ and a parameter vector $\mathbf{w}$ in the context of logistic regression. In this equation, $\hat{y}(\mathbf{x}, \mathbf{w})$ represents the predicted value of the logistic regression model for the input pattern $\mathbf{x}$ and parameter vector $\mathbf{w}$. It is obtained by performing a linear combination of the features in $\mathbf{x}$ weighted by the elements of $\mathbf{w}$.

The expression $\exp(-\hat{y}(\mathbf{x},\mathbf{w}))$ calculates the exponential of the negative of $\hat{y}(\mathbf{x},\mathbf{w})$. The negative sign is applied to invert the predicted value, making it negative if it was positive and positive if it was negative. The term $[1+\exp(-\hat{y}(\mathbf{x},\mathbf{w}))]^{-1}$ is the inverse of the quantity inside the brackets. This is done to ensure that the resulting value lies within the range $[0,1]$, which is necessary for interpreting it as a probability. Therefore, the equation computes the probability $p(\mathbf{x},\mathbf{w})$ by taking the predicted value $\hat{y}(\mathbf{x},\mathbf{w})$, applying a transformation through the exponential function, and then normalizing it to the range $[0,1]$ using the inverse operation. *This probability represents the estimated likelihood of the response variable $y$ being equal to 1 given the input pattern* $\mathbf{x}$ *and the parameter vector* $\mathbf{w}$ in logistic regression. Next, $\hat{y}(\mathbf{x},\mathbf{w})$ is defined as:

$$\hat{y}(\mathbf{x},\mathbf{w}) = \mathbf{w}^\top\mathbf{x} = \mathbf{w}^\top\begin{bmatrix}\mathbf{x}^\top & 1\end{bmatrix}^\top \tag{B.7}$$

Equation (B.7) represents the predicted value, denoted as $\hat{y}$, in a logistic regression model given an input pattern $\mathbf{x}$ and a parameter vector $\mathbf{w}$. In this equation, $\mathbf{w}$ represents the parameter vector of the logistic regression model, which contains the weights assigned to each feature or input variable. The superscript "$\top$" denotes the transpose operation, converting a row vector into a column vector. The feature vector of the data point is denoted by $\mathbf{x}$.

The expression $\begin{bmatrix}\mathbf{x}^\top & 1\end{bmatrix}$ is a concatenation of the transpose of $\mathbf{x}$ and the scalar value 1. It is a way to incorporate a bias term or intercept into the logistic regression model. The bias term allows the model to account for the influence of the intercept or baseline value. The dot product $\mathbf{w}^\top\begin{bmatrix}\mathbf{x}^\top & 1\end{bmatrix}^\top$ represents the linear combination of the features and weights. It calculates the weighted sum of the elements in $\mathbf{x}$, where each element is multiplied by the corresponding weight in $\mathbf{w}$. The bias term, represented by the appended 1, is also included in the calculation. The result is a single scalar value, which represents the predicted value $\hat{y}$ of the logistic regression model for the given input pattern $\mathbf{x}$ and parameter vector $\mathbf{w}$. The loss function $S$ is chosen such that:

$$S(y,\hat{y}) = -\left[y\log(p(\mathbf{x},\mathbf{w})) + (1-y)\log(1-p(\mathbf{x},\mathbf{w}))\right] \tag{B.8}$$

Logistic regression employs the logistic loss function (Equation (B.8)) to evaluate the effectiveness of a hypothesis $h_{\mathbf{w}} \in \mathcal{H}^n$:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^\top[\mathbf{x}^\top 1]^\top \tag{B.9}$$

This evaluation is based on a labeled training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$. The logistic loss function measures the discrepancy between the predicted output and the true label, allowing for the assessment of the model's performance. Logistic regression aims to minimize the empirical risk by minimizing the average logistic loss:

$$\hat{R}_{\mathcal{D}}(h_{\mathbf{w}}) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(p(\mathbf{x}_i, \mathbf{w})) + (1 - y_i)\log(1 - p(\mathbf{x}_i, \mathbf{w}))] \tag{B.10}$$

### B.2.1 Parameter estimation

Logistic regression aims to learn a linear hypothesis, denoted as $h_{\hat{\mathbf{w}}}(\mathbf{x})$, by minimizing the average logistic loss (Equation (B.10)). This loss is calculated for a given dataset $\mathcal{D}$, consisting of input feature vectors $\mathbf{x}_i$ in a real-valued space $\mathbb{R}^n$ and binary labels $y_i$ that can take values of either 0 or 1. The objective is to solve a smooth optimization problem by minimizing the average logistic loss for the dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, as shown in Equation (B.11).

$$\hat{\mathbf{w}} = \underset{\mathbf{w}\in\mathbb{R}^n}{\operatorname{argmin}}\, h(\mathbf{w}) \tag{B.11}$$

with

$$h(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(p(\mathbf{x}_i, \mathbf{w})) + (1 - y_i)\log(1 - p(\mathbf{x}_i, \mathbf{w}))] \tag{B.12}$$

The function $h(\mathbf{w})$ in Equation (B.12) represents the empirical risk, denoted as $\hat{R}_{\mathcal{D}}(h_{\mathbf{w}})$, which is incurred by the hypothesis $h_{\mathbf{w}}$ when applied to the datapoints in the dataset $\mathcal{D}$. After obtaining the optimal weight vector $\hat{\mathbf{w}}$ that minimizes Equation (B.10), the classification of instances is determined based on their respective features $x$ using the following procedure:

$$\hat{y} = \begin{cases} 1 & \text{if } h_{\hat{\mathbf{w}}}(\mathbf{x}) \geq 0.5, \\ 0 & \text{if } h_{\hat{\mathbf{w}}}(\mathbf{x}) < 0.5. \end{cases} \tag{B.13}$$

In order to predict a label $\hat{y} \in \{0, 1\}$, we compare the hypothesis value $h_{\hat{\mathbf{w}}}(\mathbf{x})$ with a *threshold*, namely, if the value of the hypothesis $h_{\hat{\mathbf{w}}}(\mathbf{x})$ is greater than or equal to 0.5, the instance is classified as $\hat{y} = 1$. Conversely, if $h_{\hat{\mathbf{w}}}(\mathbf{x})$ is less than 0.5, the instance is classified as $\hat{y} = 0$.

To train logistic regression, our approach is similar to that of linear regression, where the objective is to minimize the error function to zero:

$$\nabla_{\mathbf{w}} \text{MSE}_{\text{TRAIN}} = 0 \tag{B.14}$$

However, the gradient $\nabla_{\mathbf{w}} \text{MSE}_{\text{TRAIN}}$ for logistic regression is not straightforward to manipulate, which makes it impractical to obtain an analytical solution by setting it to zero. To overcome this, we will use the SGD optimization technique, which *iteratively approximates the gradient to zero.* The whole process can be seen in Algorithm 19.

**Algorithm 19:** Logistic regression with SGD

```
Input: Training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ containing feature vectors $\mathbf{x}_i \in \mathbb{R}^n$, labels
       $y_i \in \{0, 1\}$, the learning rate $\gamma > 0$, maximum number of epochs $E$, and
       convergence threshold $\epsilon$.
Output: $\mathbf{w}_r$ or $\mathbf{w}_E$, which approximates a solution $\hat{\mathbf{w}}$.
begin
    $\mathbf{w}_1 \leftarrow 0$
    $r \leftarrow 1$ // Initialize iteration counter
    converged $\leftarrow$ False // Convergence flag
    while $r \leq E$ do
        Randomly select an index $\hat{i}_r$ from $\mathcal{D}$
        $\nabla f_{\hat{i}_r}(\mathbf{w}_r) \leftarrow -\frac{y_{\hat{i}_r}}{1+\exp\left(y_{\hat{i}_r}\mathbf{w}_r^\top \mathbf{x}_{\hat{i}_r}\right)}\mathbf{x}_{\hat{i}_r}$ // Compute the gradient approximation
        $\mathbf{w}_{r+1} \leftarrow \mathbf{w}_r - \gamma \nabla f_{\hat{i}_r}(\mathbf{w}_r)$ // Update the weight vector
        $r \leftarrow r + 1$ // Increase iteration counter
        if $\|\mathbf{w}_r - \mathbf{w}_{r-1}\| < \epsilon$ then
            converged $\leftarrow$ True // Check convergence
            break
        end if
    end while
    if converged then
        return $\mathbf{w}_r$ // Return the weight vector when converged
    else
        return $\mathbf{w}_E$ // Return the weight vector at the maximum number of epochs
    end if
end
```

## B.3 Source of prediction errors

The accuracy of $\hat{f}$ as a prediction for $y$ depends on two quantities, which we will call the *reducible* error and the *irreducible* error "$\epsilon$". In general, $\hat{f}$ will not be a perfect estimate for $f$, and this inaccuracy will introduce some error. This error is reducible because we can potentially improve the accuracy of $\hat{f}$ by using the most appropriate ML algorithm to estimate $f$. However, even with a perfect estimate of $f$, $y$ may not be predicted accurately due to real-world factors such as inaccurate sensors or missing unmeasurable features. This is because $y$ is also a function of $\epsilon$, which, by definition, cannot be predicted using $x$. Therefore, variability associated with $\epsilon$ also affects the accuracy of our predictions. This is known as the irreducible error, because no matter how well we estimate $f$, we cannot reduce the error introduced by $\epsilon$.

## B.4 The bias-variance trade-off

A crucial tool for comprehending the generalization performance of algorithms is the *bias-variance decomposition*, which breaks down the anticipated generalization error of learning algorithms. Despite the training samples being drawn from the same distribution, different training sets often yield varying learning outcomes. To illustrate this, let us consider $\mathbf{x}$ as a testing sample, $y_{\mathcal{D}}$ as the label of $\mathbf{x}$ in the data set $\mathcal{D}$, $y$ as the ground-truth label of $\mathbf{x}$ and $f(\mathbf{x};\ \mathcal{D})$ as the output of $\mathbf{x}$ predicted by the model $f$ trained on $\mathcal{D}$. In the case of regression problems, the expected prediction of a learning algorithm is as follows:

$$\hat{f}(\mathbf{x}) = \mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\ \mathcal{D})] \tag{B.15}$$

The variance of using different equal-sized training sets is:

$$var(\mathbf{x}) = \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x};\ \mathcal{D}) - \hat{f}(\mathbf{x}))^2]. \tag{B.16}$$

The noise is

$$\epsilon^2 = \mathbb{E}_{\mathcal{D}}[(y_{\mathcal{D}} - y)^2] \tag{B.17}$$

The difference between the expected output and the ground-truth label is called bias, that is:

$$bias^2(\mathbf{x}) = (\hat{f}(\mathbf{x}) - y)^2 \tag{B.18}$$

For the sake of clarity in our discussion, we make the assumption that the expected value of noise is zero, represented as $\mathbb{E}_{\mathcal{D}}[(y_{\mathcal{D}} - y)] = 0$. Through expanding and combining the polynomial expressions, we can proceed to decompose the anticipated generalization error in the following manner:

$$\begin{aligned}
\text{Error}(f;\ \mathcal{D}) &= \mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\ \mathcal{D}) - y_{\mathcal{D}})^2] \\
&= \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x};\ \mathcal{D}) - \hat{f}(\mathbf{x}) + \hat{f}(\mathbf{x}) - y_{\mathcal{D}})^2] \Leftarrow \\
&= \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x};\ \mathcal{D}) - \hat{f}(\mathbf{x}))^2] + \mathbb{E}_{\mathcal{D}}[\hat{f}(\mathbf{x}) - y_{\mathcal{D}})^2] \\
&\quad + \mathbb{E}_{\mathcal{D}}[2(f(\mathbf{x};\ \mathcal{D}) - \hat{f}(x))(\hat{f}(\mathbf{x}) - y_{\mathcal{D}})] \Leftarrow \\
&= \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x};\ \mathcal{D}) - \hat{f}(\mathbf{x}))^2] + \mathbb{E}_{\mathcal{D}}[(\hat{f}(\mathbf{x}) - y_{\mathcal{D}})^2] \Leftarrow \\
&= \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x};\ \mathcal{D}) - \hat{f}(\mathbf{x}))^2] + \mathbb{E}_{\mathcal{D}}[(\hat{f}(\mathbf{x}) - y + y - y_{\mathcal{D}})^2] \Leftarrow \\
&= \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x};\ \mathcal{D}) - \hat{f}(\mathbf{x}))^2] + \mathbb{E}_{\mathcal{D}}[(\hat{f}(\mathbf{x}) - y)^2] \\
&\quad + \mathbb{E}_{\mathcal{D}}[(y - y_{\mathcal{D}})^2] + 2\mathbb{E}_{\mathcal{D}}[(\hat{f}(\mathbf{x}) - y)(y - y_{\mathcal{D}})] \Leftarrow \\
&= \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x};\ \mathcal{D}) - \hat{f}(\mathbf{x}))^2] + (\hat{f}(\mathbf{x}) - y)^2 + \mathbb{E}_{\mathcal{D}}[(y_{\mathcal{D}} - y)^2] \Leftarrow
\end{aligned} \tag{B.19}$$

The decomposition of the generalization error into bias, variance, and noise is expressed as follows:

$$\text{Error}(f;\ \mathcal{D}) = bias^2(\mathbf{x}) + var(\mathbf{x}) + \epsilon^2 \tag{B.20}$$

which means the generalization error can be decomposed into the sum of bias, variance and noise. This implies that the generalization error can be decomposed into the combined components of bias, variance, and noise. **Bias**, as expressed in Equation (B.18), is a metric that quantifies the dissimilarity between the expected prediction made by a learning algorithm and the ground-truth label associated with the data. In other words, it measures how well the algorithm can fit the training data. A low bias indicates that the algorithm can closely approximate the true relationship between the input features and the output labels. On the other hand, **variance**, as defined in Equation (B.16), captures the variability in the performance of a learning algorithm when trained on different datasets of the same size. It reflects how much the algorithm's predictions fluctuate due to variations in the training data. Higher variance implies that the algorithm is highly sensitive to the specific instances in the training set, which can lead to overfitting or underfitting. Additionally, **noise**, represented by Equation (B.17), refers to the inherent randomness or irreducible error present in the data. It represents the minimum generalization error that any learning algorithm would encounter for a given task, irrespective of its complexity. Noise characterizes the intrinsic difficulty of the learning problem itself, which cannot be eliminated by improving the algorithm or the quality of the data.

The bias-variance trade-off is a fundamental challenge in ML that involves a trade-off between bias and variance. In order to understand this dilemma, let us consider Figure B.1, which provides a visual representation of the concept.

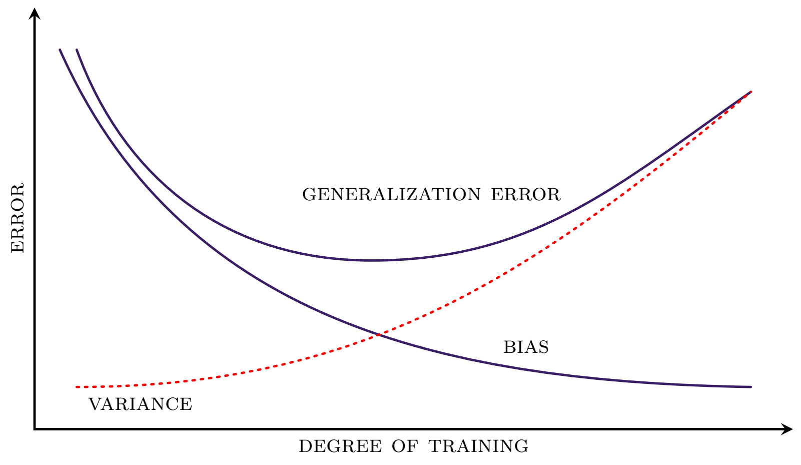


Figure B.1: Relationships between generalization error, bias and variance.

When we train a ML model, we have the ability to control the degree of training. If we limit the degree of training, the model will be undertrained. This means that it has limited capacity to capture the underlying patterns and relationships in the data. As a result, the model's predictions may deviate significantly from the true values, leading to a high level of bias. In an undertrained model, bias dominates the generalization error. As we increase the degree of training, the model's fitting capacity improves. It starts to learn from the data and adapt to its complexities. With more training, the model becomes more flexible and can capture intricate patterns present in the data. However, this increased flexibility comes at a cost; the model becomes more sensitive to variations and noise in the training data, which leads to higher variability (variance) in its predictions. At a certain point, with a substantial amount of training, the model's fitting capability becomes very strong. It can almost perfectly fit the training data, capturing even the smallest fluctuations and peculiarities present in the data. However, this high level of sensitivity to the training data comes at the expense of generalization to new, unseen data. Even slight perturbations or variations in the training data can lead to significant changes in the model's predictions. In this case, the model becomes too specialized to the training data and fails to generalize well to new data (overfitting).

## B.5 Differentiation

In this section, we provide a step-by-step calculation of two important differentiations: $\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{\lambda}{2}||\mathbf{p}_u||^2\right]$ and $\frac{\partial}{\partial \mathbf{q}_i}\left[\frac{\lambda}{2}||\mathbf{q}_i||^2\right]$. By working through these examples, we aim to illustrate the process of differentiation and provide a clear understanding of how to compute these derivatives.

### B.5.1 Differentiation of $\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{\lambda}{2}||\mathbf{p}_u||^2\right]$

To explain the outcome of this calculation step by step, let us break it down. Starting with the expression:

$$\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{\lambda}{2}||\mathbf{p}_u||^2\right] \tag{B.21}$$

Here, we are taking the partial derivative with respect to the vector $\mathbf{p}_u$. We can apply the derivative to each component of the vector $\mathbf{p}_u$ separately, assuming that $\mathbf{p}_u$ is a row vector. The expression $\frac{\lambda}{2}||\mathbf{p}_u||^2$ can be expanded as $\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top$, where $\mathbf{p}_u^\top$ represents the transpose of the vector $\mathbf{p}_u$. Let us prove that the expression $\frac{\lambda}{2}||\mathbf{p}_u||^2$ is equal to $\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top$, we can go through the following steps:

Let $\mathbf{p}_u = [p_{u1}, p_{u2}, \ldots, p_{un}]$ be a 1-dimensional row vector of length $n$. The transpose of $\mathbf{p}_u$ is denoted as $\mathbf{p}_u^\top$, resulting in a column vector:

$$\mathbf{p}_u^\top = \begin{bmatrix} p_{u1} \\ p_{u2} \\ \vdots \\ p_{un} \end{bmatrix}$$

Now, let us compute the expressions $\frac{\lambda}{2}\|\mathbf{p}_u\|^2$ and $\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top$ and compare them. Let us start with the expression $\frac{\lambda}{2}\|\mathbf{p}_u\|^2$. The Euclidean norm $\|\mathbf{p}_u\|$ of a row vector is computed by taking the square root of the sum of the squares of its elements. In this case, we have:

$$\|\mathbf{p}_u\| = \sqrt{p_{u1}^2 + p_{u2}^2 + \ldots + p_{un}^2}$$

Squaring this norm yields:

$$\|\mathbf{p}_u\|^2 = p_{u1}^2 + p_{u2}^2 + \ldots + p_{un}^2$$

Multiplying by $\frac{\lambda}{2}$ scales the result by a constant factor, the expression becomes: $\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top$. Multiplying the row vector $\mathbf{p}_u$ with its column vector transpose $\mathbf{p}_u^\top$ gives us a matrix. The entry at position $(i, j)$ of this matrix is computed by multiplying the $i$-th element of $\mathbf{p}_u$ with the $j$th element of $\mathbf{p}_u^\top$:

$$\left(\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top\right)_{ij} = p_{ui} \cdot p_{uj}$$

Since $\mathbf{p}_u$ is a row vector, $p_{ui}$ and $p_{uj}$ represent individual elements of the vector. We can rewrite this matrix by expanding the multiplication as follows:

$$\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top = \frac{\lambda}{2}\begin{bmatrix} p_{u1} \\ p_{u2} \\ \vdots \\ p_{un} \end{bmatrix}\begin{bmatrix} p_{u1} & p_{u2} & \ldots & p_{un} \end{bmatrix} = \frac{\lambda}{2}\begin{bmatrix} p_{u1}^2 & p_{u1}p_{u2} & \cdots & p_{u1}p_{un} \\ p_{u2}p_{u1} & p_{u2}^2 & \cdots & p_{u2}p_{un} \\ \vdots & \vdots & \ddots & \vdots \\ p_{un}p_{u1} & p_{un}p_{u2} & \cdots & p_{un}^2 \end{bmatrix}$$

If we focus on the diagonal elements of this matrix, we can see that they correspond to the squares of the elements of the row vector $\mathbf{p}_u$:

$$\left(\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top\right)_{ii} = p_{ui}^2 \tag{B.22}$$

The expression $(\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top)_{ii}$ represents the $i$-th diagonal element of the matrix resulting from the multiplication of $\mathbf{p}_u\mathbf{p}_u^\top$. Specifically, $(\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top)_{ii}$ corresponds to the element at position $(i, i)$ in the matrix. On the right side of the equation $p_{ui}^2$ represents the square of the $i$-th element of the row vector $\mathbf{p}_u$. This equation demonstrates that the diagonal elements of $\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top$ are equal to the squares of the elements of $\mathbf{p}_u$. Equation (B.22) matches the result we obtained in the first expression. Therefore, we can conclude that $\frac{\lambda}{2}\|\mathbf{p}_u\|^2$ and $\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top$ are equivalent expressions when $\mathbf{p}_u$ is a row vector.

Let us now continue with the differentiation process. Taking the derivative of $\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top$ with respect to $\mathbf{p}_u$ is equivalent to taking the derivative of each component of $\mathbf{p}_u$ separately. Since the norm $\|\mathbf{p}_u\|^2$ is a scalar, its derivative with respect to $\mathbf{p}_u$ will be zero. We can now calculate the derivative of $\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top$ with respect to each component of $\mathbf{p}_u$:

$$\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top\right] = \frac{\lambda}{2}\frac{\partial}{\partial \mathbf{p}_u}\left[\mathbf{p}_u\mathbf{p}_u^\top\right] \tag{B.23}$$

Applying the derivative to each component of $\mathbf{p}_u$:

$$\frac{\partial}{\partial \mathbf{p}_u}\left[\mathbf{p}_u\mathbf{p}_u^\top\right] = \begin{bmatrix} \frac{\partial}{\partial \mathbf{p}_u[1]}\left[\mathbf{p}_u\mathbf{p}_u^\top\right] \\ \frac{\partial}{\partial \mathbf{p}_u[2]}\left[\mathbf{p}_u\mathbf{p}_u^\top\right] \\ \vdots \\ \frac{\partial}{\partial \mathbf{p}_u[n]}\left[\mathbf{p}_u\mathbf{p}_u^\top\right] \end{bmatrix} \tag{B.24}$$

where $\mathbf{p}_u[i]$ denotes the $i$-th component of vector $\mathbf{p}_u$, and $n$ is the total number of components in $\mathbf{p}_u$. Taking the derivative of $\mathbf{p}_u\mathbf{p}_u^\top$ with respect to each component of $\mathbf{p}_u$:

$$\frac{\partial}{\partial \mathbf{p}_u[i]}\left[\mathbf{p}_u\mathbf{p}_u^\top\right] = 2\mathbf{p}_u[i] \tag{B.25}$$

This result comes from applying the chain rule of differentiation and recognizing that each component of $\mathbf{p}_u$ is independent of the others. Finally, substituting the derivative of $\mathbf{p}_u\mathbf{p}_u^\top$ back into the original expression:

$$\frac{\partial}{\partial \mathbf{p}_u}\left[\frac{\lambda}{2}\mathbf{p}_u\mathbf{p}_u^\top\right] = \frac{\lambda}{2}\begin{bmatrix} 2\mathbf{p}_u[1] \\ 2\mathbf{p}_u[2] \\ \vdots \\ 2\mathbf{p}_u[n] \end{bmatrix} = \lambda\mathbf{p}_u \tag{B.26}$$

The final result is $\lambda\mathbf{p}_u$, where $\lambda$ is a constant and $\mathbf{p}_u$ is the vector with its components multiplied by 2.

### B.5.2 Differentiation of $\frac{\partial}{\partial \mathbf{q}_i}\left[\frac{\lambda}{2}\|\mathbf{q}_i\|^2\right]$

To explain the outcome of this calculation step by step, let us break it down. Starting with the expression:

$$\frac{\partial}{\partial \mathbf{q}_i}\left[\frac{\lambda}{2}\|\mathbf{q}_i\|^2\right] \tag{B.27}$$

Here, we are taking the partial derivative with respect to the vector $\mathbf{q}_i$. We can apply the derivative to each component of the vector $\mathbf{q}_i$ separately, assuming that $\mathbf{q}_i$ is a column vector. The expression $\frac{\lambda}{2}\|\mathbf{q}_i\|^2$ can be expanded as $\frac{\lambda}{2}\mathbf{q}_i^\top\mathbf{q}_i$, where $\mathbf{q}_i^\top$ represents the transpose of the vector $\mathbf{q}_i$.

Taking the derivative of $\frac{\lambda}{2}\mathbf{q}_i^\top\mathbf{q}_i$ with respect to $\mathbf{q}_i$ is equivalent to taking the derivative of each component of $\mathbf{q}_i$ separately. Since the norm $\|\mathbf{q}_i\|^2$ is a scalar, its derivative with respect to $\mathbf{q}_i$ will be zero. We can now calculate the derivative of $\frac{\lambda}{2}\mathbf{q}_i^\top\mathbf{q}_i$ with respect to each component of $\mathbf{q}_i$:

$$\frac{\partial}{\partial \mathbf{q}_i}\left[\frac{\lambda}{2}\mathbf{q}_i^\top\mathbf{q}_i\right] = \frac{\lambda}{2}\frac{\partial}{\partial \mathbf{q}_i}\left[\mathbf{q}_i^\top\mathbf{q}_i\right] \tag{B.28}$$

Applying the derivative to each component of $\mathbf{q}_i$:

$$\frac{\partial}{\partial \mathbf{q}_i}\left[\mathbf{q}_i^\top\mathbf{q}_i\right] = \begin{bmatrix} \frac{\partial}{\partial \mathbf{q}_i[1]}\left[\mathbf{q}_i^\top\mathbf{q}_i\right] \\ \frac{\partial}{\partial \mathbf{q}_i[2]}\left[\mathbf{q}_i^\top\mathbf{q}_i\right] \\ \vdots \\ \frac{\partial}{\partial \mathbf{q}_i[n]}\left[\mathbf{q}_i^\top\mathbf{q}_i\right] \end{bmatrix} \tag{B.29}$$

where $\mathbf{q}_i[i]$ denotes the $i$-th component of vector $\mathbf{q}_i$ and $n$ is the total number of components in $\mathbf{q}_i$. Taking the derivative of $\mathbf{q}_i^\top \mathbf{q}_i$ with respect to each component of $\mathbf{q}_i$:

$$\frac{\partial}{\partial \mathbf{q}_i[i]} \left[\mathbf{q}_i^\top \mathbf{q}_i\right] = 2\mathbf{q}_i[i] \tag{B.30}$$

This result comes from applying the chain rule of differentiation and recognizing that each component of $\mathbf{q}_i$ is independent of the others. Finally, substituting the derivative of $\mathbf{q}_i^\top \mathbf{q}_i$ back into the original expression:

$$\frac{\partial}{\partial \mathbf{q}_i} \left[\frac{\lambda}{2}\mathbf{q}_i^\top \mathbf{q}_i\right] = \frac{\lambda}{2} \begin{bmatrix} 2\mathbf{q}_i[1] \\ 2\mathbf{q}_i[2] \\ \vdots \\ 2\mathbf{q}_i[n] \end{bmatrix} = \lambda \mathbf{q}_i \tag{B.31}$$

The final result is $\lambda \mathbf{q}_i$, where $\lambda$ is a constant and $\mathbf{q}_i$ is the vector with its components multiplied by 2.

# Appendix C

# Numerical Examples

## C.1 Baseline Predictor

To illustrate how the algorithm works, we present a toy example. The example uses a matrix, denoted as **R**, that contains forty hypothetical ratings on five movies from ten users. The matrix has five columns labeled A through E and ten rows labeled 1 through 10. The density of **R** is 80%. We randomly select thirty ratings as the training set, while the remaining ten ratings, shown in boldface, form the test set. The missing ratings are indicated by "-".

$$\mathbf{R} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 5 & 4 & 4 & - & \mathbf{5} \\ - & 3 & 5 & \mathbf{3} & 4 \\ 5 & 2 & - & \mathbf{2} & 3 \\ - & \mathbf{2} & 3 & 1 & 2 \\ 4 & - & \mathbf{5} & 4 & 5 \\ \mathbf{5} & 3 & - & 3 & 5 \\ 3 & \mathbf{2} & 3 & 2 & - \\ 5 & \mathbf{3} & 4 & - & 5 \\ \mathbf{4} & 2 & 5 & 5 & - \\ \mathbf{5} & - & 5 & 3 & 4 \end{bmatrix} \end{array}$$

Figure C.1 User-movie ratings matrix.

The average rating of the training set is $\bar{r} = 3.83$. We assume that biases $b_u$ and $b_i$ were already obtained using Algorithm 4. For the optimal biases of users we obtained,

$$\mathbf{b}_u^* = [0.62\ \ 0.42\ \ -0.28\ \ -1.78\ \ 0.52\ \ 0.49\ \ -1.24\ \ 0.45\ \ 0.40\ \ 0.23]\,,$$

for optimal biases of movies we got,

$$\mathbf{b}_i^* = [0.72\ \ -1.20\ \ 0.60\ \ -0.60\ \ 0.33]\,.$$

These values quantify the intuition of what we observe from the training data. For example:

- Users $1, 2, 5, 6$, and 8 tend to give higher ratings.
- Users 4 and 7 tend to give lower ratings.
- Movies A and C tend to receive higher ratings.
- Movies B and D tend to receive lower ratings.

We clipped any predicted rating lower than 1 to 1 and any higher than 5 to 5. Let us take $\hat{r}_{8E} = \bar{r} + b_8 + b_E = 3.83 + 0.45 + 0.33 = 4.61$. The rating matrix estimated by the Baseline Predictor (after clipping) is as follows:

$$\hat{\mathbf{R}} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 5 & 3.09 & 4.90 & - & \mathbf{4.62} \\ - & 2.89 & 4.69 & \mathbf{3.49} & 4.42 \\ 4.10 & 2.19 & - & \mathbf{2.78} & 3.71 \\ - & \mathbf{1.00} & 2.49 & 1.29 & 2.22 \\ 4.90 & - & \mathbf{4.79} & 3.58 & 4.51 \\ \mathbf{4.88} & 2.96 & - & 3.56 & 4.48 \\ 3.15 & \mathbf{1.23} & 3.03 & 1.82 & - \\ 4.84 & \mathbf{2.92} & 4.72 & - & 4.61 \\ \mathbf{4.84} & 2.92 & 4.72 & 3.51 & - \\ \mathbf{4.61} & - & 4.49 & 3.29 & 4.22 \end{bmatrix} \end{array}$$

FIGURE C.2 Rating matrix estimated by the Baseline Predictor.

We calculated the **rmse** between $\mathbf{R}$ and $\hat{\mathbf{R}}$ using Equation (2.12) and obtained 0.5147 for the training set and 0.5923 for the test set.

## C.2 $k$-NN$^{\star}_{\text{ITEMS}}$

We will illustrate an example using the matrix $\mathbf{R}$ (C.1). Before computing the similarity between the columns of the matrix, we center each row in $\mathbf{R}$ with the Baseline Predictor. This creates a matrix $\tilde{\mathbf{R}}$ that is also centered with the Baseline Predictor. To calculate similarity, we use Equation (3.24) with $\xi = 0$. We use "**?**" to indicate test set entries and "-" to denote unavailable ones (e.g., user 9 never rated movie E).

Once the ratings are centered with the Baseline Predictor, we use the same metric to calculate the similarity between movies, represented in $\tilde{\mathbf{R}}$, as the cosine in Equation (3.22). For example, let us calculate the similarity between movie **B** and **C**. According to the training data in $\tilde{\mathbf{R}}$ shown in Figure C.3.

$$\tilde{\mathbf{R}} = \mathbf{R} - \hat{\mathbf{R}} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 0.91 & -0.90 & - & \textbf{?} \\ - & 0.11 & 0.31 & \textbf{?} & -0.42 \\ 0.90 & -0.19 & - & \textbf{?} & -0.71 \\ - & \textbf{?} & 0.51 & -0.29 & -0.22 \\ -0.90 & - & \textbf{?} & 0.42 & 0.49 \\ \textbf{?} & 0.040 & - & -0.56 & 0.52 \\ -0.15 & \textbf{?} & -0.031 & 0.18 & - \\ 0.16 & \textbf{?} & -0.72 & - & 0.39 \\ \textbf{?} & -0.87 & 0.33 & 0.54 & - \\ \textbf{?} & - & 0.51 & -0.29 & -0.22 \end{bmatrix} \end{array}$$

FIGURE C.3 Matrix of estimated ratings centered with the Baseline Predictors.

Users 1, 2, and 9 rated both movies. Thus, we have:

$$d_{BC} = \frac{\tilde{r}_{1B}\tilde{r}_{1C} + \tilde{r}_{2B}\tilde{r}_{2C} + \tilde{r}_{9B}\tilde{r}_{9C}}{\sqrt{(\tilde{r}^2_{1B} + \tilde{r}^2_{2B} + \tilde{r}^2_{9B})(\tilde{r}^2_{1C} + \tilde{r}^2_{2C} + \tilde{r}^2_{9C})}} \tag{C.1}$$

$$= \frac{(0.91 \times -0.90) + (0.11 \times 0.31) + (-0.87 \times 0.33)}{\sqrt{(0.91^2 + 0.11^2 + 0.87^2)(0.90^2 + 0.31^2 + 0.33^2)}} = -0.84. \tag{C.2}$$

In the same way we can calculate the complete similarity matrix, symmetric of $5 \times 5$ (where the diagonal entries are not of interest since they refer to the same movie) as shown in Figure C.4. With the movie-movie similarity values calculated in the $\mathbf{D}$ matrix, as shown in Figure C.4, we can carry out the following procedure to calculate $\hat{r}_{ui}$.

$$\mathbf{D} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} - & -0.20 & -0.45 & -0.97 & -0.75 \\ -0.20 & - & -0.84 & -0.73 & 0.51 \\ -0.45 & -0.84 & - & -0.22 & -0.93 \\ -0.97 & -0.73 & -0.22 & - & 0.068 \\ -0.75 & 0.51 & -0.93 & 0.068 & - \end{bmatrix} \end{array}$$

FIGURE C.4 Similarity matrix between movies.

1. We find $L = 2$ neighbors with the highest absolute similarity values $|d_{ik}|$ and $|d_{il}|$.
2. Check if user $u$ has rated the movies $k$ and $l$. If so, we use both, as in Equation (C.3). If the user rated only one of them, we just use that movie. If the user did not rate any of them, then we only use the Baseline Predictor.
3. We calculate the predicted rating:

$$\hat{r}_{ui} = \bar{r} + b_u + b_i + \frac{d_{ik}\tilde{r}_{uk} + d_{il}\tilde{r}_{ul}}{|d_{ik}| + |d_{il}|} \tag{C.3}$$

For instance, we can calculate $\hat{r}_{3D}$. The two closest neighbors for movie D are A and B, whose similarity coefficients are $-0.97$ and $-0.73$, respectively. User 3 has rated A and B. Therefore:

$$\hat{r}_{3D} = \bar{r} + b_3 + b_D + \frac{d_{DA}\tilde{r}_{3A} + d_{DB}\tilde{r}_{3B}}{|d_{DA}| + |d_{DB}|} \tag{C.4}$$

$$= 2.78 + \frac{(-0.97 \times 0.90) + (-0.73 \times -0.19)}{0.97 + 0.73} = 2.35.$$

The predicted rating for the Baseline Predictor is $\hat{r}_{3D} = 2.78$. Similarly, the closest neighbors for movie B are C and D. From the training set we know that user 2 only rated movie C but not D. Thus,

$$\hat{r}_{2B} = (\bar{r} + b_3 + b_B) + \frac{d_{BC}\tilde{r}_{2C}}{|d_{BC}|} = 2.89 + \frac{-0.84 \times 0.31}{0.84} = 2.58. \tag{C.5}$$

Predictions returned by the method $k$-NN$^{\bigstar}_{\text{ITEMS}}$(after clipping) are displayed in matrix C.5. We compute the **rmse** between **R** and $\hat{\mathbf{R}}$ using Equation (2.12) and we get 0.3393 for the training set and 0.5433 for the test set.

$$\hat{\mathbf{R}} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \left[\begin{array}{ccccc} 5 & 3.99 & 3.99 & - & \mathbf{5} \\ - & 2.58 & 4.86 & \mathbf{3.38} & 4.11 \\ 4.81 & 2.19 & - & \mathbf{2.35} & 2.81 \\ - & \mathbf{1.00} & 2.71 & 1.29 & 1.71 \\ 4.46 & - & \mathbf{4.30} & 4.49 & 5.00 \\ \mathbf{4.97} & 3.52 & - & 3.52 & 4.48 \\ 2.97 & \mathbf{1.16} & 3.03 & 1.97 & - \\ 4.28 & \mathbf{3.64} & 4.16 & - & 4.77 \\ \mathbf{4.25} & 2.44 & 5.00 & 4.33 & - \\ \mathbf{4.87} & - & 4.71 & 3.29 & 3.71 \end{array}\right] \end{array}$$

FIGURE C.5 Rating matrix estimated by $k$-NN$^{\bigstar}_{\text{ITEMS}}$.

## C.3 $k$-NN$^{\bigstar}_{\text{USERS}}$

In this section, we present an an illustrative example utilizing the matrix **R** as introduced in Section C.1. User similarities are computed using Equation (3.18) with $\xi = 0$. Initially, the matrix **R** is centered, as illustrated in Figure C.6, by subtracting the Baseline Predictors obtained through the operation $\mathbf{R} - \hat{\mathbf{R}}$.

Subsequently, the user-user similarity matrix **D** is computed, as depicted in Figure C.7, employing Equation (3.18):

$$\begin{aligned} d_{12} &= \frac{(r_{1B} - \tilde{r}_1) \times (r_{2B} - \tilde{r}_2) + (r_{1C} - \tilde{r}_1) \times (r_{2C} - \bar{r}_2)}{\sqrt{(r_{1B} - \tilde{r}_1)^2 + (r_{1C} - \tilde{r}_1)^2} \cdot \sqrt{(r_{2B} - \tilde{r}_2)^2 + (r_{2C} - \tilde{r}_2)^2}} \\ &= \frac{(0.91 \times 0.11) + (-0.90 \times 0.31)}{\sqrt{(0.91)^2 + (-0.90)^2} \cdot \sqrt{(0.11)^2 + (0.31)^2}} = -0.42. \end{aligned} \tag{C.6}$$

$$\tilde{\mathbf{R}} = \mathbf{R} - \hat{\mathbf{R}} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \left[\begin{array}{ccccc} 0 & 0.91 & -0.90 & - & ? \\ - & 0.11 & 0.31 & ? & -0.42 \\ 0.90 & -0.19 & - & ? & -0.71 \\ - & ? & 0.51 & -0.29 & -0.22 \\ -0.90 & - & ? & 0.42 & 0.49 \\ ? & 0.040 & - & -0.56 & 0.52 \\ -0.15 & ? & -0.031 & 0.18 & - \\ 0.16 & ? & -0.72 & - & 0.39 \\ ? & -0.87 & 0.33 & 0.54 & - \\ ? & - & 0.51 & -0.29 & -0.22 \end{array}\right] \end{array}$$

FIGURE C.6 Matrix of estimated ratings centered with the Baseline Predictors.

$$\mathbf{D} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \left[\begin{array}{cccccccccc} - & -0.42 & 0.20 & -1 & 0 & 1 & 0.20 & 0.97 & -0.91 & -1 \\ -0.42 & - & 0.86 & 0.86 & -1 & -0.94 & -1 & -0.90 & 0.02 & 0.86 \\ 0.20 & 0.86 & - & 1 & -0.98 & -0.98 & -1 & -0.27 & 1 & 1 \\ -1 & 0.86 & 1 & - & -0.82 & 0.17 & 1 & -0.99 & 0.03 & 1 \\ 0 & -1 & -0.98 & -0.82 & - & 0.03 & -1 & 0.10 & 1 & -0.97 \\ 1 & -0.94 & -0.98 & 0.17 & 0.03 & - & -1 & 1 & -0.58 & 0.99 \\ 0.20 & -1 & -1 & 1 & -1 & -1 & - & -0.01 & 0.75 & -0.63 \\ 0.97 & -0.90 & -0.37 & -0.99 & 0.10 & 1 & -0.01 & - & -1 & -0.99 \\ -0.91 & 0.02 & 1 & 0.03 & 1 & -0.58 & 0.75 & -1 & - & 0.02 \\ -1 & 0.86 & 1 & 1 & -0.97 & 0.99 & -0.63 & -0.99 & 0.02 & - \end{array}\right] \end{array}$$

FIGURE C.7 Similarity matrix between users.

We utilize Equation (3.26) to compute $\hat{r}_{1E}$. The two closest neighbors, denoted as users 4 and 10 with similarity coefficients of $-1$ each, are identified for user 1. Upon referring to the training set, we discover that user 2 rated movie B positively, while user 5 rated it negatively. Consequently,

$$\hat{r}_{1E} = (\bar{r} + b_1 + b_E) + \frac{d_{14}\tilde{r}_{4E} + d_{10}\tilde{r}_{10E}}{|d_{14}| + |d_{10}|} \tag{C.7}$$

$$= 4.62 + \frac{(-1 \times -0.22) + (-1 \times -0.22)}{1+1} = 0.22 + 4.62 = 4.84.$$

The predictions generated by the $k$-NN$^{\bigstar}_{\text{USERS}}$ method (after clipping) are depicted in Figure C.8.

$$
\hat{\mathbf{R}} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}
\begin{array}{c}
\begin{array}{ccccc} \;A\; & \;B\; & \;C\; & \;D\; & \;E\; \end{array} \\
\begin{bmatrix}
5 & 3.57 & 3.24 & - & \mathbf{4.84} \\
- & 2.26 & 4.34 & \mathbf{3.97} & 4.23 \\
4.28 & 2.45 & - & \mathbf{2.92} & 2.25 \\
- & \mathbf{1.00} & 2.71 & 1.29 & 1.71 \\
4.46 & - & \mathbf{4.30} & 4.49 & 5.00 \\
\mathbf{4.97} & 3.52 & - & 3.52 & 4.48 \\
2.97 & \mathbf{1.16} & 3.03 & 1.97 & - \\
4.28 & \mathbf{3.64} & 4.16 & - & 4.77 \\
\mathbf{4.25} & 2.44 & 5.00 & 4.33 & - \\
\mathbf{4.87} & - & 4.71 & 3.29 & 3.71
\end{bmatrix}
\end{array}
$$

FIGURE C.8 Rating matrix estimated by $k$-NN$^{\bigstar}_{\text{USERS}}$.

We compute the **rmse** between $\mathbf{R}$ and $\hat{\mathbf{R}}$ using Equation (2.12), resulting in values of 0.4386 for the training set and 0.5932 for the test set.

## C.4 Precision, Recall and MAP

We detail the step-by-step procedure for calculating **MAP@K**. We begin by calculating **Precision@K** and **Recall@K** for a user:

- Set a classification threshold $k$.
- Calculate the % relevant in the top-$k$.
- Ignore items classified lower than the threshold.

To begin with, we will ignore all ratings "**?**" where the true value is unknown. Values without a known true rating cannot be used. We set the threshold at 3.5 stars.

| Movie | Real rating | Predicted rating |
|---|---|---|
| **1** | 4 | 2.3 |
| **2** | 2 | 3.6 |
| **3** | 3 | 3.4 |
| **4** | **?** | 4.3 |
| **5** | 5 | 4.5 |
| **6** | **?** | 2.3 |
| **7** | 2 | 4.9 |
| **8** | **?** | 4.3 |
| **9** | **?** | 3.3 |
| **10** | 4 | 4.3 |

Table C.1: Toy example.

The relevant movies are already known in the dataset:

- Relevant movie: has a real rating $\geq 3.5$
- Irrelevant movie: has a real rating $< 3.5$

The **recommended** movies are generated by a recommendation algorithm.

- Recommended movie: has a predicted rating $\geq 3.5$
- Not recommended movie: has a predicted rating $< 3.5$

| Movie | Real rating | Predicted rating |
|---|---|---|
| **7** | 2 | 4.9 |
| **5** | 5 | 4.5 |
| **10** | 4 | 4.3 |
| **2** | 2 | 3.6 |
| **3** | 3 | 3.4 |
| **1** | 4 | 2.3 |

TABLE C.2: Sorting the list $\mathcal{B}_u(k)$ in descending order.

We classify the rest of the movies based on their predicted rating in descending order as shown in Table C.2.

As mentioned in Section 3.5.2, **Precision@k** is the proportion of recommended movies in the top-$k$ set that are relevant, while **Recall@k** is the proportion of relevant movies found in the top-$k$ recommendations. Therefore, we have:

$$\textbf{Precision@k} = \frac{\text{Number of relevant movies recommended@k}}{\text{Number of movies recommended@k}}$$

$$\textbf{Recall@k} = \frac{\text{Number of relevant movies recommended@k}}{\text{Total number of relevant movies}}$$

**Relevant movies**

- Relevant movies are those with a real rating $\geq 3.5$.
- Relevant movies: Movie 5, Movie 10, and Movie 1.
- Total number of relevant movies = 3.

**Recommended movies@2**

- The recommended movies in 2 are Movie 7 and Movie 5.
- Number of recommended movies = 2.

**Recommended and relevant movies @2**

- It is the intersection ($|\mathcal{B}u(k) \cap \mathcal{S}u|$) between Recommended@2 and Relevant@2.

- Recommended@2 intersection Relevant: Movie 5.
- Number of recommended movies that are relevant = 1

As mentioned in Section 3.5.2, **Precision@k** is the proportion of recommended movies in the top-$k$ set that are relevant, while **Recall@k** is the proportion of relevant movies found in the top-$k$ recommendations. Therefore, we have:

$$\textbf{Precision@k} = \frac{\text{Number of relevant movies recommended@k}}{\text{Number of movies recommended@k}}$$

$$\textbf{Recall@k} = \frac{\text{Number of relevant movies recommended@k}}{\text{Total number of relevant movies}}$$

**Relevant movies**

- Relevant movies are those with a real rating $\geq 3.5$.
- Relevant movies: Movie 5, Movie 10, and Movie 1.
- Total number of relevant movies = 3.

**Recommended movies@2**

- The recommended movies in 2 are Movie 7 and Movie 5.
- Number of recommended movies = 2.

**Recommended and relevant movies @2**

- It is the intersection ($|\mathcal{B}u(k) \cap \mathcal{S}u|$) between Recommended@2 and Relevant@2.
- Recommended@2 intersection Relevant: Movie 5.
- Number of recommended movies that are relevant = 1.

**Recommended movies@3**

- The recommended movies in 3 are Movie 7, Movie 5, and Movie 10.

- Number of recommended movies = 3.

**Recommended and relevant movies @3**

- It is the intersection ($|\mathcal{B}u(k) \cap \mathcal{S}u|$) between Recommended@3 and Relevant@3.
- Recommended@3 intersection Relevant: Movie 5 and Movie 10.
- Number of recommended movies that are relevant = 2.

$$\textbf{Precision@2} = \frac{\text{Number of relevant movies recommended@2}}{\text{Number of recommended movies@2}} = \frac{1}{2} = 50$$

$$\textbf{Recall@2} = \frac{\text{Number of relevant movies recommended@2}}{\text{Total number of relevant movies}} = \frac{1}{3} = 33.33$$

$$\textbf{Precision@3} = \frac{\text{Number of relevant movies recommended@3}}{\text{Number of recommended movies@3}} = \frac{2}{3} = 66.67$$

We can observe that the precision is 66.67%. Here, we can interpret that only 66.67% of the recommendations are truly relevant. The following equation calculates the recall@3, which is the percentage of relevant movies recommended in the top three list of recommended movies:

$$\textbf{Recall@3} = \frac{\text{Number of relevant movies recommended@3}}{\text{Total number of relevant movies}} = \frac{2}{3} = 66.67$$

This means that 66.67% of the relevant movies were recommended in the top three list. Using the results obtained earlier, we can calculate the average precision@3 using the following equation:

$$\textbf{AP@k} = \frac{1}{\text{relevant in k}} \sum_{1}^{k} \text{Precision@k} \cdot rel(i) \tag{C.8}$$

We can calculate the precision@3 for each relevant movie, and then average them to obtain the average precision@3:

$$\textbf{AP@3} = \frac{1}{3} * (\text{Precision@1} + \text{Precision@2} + \text{Precision@3}) \tag{C.9}$$

The precision@1, precision@2, and precision@3 are calculated as follows:

$$\mathbf{AP@1} = \frac{1}{3} * (\frac{1}{1}) = 0.33 \tag{C.10}$$

$$\mathbf{AP@2} = \frac{1}{3} * (\frac{1}{1} + \frac{1}{2}) = 0.50 \tag{C.11}$$

$$\mathbf{AP@3} = \frac{1}{3} * (\frac{1}{1} + \frac{1}{2} + \frac{2}{3}) = 0.72 \tag{C.12}$$

Finally, we can calculate the mean average precision@3 (MAP@3) as the average of the average precision@1, average precision@2, and average precision@3:

$$\mathbf{MAP@3} = \frac{1}{3} * (\mathbf{AP@1} + \mathbf{AP@2} + \mathbf{AP@3}) = 1.55 \tag{C.13}$$

## C.5 Simon Funk's SVD algorithm

A real-life dataset for a recommendation system is often very sparse, resulting in rating matrices **R** with a sparsity of over 95%. This is where Simon Funk's technique comes in. $\text{Funk}_{\text{SVD}}$ ignores these missing values and finds a way to calculate latent factors using only the known rating values. To achieve this matrix factorization approach with $\text{Funk}_{\text{SVD}}$, we follow these steps:

1. We construct two matrices, **U** and $\mathbf{V}^\top$, a matrix of users by the number of chosen latent factors and a matrix of these same latent factors by movies, respectively. We then fill these matrices with random numbers. At this point, we have three matrices: **R**, **U** and $\mathbf{V}^\top$.

| | Movie 1 | Movie 2 | Movie 3 | Movie 4 |
|---|---|---|---|---|
| User 1 | - | - | 9 | 1 |
| User 2 | 3 | - | 7 | - |
| User 3 | 5 | - | - | 10 |
| User 4 | - | 2 | - | - |

Table C.3: User-movie matrix **R**.

The matrix **U** (Users by latent factors filled with random values):

| | Factor 1 | Factor 2 | Factor 3 |
|---|---|---|---|
| User 1 | 0.8 | 1.2 | -0.2 |
| User 2 | 0.2 | 1.8 | 0.4 |
| User 3 | 0.8 | 3 | 0.1 |
| User 4 | 1 | 0.8 | 2.4 |

Table C.4: Matrix of latent factors **U**

The matrix $\mathbf{V}^\top$ (Movies by latent factors filled with random values):

| | Movie 1 | Movie 2 | Movie 3 | Movie 4 |
|---|---|---|---|---|
| Factor 1 | -1.8 | 1 | -0.2 | 1 |
| Factor 2 | 0.5 | 1.2 | 0.1 | 5 |
| Factor 3 | 1.4 | 4 | 0.14 | 2 |

Table C.5: Matrix of latent factors $\mathbf{V}^\top$.

2. We search for an existing rating in matrix **R** for a user-movie pair. The first rating we find in the matrix is 9, given by user 1 to movie 3. Therefore, 9 is our true value.

3. In matrix $\mathbf{U}$, we take all the random values associated with user 1 (row). For this user, we have [**0.8,1.2,-0.2**]. Recall that 9 was the rating that user 1 assigned to movie 3. In matrix $\mathbf{V}^\top$, we also take all the random values associated with movie 3. For this movie, we have $[\mathbf{-0.2,0.1,\ 0.14}]^\top$.

   We then compute the dot product between the found row and column in order to make the prediction: $(0.8 \times -0.2) + (1.2 \times 0.1) + (-0.2 \times 0.14) = -0.07$. Thus, we have the real value $r_{13}$ as 9 and the predicted value $\hat{r}_{13}$ as $-0.07$. Therefore, the error is $e = (9 + 0.07)^2 = 82.26$. Next, we minimize the error using SGD. The formula is:

$$\mathbf{U}_i + \gamma \cdot 2(\text{real} - \text{predicted}) \times \mathbf{V}_i$$

$$\mathbf{V}_i + \gamma \cdot 2(\text{real} - \text{predicted}) \times \mathbf{U}_i$$

   Here, $\mathbf{U}_i$ is a random value from the matrix $\mathbf{U}$, $\mathbf{V}_i$ is a random value associated with the transpose matrix $\mathbf{V}^\top$, and $\gamma$ is the learning rate. In summary, these are the values taken from the matrices $\mathbf{U}$ and $\mathbf{V}$.

| | Factor 1 | Factor 2 | Factor 3 |
|---|---|---|---|
| From **U** | 0.8 | 1.2 | -0.2 |
| From **V** | -0.2 | 0.1 | 0.14 |

   We update 0.8 using SGD with a learning rate of 0.1. Therefore, the new value is:

$$\text{NewValue} = 0.8 + 0.1 \times 2(9 + 0.07) \times -0.2 = 0.44$$

$$\text{NewValue} = 1.2 + 0.1 \times 2(9 + 0.07) \times 0.1 = 1.38$$

$$\text{NewValue} = -0.2 + 0.1 \times 2(9 + 0.07) \times 0.14 = 0.053$$

   By updating all values of $U$, we obtain $0.44, 1.38$ and $0.053$. Therefore, we now have:

| | Factor 1 | Factor 2 | Factor 3 |
|---|---|---|---|
| From **U** | 0.44 | 1.38 | 0.053 |
| From **V** | -0.2 | 0.1 | 0.14 |

Then we can update the values of $\mathbf{V}$. Note that the newly updated values of $\mathbf{U}$ are already affecting the values we will calculate from $\mathbf{V}$. Finally, we obtain:

| | Factor 1 | Factor 2 | Factor 3 |
|---|---|---|---|
| From **U** | 0.44 | 1.38 | 0.053 |
| From **V** | 0.6 | 2.60 | 0.24 |

We replace updated values in matrices $\mathbf{U}$ and $\mathbf{V}^\top$

| | Factor 1 | Factor 2 | Factor 3 |
|---|---|---|---|
| User 1 | **0.44** | **1.38** | **0.053** |
| User 2 | 0.2 | 1.8 | 0.4 |
| User 3 | 0.8 | 3 | 0.1 |
| User 4 | 1 | 0.8 | 2.4 |

| | Movie 1 | Movie 2 | Movie 3 | Movie 4 |
|---|---|---|---|---|
| Factor 1 | -1.8 | 1 | **0.6** | 1 |
| Factor 2 | 0.5 | 1.2 | **2.60** | 5 |
| Factor 3 | 1.4 | 4 | **0.24** | 2 |

We have now completed one iteration, meaning that $n = 1$.

## C.6 Ranking algorithm

To illustrate how Algorithm 20 works, let us consider a small numerical example.

**Algorithm 20:** Ranking task

**Input:** Set of users $U$, set of items $I$, function $f(u, i)$ which predicts the utility score of item $i$ for user $u$, and the number of recommended items $k$.

**Output:** Top-$k$ recommendation list for each user.

```
foreach user $u \in U$ do
    foreach not-interacted item $i \in I \setminus I_u^+$ do
        $n \leftarrow 0$ // Reset the counter for each item
        $\hat{r}_{ui} \leftarrow f(u, i)$ // Compute the utility score of item i for user u
        foreach not-interacted item $j \in I \setminus I_u^+$ do
            $\hat{r}_{uj} \leftarrow f(u, j)$ // Compute the utility score of item j for user u
            if $\hat{r}_{uj} \geq \hat{r}_{ui}$ then
                $n \leftarrow n + 1$ // Increment counter
            end if
        end foreach
        $\mathcal{O}_{ui} \leftarrow n + 1$ // Assign rank to item i
    end foreach
    Sort the items in $I \setminus I_u^+$ in descending order of their rank $\mathcal{O}_{ui}$
    Select the top-$k$ items from the sorted list to recommend to user $u$
end foreach
```

The inputs for the algorithm are as follows:

- The set of users is denoted as $U = \{1, 2, 3, 4\}$.
- The set of items is denoted as $I = \{a, b, c, d\}$.
- The function $f(u, i)$ predicts the utility score of item $i$ for user $u$.
- The number of recommended items is $k = 2$.

Let us assume the utility scores predicted by the function $f(u, i)$ for each user-item pair can be represented as a matrix:

$$\mathbf{R} = \begin{bmatrix} f(1,a) & f(1,b) & f(1,c) & f(1,d) \\ f(2,a) & f(2,b) & f(2,c) & f(2,d) \\ f(3,a) & f(3,b) & f(3,c) & f(3,d) \\ f(4,a) & f(4,b) & f(4,c) & f(4,d) \end{bmatrix} = \begin{bmatrix} 0.6 & 0.8 & 0.3 & 0.7 \\ 0.5 & 0.4 & 0.9 & 0.2 \\ 0.7 & 0.6 & 0.4 & 0.8 \\ 0.3 & 0.2 & 0.5 & 0.9 \end{bmatrix}$$

Each row represents a user, while each column represents an item. The values within the matrix indicate the utility scores predicted by the function $f(u, i)$. To gain a deeper understanding of the algorithm, we will now proceed to analyze it step by step, referring to Algorithms 21 and 22 for users 1 and 2, respectively.

Upon computing the utility scores $\hat{r}_{ui}$ and ranks $\mathcal{O}_{ui}$ for all items, encompassing $a$, $b$, $c$ and $d$, for both User 1 and User 2, we can proceed to summarize the outcomes of the algorithm. The resulting summary is presented in Table C.6, showcasing the ranks ($\mathcal{O}_{ui}$) assigned to each item and offering a comprehensive overview of the ranking assignments for both users.

| **User 1** | | **User 2** | |
|---|---|---|---|
| Item | Rank ($\mathcal{O}_{ui}$) | Item | Rank ($\mathcal{O}_{ui}$) |
| $a$ | $\mathcal{O}_{1a} = 3$ | $a$ | $\mathcal{O}_{2a} = 2$ |
| $b$ | $\mathcal{O}_{1b} = 1$ | $b$ | $\mathcal{O}_{2b} = 3$ |
| $c$ | $\mathcal{O}_{1c} = 4$ | $c$ | $\mathcal{O}_{2c} = 1$ |
| $d$ | $\mathcal{O}_{1d} = 2$ | $d$ | $\mathcal{O}_{2d} = 4$ |

Table C.6: Rankings of items for users 1 and 2.

Based on the ranks $\mathcal{O}_{ui}$, we sort the items in $I \setminus I_u^+$ (not-interacted items) in ascending order for each user $u$. In this case, the sorted order for User 1 will be $b$, $d$, $a$ and $c$. For User 2, it will be $c$, $a$, $b$ and $d$. Finally, we select the top-$k$ items from the sorted list to recommend to each user. Since $k = 2$, the top-2 items for User 1 will be $b$ and $d$. The top-2 items for User 2 will be $c$ and $a$. Therefore, the top-2 recommendation list for User 1 will be $b$ and $d$. The top-2 recommendation list for User 2 will be $c$ and $a$.

**Algorithm 21:** Ranking task for User 1

```
for user $u = 1$ do
    for item $i = a$ do
        Compute $\hat{r}_{ui} = f(1, a) = 0.6$
        Compute $\hat{r}_{uj}$ for all not-interacted items $j$:
        for each not-interacted item $j$ do
            $\hat{r}_{uj}$: $f(1, b) = 0.8$, $f(1, c) = 0.3$, $f(1, d) = 0.7$
            Increment $n$ because $\hat{r}_{uj} \geq \hat{r}_{ui}$ for items $b$ and $d$
        end for
        Assign rank $\mathcal{O}_{ui} = n + 1 = 3$ for item $a$
    end for
    $n \leftarrow 0$
    for item $i = b$ do
        Compute $\hat{r}_{ui} = f(1, b) = 0.8$
        Compute $\hat{r}_{uj}$ for all not-interacted items $j$:
        for each not-interacted item $j$ do
            $\hat{r}_{uj}$: $f(1, a) = 0.6$, $f(1, c) = 0.3$, $f(1, d) = 0.7$
            No increment of $n$ because $\hat{r}_{uj} < \hat{r}_{ui}$ for all items
        end for
        Assign rank $\mathcal{O}_{ui} = n + 1 = 1$ for item $b$
    end for
    $n \leftarrow 0$
    for item $i = c$ do
        Compute $\hat{r}_{ui} = f(1, c) = 0.3$
        Compute $\hat{r}_{uj}$ for all not-interacted items $j$:
        for each not-interacted item $j$ do
            $\hat{r}_{uj}$: $f(1, a) = 0.6$, $f(1, b) = 0.8$, $f(1, d) = 0.7$
            Increment $n$ because $\hat{r}_{uj} > \hat{r}_{ui}$ for items $a$, $b$, and $d$
        end for
        Assign rank $\mathcal{O}_{ui} = n + 1 = 4$ for item $c$
    end for
    $n \leftarrow 0$
    for item $i = d$ do
        Compute $\hat{r}_{ui} = f(1, d) = 0.7$
        Compute $\hat{r}_{uj}$ for all not-interacted items $j$:
        for each not-interacted item $j$ do
            $\hat{r}_{uj}$: $f(1, a) = 0.6$, $f(1, b) = 0.8$, $f(1, c) = 0.3$
            Increment $n$ because $\hat{r}_{uj} \geq \hat{r}_{ui}$ for item $b$
        end for
        Assign rank $\mathcal{O}_{ui} = n + 1 = 2$ for item $d$
    end for
end for
```

**Algorithm 22:** Ranking task for User 2

```
for item $i = a$ do
    Compute $\hat{r}_{ui} = f(2, a) = 0.5$
    Compute $\hat{r}_{uj}$ for all not-interacted items $j$:
    for each not-interacted item $j$ do
        $\hat{r}_{uj}$: $f(2, b) = 0.4$, $f(2, c) = 0.9$, $f(2, d) = 0.2$
        Increment $n$ because $\hat{r}_{uj} \geq \hat{r}_{ui}$ for item $c$
    end for
    Assign rank $\mathcal{O}_{ui} = n + 1 = 2$ for item $a$
end for
$n \leftarrow 0$
for item $i = b$ do
    Compute $\hat{r}_{ui} = f(2, b) = 0.4$
    Compute $\hat{r}_{uj}$ for all not-interacted items $j$:
    for each not-interacted item $j$ do
        $\hat{r}_{uj}$: $f(2, a) = 0.5$, $f(2, c) = 0.9$, $f(2, d) = 0.2$
        Increment $n$ because $\hat{r}_{uj} \geq \hat{r}_{ui}$ for items $c$ and $a$
    end for
    Assign rank $\mathcal{O}_{ui} = n + 1 = 3$ for item $b$
end for
$n \leftarrow 0$
for item $i = c$ do
    Compute $\hat{r}_{ui} = f(2, c) = 0.9$
    Compute $\hat{r}_{uj}$ for all not-interacted items $j$:
    for each not-interacted item $j$ do
        $\hat{r}_{uj}$: $f(2, a) = 0.5$, $f(2, b) = 0.4$, $f(2, d) = 0.2$
        No increment of $n$ because $\hat{r}_{uj} < \hat{r}_{ui}$ for all items
    end for
    Assign rank $\mathcal{O}_{ui} = n + 1 = 1$ for item $c$
end for
$n \leftarrow 0$
for item $i = d$ do
    Compute $\hat{r}_{ui} = f(2, d) = 0.2$
    Compute $\hat{r}_{uj}$ for all not-interacted items $j$:
    for each not-interacted item $j$ do
        $\hat{r}_{uj}$: $f(2, a) = 0.5$, $f(2, b) = 0.4$, $f(2, c) = 0.9$
        Increment $n$ because $\hat{r}_{uj} \geq \hat{r}_{ui}$ for all items
    end for
    Assign rank $\mathcal{O}_{ui} = n + 1 = 4$ for item $d$
end for
```

# Appendix D

# Model Selection Illustration

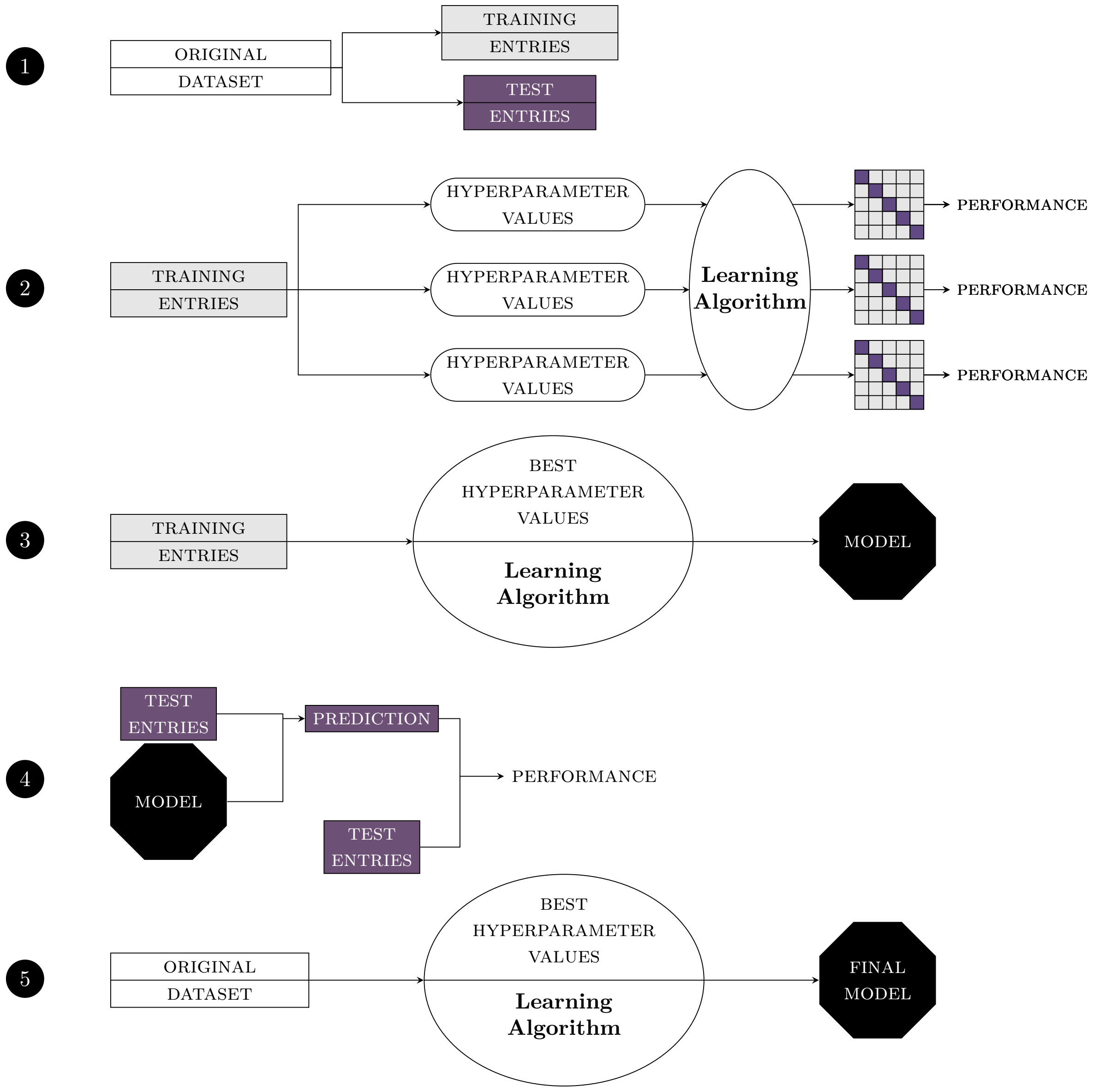


FIGURE D.1: $k$-fold cross-validation for model selection. (Reference:[Ras20])

# Bibliography

[AC16] Deepak K. Agarwal and Bee-Chung Chen. *Statistical Methods for Recommender Systems.* Cambridge University Press, USA, 1st edition, 2016.

[AC19] Francesco Archetti and Antonio Candelieri. *Bayesian Optimization and Data Science.* Springer Publishing Company, Incorporated, 1st edition, 2019.

[Agg16] Charu C. Aggarwal. *Recommender Systems: The Textbook.* Springer Publishing Company, Incorporated, 1st edition, 2016.

[And04] Chris Anderson. The long tail. *Wired Magazine*, 2004.

[AT05] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, June 2005.

[BB08] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, pages 161–168, 2008.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[BCN18] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. 2018.

[Ber19] Hadrien Bertrand. *Hyper-parameter optimization in deep learning and transfer learning : applications to medical imaging.* PhD dissertation, Université Paris-Saclay, NNT : 2019SACLT001ff, 2019.

[BHK98] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the*

*Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, page 43–52, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[BHMM19] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[Bjo19] Emil Bjornson. Reproducible research: Best practices and potential misuse [perspectives]. *IEEE Signal Processing Magazine*, 36(3):106–123, 2019.

[BK07a] Robert M. Bell and Yehuda Koren. Lessons from the netflix prize challenge. *SIGKDD Explor. Newsl.*, 9(2):75–79, December 2007.

[BK07b] Robert M. Bell and Yehuda Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 43–52, 2007.

[BOHG13] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, 2013.

[Cao16] Longbing Cao. Non-iid recommender systems: A review and framework of recommendation paradigm shifting. *ArXiv*, abs/2007.07217, 2016.

[Chi12] Mung Chiang. *Networked Life: 20 Questions and Answers.* Cambridge University Press, New York, NY, USA, 2012.

[CKT10] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. Performance of recommender algorithms on top-n recommendation tasks. pages 39–46, 01 2010.

[CP04] B. Jack Copeland and Diane Proudfoot. *The Computer, Artificial Intelligence, and the Turing Test*, pages 317–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[DK11] Christian Desrosiers and George Karypis. *A Comprehensive Survey of Neighborhood-based Recommendation Methods*, pages 107–144. Springer US, Boston, MA, 2011.

[DOM02] Stephan Dreiseitl and Lucila Ohno-Machado. Logistic regression and artificial neural network classification models: A methodology review. *J. of Biomedical Informatics*, 35(5/6):352–359, oct 2002.

[Fis36] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(7):179–188, 1936.

[Fun06] Simon Funk. Netflix update:try this at home. `http://sifter.org/simon/journal/20061211.html`, 2006.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[GNOT92] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, December 1992.

[Got97] L. Gottfredson. Mainstream science on intelligence: An editorial with 52 signatories, history, and bibliography. *Intelligence*, 24:13–23, 1997.

[GSB$^+$16] María N. Moreno García, Saddys Segrera, Vivian F. López Batista, María Dolores Muñoz Vicente, and Angel L. Sánchez. Web mining based framework for solving usual problems in recommender systems. a case study for movies' recommendation. *Neurocomputing*, 176:72–80, 2016.

[HK15] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015.

[Hof04] Thomas Hofmann. Latent semantic models for collaborative filtering. *ACM Trans. Inf. Syst.*, 22(1):89–115, January 2004.

[JRTZ16] Dietmar Jannach, Paul Resnick, Alexander Tuzhilin, and Markus Zanker. Recommender systems — beyond matrix completion. *Commun. ACM*, 59(11):94–102, October 2016.

[JSW98] D. Jones, Matthias Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998.

[KAU16] Shah Khusro, Zafar Ali, and Irfan Ullah. *Recommender Systems: Issues, Challenges, and Research Opportunities*, pages 1179–1189. 02 2016.

[KB11] Yehuda Koren and Robert Bell. *Advances in Collaborative Filtering*, pages 145–186. Springer US, Boston, MA, 2011.

[KEK18] Daniel Kluver, Michael D. Ekstrand, and Joseph A. Konstan. *Rating-Based Collaborative Filtering: Algorithms and Evaluation*, pages 344–390. Springer International Publishing, Cham, 2018.

[Kor08] Yehuda Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, page 426–434, New York, NY, USA, 2008. Association for Computing Machinery.

[Kus64] H. J. Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86(1):97–106, mar 1964.

[KY05] Dohyun Kim and Bong-Jin Yum. Collaborative filtering based on iterative principal component analysis. *Expert Systems with Applications*, 28(4):823 – 830, 2005.

[LDS11] Pasquale Lops, Marco Degemmis, and Giovanni Semeraro. Content-based recommender systems: State of the art and trends. In *Recommender Systems Handbook*, 2011.

[LM05] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering, 2005.

[LT15] Aristomenis S. Lampropoulos and George A. Tsihrintzis. *Machine Learning Paradigms: Applications in Recommender Systems.* Springer Publishing Company, Incorporated, 2015.

[McC98] John McCarthy. What is artificial intelligence. 1998.

[McC07] John McCarthy. What is artificial intelligence?, 2007. http://www-formal.stanford.edu/jmc/.

[Mit97] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[MMRS06] John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Magazine*, 27(4):12–14, 2006.

[MRT12] M. Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. Foundations of machine learning. In *Adaptive computation and machine learning*, 2012.

[OD19] Graham Oppy and David Dowe. The Turing Test. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2019 edition, 2019.

[OMT$^+$08] Thomas Oommen, Debasmita Misra, Navin KC Twarakavi, Anupma Prakash, Bhaskar Sahoo, and Sukumar Bandopadhyay. An objective analysis of support vector machine based classification for remote sensing. *Mathematical geosciences*, 40(4):409–424, 2008.

[Pat07] Arkadiusz Paterek. Improving regularized singular value decomposition for collaborative filtering. *Proceedings of KDD Cup and Workshop*, 01 2007.

[PB07] Michael J. Pazzani and Daniel Billsus. Content-based recommendation systems. In *The Adaptive Web*, 2007.

[PVG$^+$12] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Edouard Duchesnay, and Gilles Louppe. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12, 01 2012.

[Ras20] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning, 2020.

[RIS$^+$94] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, pages 175–186, New York, NY, USA, 1994. ACM.

[RW05] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[SB18] Alan Said and Alejandro Bellogín. Coherence and inconsistencies in rating behavior: Estimating the magic barrier of recommender systems. *User Modeling and User-Adapted Interaction*, 28(2):97–125, jun 2018.

[Sea80] John R. Searle. Minds, brains, and programs. *Behavioral and Brain Sciences*, 3(3):417–424, 1980.

[SKKR00] Badrul Munir Sarwar, George Karypis, Joseph A. Konstan, and John Thomas Riedl. Application of dimensionality reduction in recommender system - a case study. 2000.

[SKKR01] Badrul Sarwar, George Karypis, Joseph A Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web, WWW 2001*, WWW '01, pages 285–295. Association for Computing Machinery, Inc, 4 2001.

[SKKS10] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 1015–1022, Madison, WI, USA, 2010. Omnipress.

[SSBD14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, USA, 2014.

[SSW+16] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, jan 2016.

[Tur50] A. M. Turing. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[UFA+98] Lyle Ungar, Dean Foster, Ellen Andre, Star Wars, Fred Star Wars, Dean Star Wars, and Jason Hiver Whispers. Clustering methods for collaborative filtering. AAAI Press, 1998.

[Vap00] Vladimir Vapnik. *The Nature of Statistical Learning Theory.* Springer: New York, 2000.

[Vel12] Rosemarie Velik. Ai reloaded: Objectives, potentials, and challenges of the novel field of brain-like artificial intelligence. *BRAIN Broad Research in Artificial Intelligence and Neuroscience*, 3:25–54, 10 2012.

[YWZ+16] Z. Yang, B. Wu, K. Zheng, X. Wang, and L. Lei. A survey of collaborative filtering-based recommender systems for mobile internet applications. *IEEE Access*, 4:3273–3287, 2016.