A Recommender System for e-Retail

University of Cape Town

A minor dissertation submitted by

Thomas Walwyn

and supervised by

Melvin Varughese

in partial satisfaction of the

requirements for the degree of

Master of Science

in

Advanced Analytics and Decision Sciences

in the

Department of Statistical Sciences

of the

University of Cape Town

June 2016

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

A Recommender System for e-Retail

Copyright 2016
by
Thomas Walwyn

Abstract

A Recommender System for e-Retail

by

Thomas Walwyn

supervised by

Melvin Varughese

Master of Science in Advanced Analytics and Decision Sciences

Department of Statistical Sciences

University of Cape Town

The e-retail sector in South Africa has a significant opportunity to capture a large portion of the country's retail industry. Central to seizing this opportunity is leveraging the advantages that the online setting affords. In particular, the e-retailer can offer an extremely large catalogue of products; far beyond what a traditional retailer is capable of supporting. However, as the catalogue grows, it becomes increasingly difficult for a customer to efficiently discover desirable products. As a consequence, it is important for the e-retailer to develop tools that automatically explore the catalogue for the customer. In this dissertation, we develop a recommender system (RS), whose purpose is to provide suggestions for products that are most likely of interest to a particular customer.

There are two primary contributions of this dissertation. First, we describe a set of six characteristics that all effective RS's should possess, namely; accuracy, responsiveness, durability, scalability, model management, and extensibility. Second, we develop an RS that is capable of serving recommendations in an actual e-retail environment. The design of the RS is an attempt to embody the characteristics mentioned above.

In addition, to show how the RS supports model selection, we present a proof-of-concept experiment comparing two popular methods for generating recommendations that we implement for this dissertation, namely, implicit matrix factorisation (IMF) and Bayesian personalised ranking (BPR).

Contents

Contents **i**

List of Figures **iv**

1 Introduction **1**
1.1 Research Objectives . . . 4
1.2 Research Approach and Chapter Overview . . . 4

2 Literature Review **5**
2.1 Retail in South Africa . . . 6
2.1.1 Retail Industry Definition . . . 6
2.1.2 Overview of the Retail Industry . . . 7
2.1.3 Top 3 Retail Companies . . . 9
2.2 Overview of e-Retail . . . 11
2.2.1 e-Retail Business Model . . . 12
2.2.2 Market Leaders in e-Retail . . . 13
2.2.3 State of e-Retail in SA . . . 14
2.2.4 Top 5 SA e-Retail Companies . . . 15
2.3 Advantages of e-Retail . . . 16
2.3.1 Greater Convenience . . . 17
2.3.2 Lower Prices . . . 18
2.3.3 Wider Selection . . . 18
2.4 Recommender Systems in e-Retail . . . 19
2.5 Data Sources . . . 23
2.6 Theoretical Framework . . . 25
2.7 Recommendation Techniques . . . 26
2.7.1 Content-Based Filtering . . . 26
2.7.2 Collaborative Filtering . . . 27
2.7.3 Hybrid Methods . . . 37
2.8 Evaluating Recommendations . . . 39
2.8.1 Offline Experiments . . . 40
2.8.2 Evaluation Metrics . . . 42
2.8.3 Online Experiments . . . 46
2.9 Software Architecture . . . 47
2.9.1 Oryx . . . 48
2.9.2 Velox . . . 51
2.9.3 TencentRec . . . 53

2.10 Summary 55

3 System Description **56**
3.1 System Input 58
3.2 System Output 58
3.2.1 Product Recommendations 59
3.2.2 Model Evaluation 59
3.3 Hadoop Distributed File System 59
3.4 Apache Spark 63
3.5 Luigi 69
3.6 Data Ingestion 71
3.6.1 Ingestion Pipeline 72
3.7 Data Preparation 73
3.7.1 Prefiltering and Data Augmentation 73
3.7.2 Data Splitting 74
3.8 Model Building 77
3.8.1 Implicit Matrix Factorisation 78
3.8.2 Bayesian Personalised Ranking 86
3.9 Model Evaluation and Testing 91
3.9.1 Model Evaluation 92
3.9.2 Evaluation Summaries 94
3.9.3 Cross-Validation Pipeline 96
3.9.4 Model Testing Pipeline 96
3.10 Model Publishing 98
3.10.1 Model Publishing Pipeline 98
3.10.2 Parameter Server 99
3.11 Model Serving 99
3.11.1 Redis 100
3.11.2 Model Importer 100
3.11.3 Model Server 101
3.12 Summary 102

4 Takealot Deployment and Discussion **104**
4.1 Deployment 104
4.1.1 Amazon Elastic Cloud Compute 105
4.1.2 Takealot Data Centre 107
4.1.3 Takealot Mobile Applications 107
4.2 Discussion 108
4.2.1 Accuracy 108
4.2.2 Low Latency 109
4.2.3 Scalability 110
4.2.4 Durability 110
4.2.5 Model Management 111
4.2.6 Extensibility 118
4.3 Summary 118

5 Conclusion **120**

5.1 Limitations 121
5.2 Future Research 122
5.3 Acknowledgements 123

References **124**

List of Figures

2.1 Breakdown of sales in 2015 by cluster (in Rm). 8
2.2 Breakdown of growth by cluster for the period 2011 to 2015 (source: Statistics South Africa (2015b)). 8
2.3 Recommendations on the Amazon home page. 20
2.4 Recommendations on Netflix iOS application home screen. 21
2.5 Logistic sigmoid loss function, $\sigma(x) = \frac{1}{1+\exp(-x)}$. Very positive values of x result in $\sigma(x)$ being closer to one, while very negative values of x result in $\sigma(x)$ being closer to zero. At $x = 0$, $\sigma(x)$ is 0.5. 34
2.6 Example ROC curve (source: Dernoncourt (2015)). 44
2.7 Oryx architecture (source: Owen (2015)). 49
2.8 Velox architecture (source (Crankshaw et al., 2014)). 52
2.9 TencentRec architecture (source (Huang et al., 2015)). 54

3.1 Flow of data through the recommender system. 56
3.2 HDFS architecture (source: Hadoop Development Team (2015)). 61
3.3 Spark runtime architecture (source Zaharia et al. (2012)). 64
3.4 Example Luigi pipeline. 70
3.5 Definition of an example Luigi task. 71
3.6 Luigi tasks making up the ingestion framework. 73
3.7 Luigi tasks making up the data preparation framework. 73
3.8 Model building tasks fit into the workflow after the data preparation framework. 77
3.9 Example transaction matrix. 81
3.10 Customer inlink block example. 81
3.11 Product outlink block example. 82
3.12 An example of how blocks could be distributed throughout a Spark cluster with three worker nodes. 82
3.13 Factor blocks are distributed alongside inlink and outlink blocks. 83
3.14 Shuffling product factors required by the orange customer inlink block. 84
3.15 Model evaluation and testing framework. 91
3.16 Cross-validation pipeline. 96
3.17 The parameter search space is converted into a summary task for each combination of parameters. 97
3.18 Model testing pipeline. 97
3.19 Model publishing framework. 98
3.20 Model publishing pipeline. 99
3.21 Model serving framework. 100

4.1 Physical layout of the machines across EC2 and the Takealot DC. 105

4.2 Recommendations on the Takealot mobile applications. 108
4.3 Histogram of median latency between 5 AM and 12 AM for the model server. . . 109
4.4 Number of purchases accounted for by each customer group in the training set. . 113
4.5 Number of purchases accounted for by each product group in the training set. . . 114
4.6 Boxplots for each of the parameter settings in table 4.4, ordered by average AUC(highest on the left, lowest on the right). The upper and lower whiskers represent the 98 th and second percentile of AUC_u, respectively. 115
4.7 Boxplots for each of the parameter settings in table 4.5, ordered by average AUC. 116
4.8 Boxplots comparing the distributions of AUC_u for IMF and BPR over the various customer purchase frequency groups. 117

Acknowledgements

There are four groups of people to whom I owe credit for this dissertation: those at UCT, those at Takealot, my funders, and those closest to me.

First, I must acknowledge my supervisor at UCT, Melvin Varughese. Melvin has continually been there to prod me in the correct research direction. His comments always seemed unlock my thought processes during times of "writers block".

My colleagues at Takealot, both in the machine learning team and at management level have had a significant impact on this dissertation. The academic bent of the machine learning team was evident from early on in our inception; initially with Carl Scheffler, Philip Sterne, and myself, and later adding Maien Hamed. Ten-minute "stand-up" sessions often turned into hours-long discussions on any subject, from statistics and machine learning, to software development practices. The team has been a source of knowledge and inspiration throughout the course of this project. The management at Takealot, in particular Willem van Biljon (whose brainchild led to the project) and Pieter Rautenbach, have been immensely supportive of the project. They have a deep understanding of what is required to complete a dissertation, and have accommodated me at all points of the process.

This project would not have been possible without the financial support of the Oppenheimer Memorial Trust, whose support has been unquestioning. The opinions expressed in this dissertation are my own, and are not necessarily to be attributed to OMT.

My family, Sheila, David and Benjamin have been intimately involved in this dissertation (perhaps more than they would like). David and I have spent many hours talking over the structure and content of the final document. His input has been invaluable. While not as involved in the actual document, Sheila has provided equally valuable emotional motivation, encouraging me continuously to "get it done". I would also like to thank my brother, Benjamin, who has over the last two years, provided me with endless tutoring during my transition to his field of study, Statistics.

Finally, I must thank my love and partner-in-crime, Toni Huddy. Toni has always been there to listen to my concerns, put up with my odd working schedule consisting of early mornings and weekends, and drag me out for runs and social gatherings during times when I was losing my sanity.

Chapter 1

Introduction

Over the last decade, online shopping, also known as e-retail, has expanded rapidly around the globe. The South African e-retail market is small by international standards, but has an opportunity to quickly capture a significant portion of an retail sector dominated by large incumbents. Key to seizing this opportunity is leveraging advantages which are out of reach for traditional retailers. In particular, the e-retailer is able to offer an extremely large catalogue of products, far beyond what a traditional retailer can support.

Broadly defined, the retail trade sector in South Africa (SA) is the composition of retail outlets that sell goods and products to the general public for household use (Statistics South Africa, 2015a). It is a significant contributor to the South African economy, and has grown consistently in recent years. A handful of major firms with similar operational strategies dominate the industry, including Shoprite, Pick n Pay and Massmart.

The e-retail industry is contained within the overall retail trade industry. It consists of two main groups of retailers. The first group is those retailers that primarily offer their products to customers in physical stores, but also offer a secondary catalogue of products over the internet. The second group consists of those retailers who offer their products exclusively online through a website, mobile application, or both. These companies are known as "pure-play" e-retailers (Chiles and Dau, 2005). In this dissertation, our focus is on the pure-play environment.

Pure-play e-retail companies generally operate with one of two business models, namely, a first-party marketplace or a third-party marketplace. In the first-party model,the e-retailer owns the inventory it sells to customers. This is similar to the traditional retail model, where retailers have purchasing agreements with a range of suppliers, who deliver stock to stores, and this stock is sold to customers through the store-front. In the third-party marketplace model, the e-retailer effectively sells their brand and fulfilment network to other enterprises as a service. Here, the e-retailer acts as a central online hub on which a wide range of businesses can sell their

merchandise. This enables businesses to tap huge aggregated traffic flow that the e-retailer has already built (Devitt et al., 2013). A few international e-retailers have captured the imagination when thinking about these models. Amazon in the United States (US) comes to mind as a top e-retailer that operates with a mix of the two models, while eBay from the US and Alibaba from China dominate the global third-party marketplace.

The South African (SA) e-retail industry is small by international standards, and as a result, there is significant opportunity for growth. This opportunity has not gone unnoticed; there are a growing number of e-retail companies vying for the attention of SA internet users, including, Takealot, Zando, Loot, Superbalist and Spree.

There are a number of opportunities provided by the e-retail environment which have lead to the success of e-retail globally. First, e-retail offers greater convenience. Online shopping is not constrained to a physical store, therefore, as long a customer has access to the internet, they can shop. Second, e-retailers can potentially offer lower prices; in 2012, Amazon offered 5 % to 13 % lower prices on average compared to the top-five brick-and-mortar retailers in the US. Finally, e-retailers can offer a much wider selection of products. Since the e-retail catalogue is virtual, a product does not have to be in stock to be on display. With the addition of a third-party marketplace, the online catalogue could potentially be extended to tens, or hundreds of millions of products (Aufreiter et al., 2012).

Yet, with these opportunities come significant challenges for an e-retailer. For example, to realise the convenience of online shopping, an e-retailer must build accessible mobile platforms and efficient delivery infrastructure. Lower prices also come at a cost; in 2012, Amazon had invested three times more than offline competitors in technology research and development (R&D) to improve the efficiency of their supply chain. Even a large catalogue of products is no panacea for an e-retailer. As the catalogue grows, it becomes increasingly difficult for a customer to effectively discover desirable products. As a consequence, it is important to develop tools that intelligently explore the catalogue for the customer; surfacing relevant products based on observed behaviour.

In this dissertation, we develop a system that fulfils this requirement. Widely known as a recommender system (RS), our system is a set of "software tools and techniques that provide suggestions for items that are most likely of interest to a particular user" (Ricci et al., 2015). In the context of e-retail, items are generally products, and users are generally potential customers. Here, the primary role of a RS is to generate personalised rankings of products for customers. As a customer interacts with the e-retail platform, they are provided suggestions from this ranking. From the perspective of the e-retailer, the hope is that these suggestions meet the needs of the customer, generating a purchase. From the perspective of the customer, the recommendations should make their shopping experience more efficient by providing a personalised exploration of the product catalogue.

There are a number of steps involved in generating personalised recommendations. First, we must acquire the data necessary for building recommendations. This is important because it determines the recommendation techniques that should be supported by the RS. More specifically, some recommendation techniques require more "knowledge" than others. For example, some techniques need only know a unique identifier of a product with which a customer interacted, while others require detailed product descriptions and taxonomy information. Furthermore, some techniques require data that indicates customers' explicit preference for products, while others need only interaction data that implicitly indicates product preferences.

Next we need to actually generate recommendations. The literature on this subject has exploded in the last decade, due, in part, to the advent of the Netflix Prize (Barbieri et al., 2014). In 2006, Netflix, a leader in the film rental market promoted a competition with the goal of improving the prediction accuracy of their recommendation algorithm by 10 %. The competition lasted three years and involved research groups from across the globe, and inspired continuous and fruitful research. During this period, a huge number of recommendation approaches were proposed, substantially advancing the literature. Many of these approaches are in the subclass of recommendation techniques called collaborative filtering (CF), which tries to find similar customers, and recommends products they have bought. Earlier research on recommendation techniques focuses on content based filtering (CBF), which recommends to a customer products which contain similar attributes to products they have bought before. Since the Netflix prize, researchers have seen much success in combining CF and CBF using powerful hybrid recommendation techniques.

Finally, we need some way of evaluating recommendations. The adoption of a recommendation algorithm requires some evaluation of its ability to provide benefit to the customer or e-retailer. This is a challenging task, and according to (Barbieri et al., 2014) is one of "the biggest unsolved problems in RS." There are two main approaches for tackling this issue, namely, offline and online evaluation. In online evaluation, a small percentage of customers are directed to recommendations generated by the algorithm, and their interaction with the system is recorded. Their level of interaction is then compared to a baseline to determine the material impact of the recommendations. This gives a strong indication of the effectiveness of the recommendations, but carries significant risk; should system serve poor recommendations, there is a chance that customers become despondent and leave the platform without making a purchase. To mitigate this risk, an offline evaluation can be carried out beforehand to determine a candidate set of effective algorithms (Shani and Gunawardana, 2011). An offline evaluation involves evaluating the performance of the algorithm on a historical dataset. Typically, the procedure is to train the recommendation model on a portion of the dataset, then to check whether the model predicted interactions that occurred in the remainder of the dataset.

Thus, in building an RS, there are a series of diverse challenges about which the researcher

must reason. As a consequence, this research is situated at the intersection of multiple fields - statistics, machine learning, information retrieval, computer science and software engineering. The challenge is to pull together research in all these fields to build a system capable of serving effective recommendations, beyond the laboratory. This process is referred to as Data Science – converting large amounts of data into usable "data products" (Crankshaw et al., 2014).

1.1 Research Objectives

The focus of this dissertation has been to develop a prototype RS for use in an e-retail environment. In particular, the goals of this thesis are to;

1. develop set of characteristics with which we can evaluate the effectiveness of a RS, and,
2. using this framework, build a prototype RS for the South African e-retailer Takealot.

1.2 Research Approach and Chapter Overview

Including this chapter, the dissertation consists of five chapters. The purpose of chapter 2 is to perform an extensive review of the existing RS literature and extract the characteristics of a RS which generates effective recommendations, in the e-retail context. Toward that end, we first contextualise the role of a RS in e-retail, then develop our characteristics by stepping through the various aspects of a RS, including, data sources, recommendation techniques, and methods for evaluating recommendations. In chapter 3, we detail the design and implementation of the RS. We attempt to justify the design of each component by showing that it realises one of the characteristics which we have developed in chapter 2. Chapter 4 details the deployment of the RS, and evaluates the system against the characteristics we developed in chapter 2. Finally, in chapter 5, we conclude the research and provide a summary of possible future research directions.

Chapter 2

Literature Review

There are two main goals for this literature review. First, we aim to contextualise the role of a recommender system (RS) within the South African e-retailer, and second, we want to develop the characteristics of an effective RS.

In section 2.1 we start with an overview of the South African retail industry at large. We look at key segments of the industry, and examine some of the firms that dominate the market.

In section 2.2 we begin our exploration of the e-retail industry with a description of the various business models that have emerged in e-retail over the years, and give examples of the international firms who have used e-retail to gain a significant share of the broader retail market in countries like the United States (US) and China. Until now, this has been in stark contrast to the South African context, where e-retail has yet to make a significant impact. The remainder of section 2.2 introduces some of the key players in the South African e-retail industry and unpacks the issues that are preventing South Africans from converting to "e-shoppers". We also propose a series of potential solutions to these problems.

The success of e-retail abroad has largely been the result of realising customer conveniences which are not available to offline retailers. In section 2.3, we describe these conveniences and show how the South African e-retailers introduced in section 2.2 are working towards realising them as well. Customer convenience is only one side of the coin, though. Operationally, e-retailers have a much broader set of tools at their disposal than those available to brick-and-mortar retailers, largely as a result of the data-rich environment in which they operate. If South African e-retailers are to see the same success as the large American and Chinese firms, they must implement systems that take advantage of this environment as well. The remainder of section 2.3 is concerned with describing some of these systems.

In section 2.4 we begin to shift focus toward the second goal of this chapter, namely developing the characteristics of an effective RS. The primary role of a RS in e-retail is to provide

customers with a personalised exploration of an otherwise overwhelming catalogue of choices. However, it is not the only possible function. In this section, we explore the various roles of a RS in the e-retailer, which give rise to three important RS characteristics; (1) an RS should serve accurate recommendations, (2) requests for recommendations should be responded to within the window of interactivity, and (3) recommendations should always be available.

In section 2.5, we move on to the various sources of data that a RS might use to generate recommendations in the context of e-retail. This discussion gives rise to another of the key characteristics of an effective RS, namely, scalability in the variety and amount of input data.

Using these data sources, data scientists generate recommendations with a multitude of techniques. Thus, one of the important features of a RS is support for a few effective techniques and an extensible software interface with which new techniques can be implement new techniques. In section 2.6 we develop a framework for describing recommendation techniques, and in section 2.7 we survey the landscape of recommendation algorithms.

Once we have established a set of techniques for generating recommendations, how do we confirm that they are accurate? In section 2.8 we describe how a RS can effectively answer this question. A RS should have at its disposal a variety of metrics that can be used within an offline evaluation to ensure the recommendations remain effective as time passes.

Finally, in section 2.9, we present the architecture of three RS's, namely, Oryx, Velox, and TencentRec. We analyse to what extent these systems support our characteristics.

2.1 Retail in South Africa

Broadly defined, the retail trade sector is comprised of outlets that sell goods and products to the general public for household use (Statistics South Africa, 2015a). It is a significant contributor to the South African (SA) economy, and has grown consistently in recent years. A handful of major firms with similar operational strategies dominate the industry. In this section, we give a breakdown of the major segments of the retail industry and examine the structure of three of the largest retail companies, namely Shoprite, Pick n Pay and Massmart.

2.1.1 Retail Industry Definition

In South Africa, the retail industry forms a major part of the trade division, the components of which are: wholesale and retail trade, repair of motor vehicles and motor cycles, and hotels and restaurants (Provincial Treasury - Gauteng Province, 2012). The major source of information

about the retail industry for this dissertation has been the reports generated by Statistics South Africa[1] (Stats SA). They divide the statistics about the retail industry into seven clusters:

General Dealers Non-specialised trade across all segments, but predominantly in food.

Food Specialised trade in fresh fruit and vegetables, meat and meat products, bakery products, beverages, tobacco, and in other food in specialised stores.

Personal Care Specialised trade in pharmaceutical and medical goods, cosmetics and toiletries.

Fashion Specialised trade in textiles, clothing, footwear and leather goods.

Household Specialised trade in furniture, appliances and equipment.

Hardware Specialised trade in hardware, paint and glass.

Other Specialised trade in reading matter and stationery; jewellery, watches and clocks; sports goods; entertainment equipment; and second-hand goods. This includes repair of all personal and household goods, as well as all informal trade not in stores.

In the following section, we give an overview of the retail industry in terms of recent sales and historical growth. The overview includes a breakdown of sales and growth according the seven clusters above.

2.1.2 Overview of the Retail Industry

According to Stats SA, retail trade accounts for approximately 6 % of SA's Gross Domestic Product (GDP), with total sales of R868 billion in 2015 (Statistics South Africa, 2015b,a). By the end of 2014, the retail industry had more than 1.8 million employees. This represents approximately 20 % of the SA workforce[2] (Statistics South Africa, 2014). Thus, the retail industry is very important to the SA economy.

Figure 2.1 shows the breakdown of retail sales by cluster. With R351 billion in revenue, general dealers account for approximately 40 % of retail trade revenue. Next are fashion specialty stores with R177 billion in sales, which account for around 20 % of revenue. Together, general dealers and fashion retailers account for 60 % (over R520 billion) of sales.

Despite difficult broader economic circumstances, retail trade is one of the few industries in SA to have experienced persistent positive growth over the previous decade[3] (Statistics South

[1] http://www.statssa.gov.za/

[2] Not including the agricultural sector.

[3] Retail trade has grown every year since 2005, apart from 2009, in which all industries experienced the aftermath of the global financial crisis of 2008.

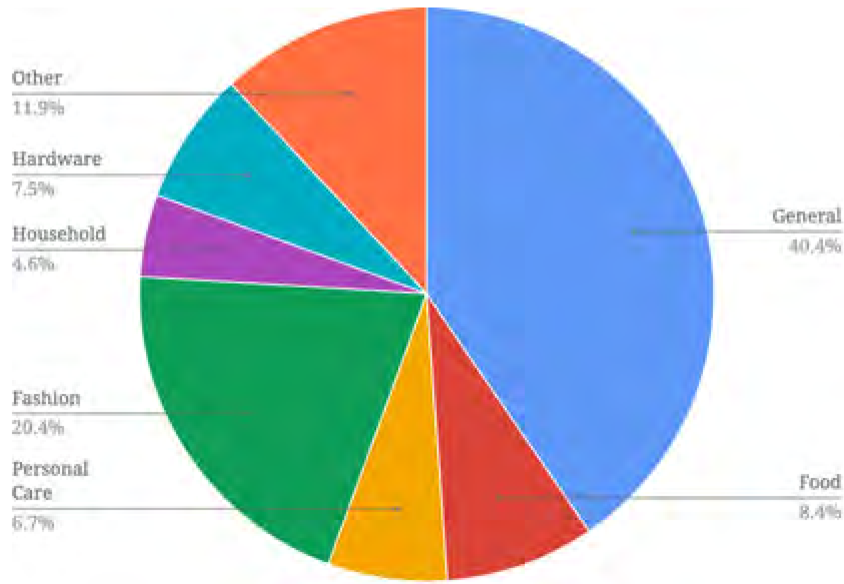


Figure 2.1: Breakdown of sales in 2015 by cluster (in Rm).

Africa, 2015b). Figure 2.2 gives a breakdown of the growth in retail sales over the period from 2011 to 2015. Total growth slowed from 2011 to 2013, from which point it picked up, reaching just above 7.5 % in 2015.

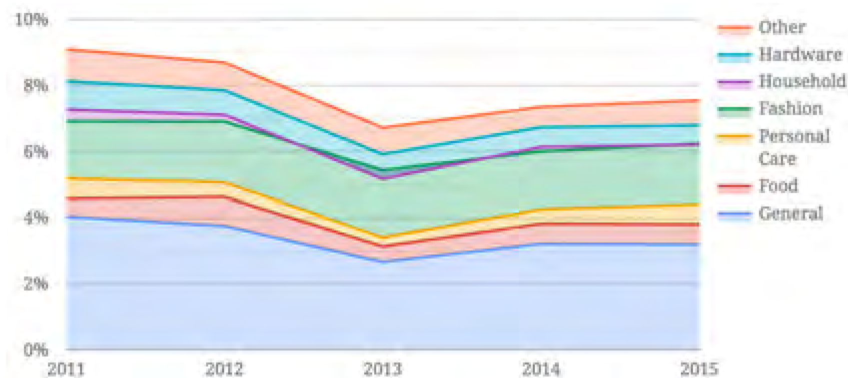


Figure 2.2: Breakdown of growth by cluster for the period 2011 to 2015 (source: Statistics South Africa (2015b)).

Overall growth in the retail industry can be attributed largely to general dealers and fashion retailers. General dealers, consistently account for the largest proportion of growth. Fashion retailers also account for a significant portion. Household specialty stores have struggled in this period; their revenue has shrunk in two of the five years, namely 2013 and 2015.

While general dealers account for the largest share of the retail market by revenue and growth, the segment itself is dominated by a few key firms. In the following section, we examine the structure of three of these firms.

2.1.3 Top 3 Retail Companies

The top three SA retail companies are determined by their revenue in 2015 according to their various financial results. These companies – Shoprite, Massmart and Pick n Pay – posted combined sales of R239 billion in 2015, approximately 28 % of the entire industry (Shoprite Holdings, Ltd., 2015; Massmart, 2015; Pick n Pay, 2015). Shoprite is the largest retailer in SA, with R88 billion in revenue. Massmart and Pick n Pay generated R84 billion and R67 billion, respectively.

Shoprite

Shoprite consists of three supermarket chains, namely, Shoprite, Usave and Checkers. Shoprite is the flagship brand of the group with 419 stores in SA as of 2015. Its core focus is to be the low-cost leader for all of its products, which include groceries and household items. Its target market is the middle income market of SA, and as a result it is concerned mainly with keeping low prices on basic commodities such as as maize meal, rice, sugar, oil, etc (Shoprite Holdings, Ltd., 2015).

Usave is a “no-frills” basic commodity discounter. It operates 272 stores in SA, mainly in rural areas of the country. Usave targets lower income consumers with discount prices on basic commodities (Shoprite Holdings, Ltd., 2015).

Checkers operates 191 stores in SA, focusing on higher income consumers. Checkers offers a wide range of groceries and household items, and tries to differentiate itself through specialty ranges of meats, cheeses, wines and coffee (Shoprite Holdings, Ltd., 2015).

Pick n Pay

Pick n Pay operate a similar set of stores to Shoprite. Its major supermarket chains are Pick n Pay and Boxer. As of 2015, there were 510 Pick n Pay supermarkets throughout South Africa. In contrast to Shoprite which offers different brands to middle and upper income markets, Pick n Pay tries to suit the needs of both market segments. Pick n Pay specialise in groceries and household items. As a result it competes directly with Shoprite and Checkers. In order to differentiate, they have introduced additional product lines in the form of basic apparel. In addition, Pick n Pay is a leader in hypermarkets, which are warehouse-like retail outlets that sell groceries, household items, and apparel under a single roof (Pick n Pay, 2015). The idea behind hypermarkets is that consumers benefit from a “one stop shopping” experience (Chiles and Dau, 2005).

Boxer stores are Pick n Pay's offer to lower income shoppers. Boxer operate 189 stores mainly in rural SA. The stores sell groceries, specialising in basic commodities, and general household items. It competes directly with Usave, and in order to differentiate, it offers liquor stores at many of their locations. In addition, Boxer has opened a number of discount hardware stores, called Boxer Build. These stores attempt to fill the need for a low cost building and hardware supplier (Pick n Pay, 2015).

Massmart

Compared to Pick n Pay and Shoprite, Massmart operates a more diverse portfolio of retailers. It consists of four divisions, each focused on high-volume, low-margin distribution of consumer goods. As of 2015, the divisions were made up of "Massdiscounters", "Masswarehouse", "Masscash" and "Massbuild" (Massmart, 2015).

Massdiscounters is comprised of Game and DionWired. Game operates 137 stores in the urban centres of SA. It is known for selling general household items aimed at the middle-income market. Recently, it has started selling groceries, which puts it in direct competition with Shoprite and Pick n Pay. DionWired is a much smaller operation with 24 stores across SA. It specialises in appliances and hi-tech electronics such as cameras, laptops, tablets and mobile phones. In 2015, these two chains made up just over R19.5 billion, or 23 % of Massmart's revenue (Massmart, 2015).

The Masswarehouse division of Massmart comprises primarily of Makro, the largest warehouse-club trading chain in South Africa. In this model, a small number of large warehouses serve as store-fronts for selling groceries, general household items, hardware and liquor, often in bulk at discount prices. Makro has 19 warehouses across SA. Although this number is small in comparison to the number of Game outlets, these warehouses accounted for over R23 billion, or 27.9 % of Massmart's revenue in 2015. The warehouses also act as distribution centres from which Makro operates a relatively small, but growing online business, which accounted for 2 % of revenue in those categories that are listed online (Massmart, 2015).

Masscash consists of a retail division and a wholesale division, each operating a number of brands. The Masscash brands target lower income markets. Its primary retail brand, Cambridge Foods, competes directly with Boxer and Usave, selling groceries and specialising in basic commodities. The retail division consists of 47 outlets throughout SA. The wholesale division consists of 75 stores under the brands CBW, Jumbo Cash and Carry, Trident, and Shield. These brands sell bulk groceries, liquor and personal care items to individuals, or other independent dealers at discount prices. They generally operate in the rural areas of SA. In 2015, the Masscash division accounted for R29.5 billion, or 35 % of Massmart's revenue. Although Masscash accounts for the largest proportion of revenue of the Massmart divisions, its proportion of profit

is the smallest due to their low margin structure (Massmart, 2015).

Massbuild consists of the Builders brand, which is further segmented into four complementary brands: Builders Warehouse, Builders Express, Builders Superstore, and Builders Trade Depot. Builders Warehouse is the equivalent of Makro for Do it Yourself (DIY) enthusiasts, or building and maintenance contractors. It operates 34 warehouses in major urban centres, offering a comprehensive range of home and garden improvement products, as well as building and maintenance supplies. Builders Express operates smaller home and garden improvement stores in higher income suburban environments. There are 36 Builders Express stores in SA. Builders Superstore is the equivalent offering for the rural market. It has been designed to complement the sites of Masscash with a home improvement offering. This puts it in direct competition with Boxer Build. Finally, Builders Trade Depot is a chain of 20 building contractor outlets in industrial sites in outlying urban areas. In 2015, the Builders brand accounted for R12 billion, or 15 % of Massmart's revenue (Massmart, 2015).

The traditional retailers in SA have a similar operating model. Apart from the warehouse-like outlets, retail chains have large footprints throughout the country, with tens or hundreds of physical stores. There are often different brands for higher, middle, and lower income markets. All three retailers focus on food and household items, with only Massmart branching out in a meaningful way. In the next section, we examine the operating models available to e-retailers and discuss international firms which exemplify these models.

2.2 Overview of e-Retail

There are two main models of operation followed by e-retailers, namely first-party models and third-party models. In addition, an e-retailer may choose to operate with a combination of the two models. In the first part of this section, we discuss these models in more detail. Next, we look at the strength of e-retail globally, which has been driven by three of the largest international e-retailers, namely Amazon and eBay from the US, and Alibaba from China. However, the situation in South Africa is somewhat different, the reasons for which we discuss in the fourth part of this section. Finally, we give an overview of the key players in the South African e-retail industry.

The e-retail industry is contained within the overall retail trade industry. It consists of two main groups of retailers. The first group is those retailers that primarily offer their products to customers in physical stores, but also offer a secondary catalogue of products over the internet. Makro is an example of an SA retailer who operates in this manner. These retailers are often referred to as "click-and-mortar" retailers. The second group consists of those retailers who offer their products exclusively online through a website, mobile application, or both. These

companies are known as “pure-play” e-retailers (Chiles and Dau, 2005).

2.2.1 e-Retail Business Model

e-Retail properties generally follow one of two business models; namely, first-party marketplace, and third-party marketplace (Dobbs et al., 2013). In the first-party model, the e-retailer owns the inventory it sells to customers. This is similar to the traditional retail model, where retailers have purchasing agreements with a range of suppliers, who deliver stock to stores, and this stock is sold to customers through the store-front. The difference in e-retail is that the products are listed on a virtual storefront, and customers use a website or mobile application to browse the catalogue. To complete the purchase, the customer provides credit card details or electronic fund transfer (EFT) information to the e-retailer during a virtual checkout process. The transaction value is then deducted from the customer’s account. Fulfilment is achieved through a number of channels. Primarily, e-retailers use delivery services to bring purchased products to the location of the customer. Alternatively, the e-retailer could make provisions for the customer to pick up the purchase. Typically, this would be from a centralised fulfilment centre, but could also be from a distributed network of pick up locations (Bensinger, 2012).

In the third-party marketplace model, the e-retailer effectively sells their brand and fulfilment network to other enterprises as a service. The e-retailer acts as a central online hub on which a wide range of businesses can sell their merchandise. This enables businesses to tap huge aggregated traffic flow that the e-retailer has already built. In addition, the e-retailer provides tools which businesses can use to launch quickly with minimal startup overhead. In particular, these merchants can use the existing storefront, payment collection, warehousing, and shipping infrastructure of the e-retailer (Devitt et al., 2013). The e-retailer then charges a fixed fee for each of these services.

Combining both models of business offers additional opportunities to the e-retailer. While it might seem counterintuitive for an e-retailer to allow third-party sellers to sell in direct competition with their own products, there are some important advantages. For example, increased competition leads to lower prices on those produces sold by multiple third-parties, which increases customer adoption of the platform as a whole. Increased sales from greater customer adoption outweighs the potential loss in profits due to lower prices (Devitt et al., 2013). In addition, a third-party marketplace significantly extends selection on the the e-retail platform. The marketplace backfills products that are inefficient to sell through an e-retailer’s first-party channels, as well as products that are temporarily out of stock. This increases “stickiness” of the platform, as customers are likely to find the products that they need at any time.

2.2.2 Market Leaders in e-Retail

Over the last decade e-retail has expanded rapidly in many countries and is expected to maintain this high growth rate for the foreseeable future (Devitt et al., 2013; Dobbs et al., 2013; Manyika et al., 2013). For instance, in the US, e-retail accounts for 7 % of all retail sales and is growing at about 15 % per annum (U.S. Census Bureau, 2015).

In Europe, e-retail accounts for approximately 7.2 % of the total retail market. It is also the fastest growing sector of the retail market, reaching about 18 % per annum, while traditional off-line sales have declined by 1.4 % (Centre for Retail Research, 2015).

In Asia, and particularly China, e-retailing has achieved astounding growth. Until 2011, the Chinese e-retail market had seen 120 % compound annual growth for nine years, reaching 5 % to 6 % of total retail sales in 2012 (Dobbs et al., 2013). South Korea has the highest level of e-retail penetration globally, accounting for approximately 15 % of retail trade in 2012 (Devitt et al., 2013).

Examples of global e-retail companies include Amazon and eBay from the US, and Alibaba from China. Amazon is the flagship example of an e-retailer operating with a combination of the first-party and third-party models. It sells from an expansive catalogue of products that includes books and media, electronics, household items, consumables, and apparel. Amazon has not traditionally not been in direct competition with supermarkets in the US as it has been unable to support the storage and delivery of fresh produce. However, it has recently developed a successful on-demand grocery service called Amazon Fresh, which it operates in a few cities around the US. With the addition of Amazon's third-party sellers, its catalogue is tens of millions of products in size, far beyond what could be maintained by a traditional retailer. Quality of service is guaranteed by providing third-parties with access to the advanced fulfilment infrastructure.

eBay is one of the largest third-party marketplace operators in the world. Its operations are primarily based in the US and Europe, but it also has a number of Asian operations. The business model is one where sellers pay a "seller fee" for the services provided by eBay, which include a virtual storefront and payment collection services (Devitt et al., 2013). Fulfilment services are the responsibility of the seller. In addition, seller registration is completely unfettered. These two factors mean that eBay has a challenging task in ensuring that buyers receive quality service. One effective mechanism is to promote sellers who consistently provide good service to buyers. Toward this end, eBay has developed a detailed seller ratings system. The system allows buyers to rate sellers on four aspects of their transactions, including; item description accuracy, level of communication, shipping time and shipping charges. If a seller manages to maintain a high seller rating, they are awarded certain advantages over other sellers, such as lower seller fees and a higher level of discovery on eBay's search engine.

Alibaba is the largest third-party marketplace operator in China. Two of its brands dominate the online shopping industry, namely; Taobao and Tmall (Dobbs et al., 2013). Like eBay, Taobao is a consumer-to-consumer e-retail platform that allows individual sellers to provide merchandise to buyers. One difference is that Taobao is built on an advertising revenue model and does not require sellers to pay fees to join the platform. Tmall takes a slightly different approach. It is exclusively a business-to-consumer platform that allows businesses to list catalogues of products on virtual storefronts.

2.2.3 State of e-Retail in SA

Compared to the global e-retail market, the scenario in SA is somewhat different. Consumers have been relatively slow to adopt e-retail. For example, in 2012, estimates by Manyika et al. (2013) assigned 0.5 % of retail trade to online commerce, compared to 5 % to 6 % in China. The reasons for this disparity appear to include both systemic and perception-based factors, with the critical systemic issue being the low rate of broadband internet penetration, and the critical perception-based issues being security and reliability.

The main systemic issue for e-retail in SA has been the low rate of broadband internet penetration. By 2013, only 3 % of the SA population had access to fixed-line broadband internet, compared to 14 % and 26 % for China and the US, respectively (Broadband Commision, 2014). The solution to this problem appears to lie with mobile internet access. While fixed-line internet penetration was extremely low in 2013, mobile internet access had proliferated much more widely, with 31 % of SA households accessing the internet through mobile devices (Statistics South Africa, 2013). This is not a trend limited to SA; for many in the developing world, a mobile device is their first and only internet-connected device (Fletcher and Crawford, 2013). Thus, in order to access the whole population, SA e-retailers need to provide comprehensive mobile experiences. Then, as more potential customers connect to the internet through their mobile devices, e-retailers will be ready to meet their needs.

The main perception-based issues are security and reliable service delivery. Market research has shown that South Africans perceive purchasing goods online to be unsecure (Manyika et al., 2013), and as a result they are reluctant to give credit card information to e-retailers. This is largely due to people's general mistrust for new technologies. There is no immediate solution to this problem, but the growth of e-retail globally shows that as people are increasingly exposed to the online environment, they overcome their fears. As a result, the strategy from SA e-retailers in this regard should be to promote awareness around security features and their relationships with reputable payment providers.

Concerns around reliable service delivery include doubts about the return path for unwanted products, high shipping costs (World Wide Worx, 2014), and extended delivery times (Devitt

et al., 2013). There are a number of strategies that SA e-retailers could use to ease these doubts. First, they could provide a simple and clear returns policy. Second, a range of shipping options could be provided, allowing a customer to make the trade off between cost and time that suits their situation. For example, the e-retailer could provide the option to pay more for immediate shipment. In addition, there could be a free shipping option for orders that meet certain requirements, such as exceeding some purchase value. Another possible strategy is to leave shipping out of the equation and allow customers to collect their orders for no charge (Devitt et al., 2013).

The challenges faced by the SA e-retail industry are not insurmountable. The global success of e-retail, especially in developing markets such as China, is testament to this. Some issues will take years to disappear, but when they do, SA e-retailers need to be ready to serve the needs of the entire population. In the following section, we introduce some key SA e-retail firms.

2.2.4 Top 5 SA e-Retail Companies

The top five SA pure-play e-retail companies are determined through an estimate of the number of visitors to their websites for the month of February 2016. Since these companies are relatively small, and often privately held, their financial results are not available to the public. An alternative heuristic measure of size is website traffic. The estimates summarised in table 2.1 have been obtained from SimilarWeb[4]. By this measure, Takealot, Zando, Loot, Superbalist and Spree are the top-five e-retailers in SA. Note that the estimates could change from month-to-month, thus in another month we could have a different set of companies, especially in the lower places where the difference between visits is small. However, the objective is to give some insight into the services and products available to SA e-shoppers, and we believe that the e-retailers presented in table 2.1 are a good representative sample.

Retailer	Estimated Visits
Takealot	6.7 M
Zando	1.2 M
Loot	934.8 K
Superbalist	684.5 K
Spree	680.2 K

Table 2.1: Estimated visits in February 2016 for the top-five e-retail websites in South Africa.

We start with Takealot, since it is the largest e-retailer according to our heuristic measure. Takealot has a similar operating model to Amazon. It sells from an extensive range of general merchandise sourced through agreements with various suppliers. In addition, Takealot operates

[4] https://www.similarweb.com/

a third-party marketplace, which allows sellers to make use of its fulfilment infrastructure for a fee. Its catalogue includes books, household items, consumables, electronics, and clothing.

Loot is the other general merchandiser in table 2.1. It is smaller than Takealot in terms of absolute visitor count, with just below one million visitors for February 2016. In contrast to Takealot, its business model is first-party only. As a result, Loot does not have the same category depth as Takealot, which could partly explain their significantly lower traffic.

The remaining e-retailers in table 2.1 are fashion merchandisers. Superbalist and Spree offer a magazine-styled shopping experience. This means that in addition to a traditional e-retail shopping experience, they offer “fashion stories” where content creators weave “shoppable” fashion journalism, style advice and curated collections in a magazine-like interface. This is a more targeted experience compared to the more general strategy of Takealot and Loot, hence their relatively smaller estimated visitor counts.

Zando offers a more traditional e-retail shopping experience for fashion, with a deep catalogue of women’s and men’s clothing at discount prices. As as result, Zando has been successful in appealing to a wide-range of South Africans, making it the second largest pure-play e-retailer in SA, according to our heuristic measure.

Not listed in table 2.1, but also popular according to the same estimate of website traffic are the SA “click-and-mortar” retailers. Of these, Makro is the largest with an estimated 1.7 million visitors in February 2016. This is in-line with results reported by Massmart (2015), where e-retail accounted for 2 % of sales in 2015 for categories that have been listed online.

While Takealot dominates in our heuristic measure, it is orders of magnitude smaller than the large international e-retailers. For example, SimilarWeb estimates that Amazon had 2.1 billion visits in February 2016. This staggering difference is a symptom of the barriers faced by SA e-retailers in driving adoption of online shopping.

2.3 Advantages of e-Retail

Global popularity of e-retail is a result of the three-fold advantage e-retail holds over offline avenues: first, consumers experience a greater level of convenience online than offline; second, e-retailers potentially offer lower prices to their customers; and third, e-retailers potentially offer a much broader catalogue of products than what is typically available in brick-and-mortar retailers. These advantages have been realised by successful international e-retailers through higher-than-average investment in technology research and development (R&D). In this section, we provide more detail about each of these advantages. We examine the efforts of SA e-retailers to achieve them, and explore how successful e-retailers abroad have achieved them.

2.3.1 Greater Convenience

e-Retail succeeds when the overall level of convenience becomes higher than offline channels (Devitt et al., 2013). Shopping in-store has benefits for the customer, including; the ability to physically interact with products; the instant availability of assistance should it be required, and the instant gratification received by taking ownership of a product immediately after purchase. In addition, the customer is well-aware of the return path for defective or unwanted products, should it come to that. On the other hand, the convenience of online shopping is driven largely by the fact that it is not constrained by a physical store – as long as a customer has access to the internet, they can shop. As discussed in section 2.2.3, to realise this advantage, the e-retailer should implement a number of measures, including a comprehensive mobile experience, excellent customer support, an accelerated and affordable delivery program, and a hassle-free return path.

SA firms are working to improve the convenience of shopping on their platforms by addressing issues around accessibility of their service, affordability of delivery, extended delivery times and returns, as the following factors indicate:

- Makro is the only e-retailer discussed in section 2.2.4 that does not offer a mobile experience. Takealot, Zando and Spree offer native Android and iOS applications, while Loot offers a mobile-friendly website.
- Makro is also the only e-retailer that does not offer some sort of free delivery option. The other e-retailers offer an option for free delivery on orders over a certain value. On average, these companies promise to deliver to customers in urban areas within three days, and customers in outlying areas within five days.
- Takealot, Zando and Superbalist offer an expedited delivery option, where for a fee, customers can request that packages be delivered on the same day of purchase. After hours and weekend delivery is also available for a fee.
- Takealot, Zando, Makro and Loot offer a "click-and-collect" option where a customer can collect their order from a centralised location with no charge. For example, in the case of Takealot, customers can collect their orders from the Takealot warehouse. In the case of Makro, collection can be arranged from the store most convenient for the customer. In addition, Makro has been testing pick up lockers, where an order is delivered to a secure locker from which the customer can retrieve their order at any time that is convenient for them.
- Takealot, Zando, Spree and Superbalist include in their returns policy a free pick up service. This means that unwanted merchandise is collected from a customer, instead of the customer having to return a product themselves.

Although the efforts of the e-retailers to make online shopping more accessible and convenient are necessary, they are not sufficient to ensure the success of the industry. In the following sections, we detail some of the additional measures international e-retailers have implemented to ensure success.

2.3.2 Lower Prices

By leveraging the data-rich environment in which an e-retailer operates, it can achieve a lower cost structure than offline peers. These cost savings can be passed on to customers in the form of lower prices. For example, in 2012, Amazon was able to offer 5 % to 13 % lower prices on average (including shipping) compared to the top-five brick-and-mortar retailers in the US (Aufreiter et al., 2012). This was the result of aggressive investment in technology R&D. Amazon had invested three times more in R&D compared to the other top retailers, all the while operating in an environment which allows detailed tracking of customer actions. This investment resulted in systems capable of leveraging customer behaviour data to automatically perform many of the tasks necessary to operate efficiently. For example, in supply chain management, customer browsing, searching and purchase behaviour can be used for accurate product demand estimation. This reduces the need to hold stock for extended periods, giving Amazon superior working capital management, and resulting in significant cost saving.

2.3.3 Wider Selection

An e-retailer truly becomes a “one-stop” shopping destination when it offers a range of products not available in any single offline store. In 2012, Amazon offered seven times more choice across product categories than offline competitors, made possible by two important features. First, a product does not have to be in stock to be on display in the online environment. Second, their third-party marketplace enabled Amazon to significantly extend the reach of their catalogue; by 2012, the marketplace accounted for 30 million products stocked by two million sellers (Devitt et al., 2013).

The advantage of a massive catalogue is potentially outweighed by the problem of product discovery. Without mechanisms to effectively explore the catalogue, a customer quickly becomes overwhelmed and leave the site. Broadly, products are discovered through two forms of queries; explicit queries, and implicit queries. In the explicit case, a customer arrives the desire to buy a specific product, or set of products. If the customer cannot find that product, they leave without making a purchase. In the implicit case, a customer comes to the site with the intention of browsing, without a specific product in mind. If the customer is presented with products that appeal to their tastes, they are likely to make a purchase. An e-retailer that effectively deals

with the problem of product discovery minimises the number of sales lost in the first case, and maximises the number of sales gained in the second case.

Two mechanisms are generally provided by e-retailers to tackle the problem of product discovery. In the explicit case, the customer is provided with a product search engine, which functions as follows; a customer inputs the name or description of a product they have in mind, they are then presented with a candidate list of products. In the ideal case, their desired product is near the top of the candidate list and the customer makes a purchase. This is effectively a problem of information retrieval (IR), which has a long history in literature (Barbieri et al., 2014). A newer research area has emerged to tackle the implicit case; that of the RS.

In the following section we introduce the role of the RS in e-retail.

2.4 Recommender Systems in e-Retail

The aim of this section is to highlight the prominence of RS's in e-retail, to justify the importance of RS's in e-retail, and finally to analyse the implications of this importance for the designer of a RS.

Ricci et al. (2015) define a RS to be a set of "software tools and techniques that provide suggestions for items that are most likely of interest to a particular user." In the context of e-retail, items are generally products, and users are generally potential customers.

There are many scenarios in which customers might be presented with a list of products generated by a RS. For example, when a customer arrives at the home screen of an e-retail site or mobile app, they might be presented with recommendations that relate to their tastes, which have been inferred by the system through their behaviour on the platform. Customers might also be sent recommendations via email to entice them back to the e-retail platform after an extended period of absence. In both cases, the assumption is that customers generate implicit queries through their behaviour on the platform, and it is the responsibility of the e-retailer to surface products relevant to these queries through the use of a RS.

RS's play an important role in many of the largest e-retail organisations around the globe. Figure 2.3 gives an example of recommendations shown to customers who are logged in to the Amazon home page. These recommendations have clearly been personalised to the tastes of the customer. The top strip of products reads "Inspired by your browsing history," i.e. personalised by the browsing behaviour of the customer. The bottom strip of recommendations encourages the customer to explore the catalogue with the statement "Additional items to explore".

Over the last decade, Amazon has invested heavily in its RS. In 2006, Amazon CEO, Jeff Bezos stated, "If I have 3 million customers on the Web, I should have 3 million stores on the

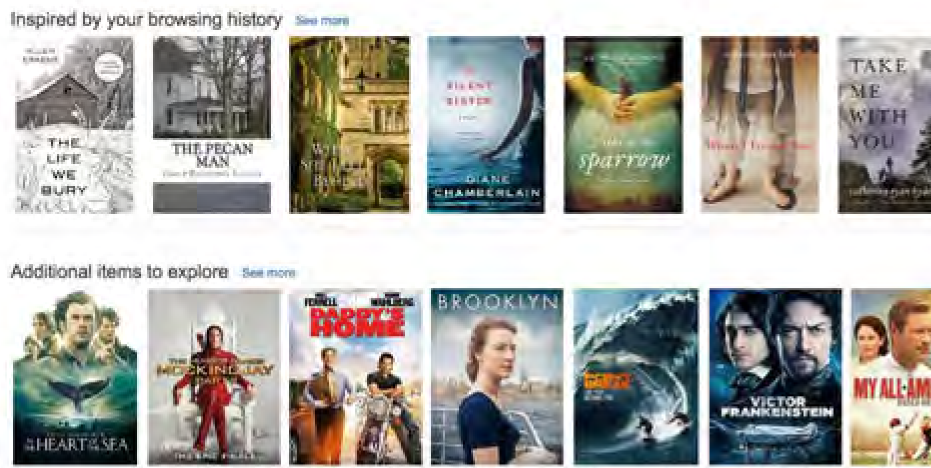


Figure 2.3: Recommendations on the Amazon home page.

Web," implying that the experience of the site should be completely personalised to the needs of individual customers (Barbieri et al., 2014). In the same year, Amazon attributed 35 % of its sales to recommendations (Aufreiter et al., 2012).

Netflix is another company to have invested heavily in recommendations over the last decade. Netflix owns a large catalogue of films and television series, to which it gives streaming access for a monthly subscription fee. While technically not an e-retail company, we include Netflix here because of its tremendous impact on RS research. In 2006, Netflix launched the now famous Netflix Prize Competition (Bennet and Lanning, 2007). It offered one million US dollars to the research team that could first improve their rating prediction algorithm above some predetermined threshold. The competition was successfully concluded in 2009, when the leading team achieved the specified improvement. It spurred a flurry of research in RS's, and is widely considered to have catalysed the high level of interest we see in the field today (Barbieri et al., 2014).

Figure 2.4 shows an example of recommendations shown to customers on the home screen of the Netflix iOS application. The experience is very similar to the Amazon home page. The top and bottom recommendation strips read, "Because you watched ...", which implies that recommendations have been inspired by viewing behaviour of the customer. In addition, the recommendations completely dominate the application, filling almost the entire home screen. One difference to figure 2.3 is that fewer recommendations can be seen by the customer at any given time.

Amazon and Netflix are not the only companies who rely on RS's for a portion of their business functions. Yahoo has been heavily involved in RS research. In fact, a team involving researchers from Yahoo won the Netflix competition (Koren, 2009). Yahoo uses RS's extensively for e-retail, news and content recommendation. YouTube has a RS for personalised video

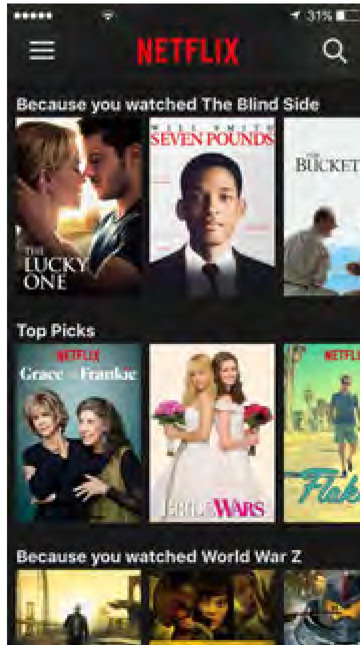


Figure 2.4: Recommendations on Netflix iOS application home screen.

recommendations. Spotify uses a RS for recommending music (Johnson, 2014). Facebook and LinkedIn make use of RS's for suggesting social and professional connections (Ricci et al., 2015).

Some companies have made RS's their core business. These companies effectively run RS's "as a service". This allows e-retail startup companies to implement the functionality of a RS without the prohibitive capital expenditure and expertise required to build a RS in-house. Companies who provide a RS as a service include Smart Focus[5], RichRelevence[6], and Salesforce, with their Marketing cloud[7] product.

Ricci et al. (2015) have compiled a fairly comprehensive handbook covering the various aspects of a RS. They reference a number of factors which may contribute to interest in a RS from a general service provider. These are largely consistent with observations made in our own research, so we have adapted them for the more specific context of e-retail. These factors include; selling more products, selling more diverse products, increasing customer satisfaction, and better understanding their customers.

Selling more products is arguably the most important factor in considering the utility of a RS to the e-retailer. Effective personalised recommendations achieve this goal because products it recommends are likely to be of interest to the customer. As a result, the customer is more likely to make a purchase than if the RS were not present.

Another major function of a RS is to enable the customer to purchase products that might

[5] http://www.smartfocus.com/
[6] http://www.richrelevance.com/
[7] http://www.exacttarget.com/

be difficult to find without a precise recommendation. For example, an e-retailer with a third-party marketplace needs to ensure that infrequent items from third-party sellers receive their fair-share of attention. This could be challenging without a RS since the e-retailer cannot afford to degrade customer experience by exposing customers to products in which they have no interest. An effective RS recommends infrequent products to the correct customers to ensure that sellers get their fair-share of exposure.

A well designed RS is likely to increase customer satisfaction with the experience of shopping. It is plausible that customers find effective recommendations relevant and interesting, and in combination with a well designed user interface (UI), they are likely to enjoy engaging with the system. As a result, their level of interaction with the system will increase, and with it the likelihood that they make a purchase.

Finally, RS's provide opportunities for insight into customer behaviour beyond recommendations. For example, preferences modelled by the RS might serve as the basis for clustering customers based on similar preferences. These clusters could then be analysed according to a variety business metrics, such as average transaction value. Marketing efforts could then be directed at clusters whose average purchase value is low. This provides a more targeted means for improving customer lifetime value.

The utility of a RS to the e-retailer holds under the assumption that recommendations are accurate. Accuracy can be evaluated along several axes, and this issue is deconstructed in detail in section 2.8. It essentially boils down to the property that products recommended by a RS should be of interest to customers. This is a difficult task. In figure 2.3, The Amazon RS has a dozen chances to recommend relevant products from a catalogue of tens of millions. To recommend for the mobile context is even more challenging. As we have mentioned, in figure 2.4 Netflix has even fewer chances to recommend effectively. Thus, accurate recommendations becomes our first characteristic of an effective RS.

Two technical characteristics of an effective RS also arise from this discussion, namely low latency and durability. We have illustrated the central role of a RS in the websites and applications of an e-retailer. The RS is one of the first systems that customers interact with when entering the platform. It is therefore essential that they respond within the window of interactivity. Recommendations are useless if they arrive after a customer has left the application. To prevent this scenario, recommendations should be served in a matter of milliseconds.

An even worse scenario than the one outlined above is where recommendations never appear. In this case, we can be sure that the customer closes the application without further interaction. As a result, should the RS fail totally, the usability of the entire e-retail platform is put in jeopardy. Thus, the design of a RS should preclude total failure. The RS should always be able to at least serve recommendations, even under degraded operating conditions.

In the following section, we introduce the various data sources that the RS might use for generating recommendations.

2.5 Data Sources

Data sources used for generating recommendations can be diverse and of large scale (Crankshaw et al., 2014). Whether this data can be exploited for generating recommendations is largely dependent on the techniques supported by a RS. Ricci et al. (2015) describe recommendation techniques as either "knowledge poor" or "knowledge dependent". Knowledge poor techniques employ algorithms that do not require much information about the entities described in the data sources. For example, some techniques only need to know a unique identifier of a product or customer, and infer preferences through relationships between these identifiers discovered in the transaction data (Hu et al., 2008; Rendle et al., 2009). Knowledge dependent techniques require much more knowledge about the entities described in the data sources. For example, some techniques require descriptions of products, or information about the category tree (Zhang et al., 2014; Kanagal et al., 2012; Ahmed et al., 2013). We give a detailed overview of various knowledge poor and knowledge dependent recommendation techniques in section 2.7. In general, RS data sources in e-retail describe products, customers, and interactions between customers and products.

Products are comprised of a wide range of attributes, any of which may contribute to the interest of a customer in a product. When dealing with a large and diverse catalogue of products, attributes are largely dependent on the type of product. For example, a film DVD could be described by the director and actors, while a T-shirt could be described by brand or style. Some attributes are shared across products, for example, all products have a unique identifier, a description and a price.

In order to personalise recommendations for a specific customer, a RS can model the customer along a number of axes. Some RS's model customers as a function of their interactions with products (Hu et al., 2008). In other cases, sociodemographic information such as age, gender, location, and profession may be used (Ricci et al., 2015). Customer data might also include information about trust relationships between customers. In this case, the RS might utilise this information to recommend products that are preferred by friends.

Customer interactions with a catalogue of products can be placed in one of two categories; explicit feedback and implicit feedback. Explicit feedback includes direct indications from customers regarding their preference for products. For example, a RS might collect ratings by asking customers to rate the products they have bought. These ratings can take various forms, including (Schafer et al., 2007):

- Numeric ratings, which might be on a scale of one to five. Amazon and Netflix are examples of companies that use this strategy.
- Binary ratings, where a customer is asked to indicate whether they "like" or "dislike" a product. YouTube's rating system is an example of this scheme.

Explicit feedback might also include extra customer-generated metadata about their interactions with a product (Lops et al., 2011). For example, Amazon's rating system includes the ability to add comments and pictures relating to their experience with a product.

More abundant though is implicit feedback, which indirectly reflects customer opinion through behaviour (Hu et al., 2008). Ricci et al. (2015) provide the example of a customer searching for a book on Amazon. When a customer searches for a product, they are provided with a list of candidates. If the customer clicks on a candidate product, we might assume that the customer is somewhat interested in that product. A stronger indication of customer interest arises when a customer purchases one of the products. As a result, transaction data is a widely used form of implicit feedback (Rendle et al., 2009; Kanagal et al., 2012; Ahmed et al., 2013; Zhang et al., 2014).

Much of the earlier research on recommendation techniques focuses on scenarios where customers provide explicit feedback. This is probably due to the convenience of the data; we know that a customer dislikes some products and approves of others based on the opinions which they have explicitly expressed. On the other hand, there exists a fundamental asymmetry in the implicit case – there is only positive feedback (Hu et al., 2008). We can only infer which products a customer probably favours by observing their consumption patterns. The non-observed products are a mixture of real negative feedback (a lack of interest on the part of the customer) and missing data (the customer may want to consume the product in the future) (Rendle et al., 2009). Even then, our inference is noisy because we cannot be sure of the true motive and preference behind a behaviour. Consider, for example, the case where a customer purchases an product as a gift. Consequently, modelling with explicit data has fewer challenges.

In many cases however, the recommender system needs to be centred on implicit feedback. This could be due to the reluctance of customers to rate products, or limitations of the e-retail platform that prevent efficient collection of explicit feedback.

We have already seen that the data sources which a RS might utilised are diverse. They also quickly become extremely large. Consider an example where a RS models customers based on their interaction with the site. In this case, a RS might collect all of the product views, searches, purchases and clicks. Aggregated over hundreds of thousands or millions of customers, this dataset quickly becomes extremely large. This has implications for both storage and processing capabilities of a RS. More specifically, a RS should use scalable storage solutions, and the techniques with which the recommendation models are built should also scale gracefully.

In the next section, we present a theoretical framework against which recommendation algorithms can be developed. Before we move on, however, we must consider the specific context for which our prototype has been built. When our research commenced, the Takealot platform had limited support for collection of explicit feedback data. As a result, we have limited the scope of our prototype to utilising implicit feedback data. Thus, our theoretical framework and the techniques we describe subsequently focus on modelling implicit feedback data.

2.6 Theoretical Framework

This section generally follows the notation of Rendle and Freudenthaler (2014). Let,

$$U = \{u_1, \ldots, u_M\}$$

be a set of M customers and,

$$I = \{i_1, \ldots, i_N\}$$

a set of N products. We reserve special indexing letters for distinguishing customers from products: for customers u, v and for items i, j.

Then,

$$S \subseteq U \times I$$

is the set of observed customer actions, which could, for example, represent purchases of products by customers. Also, let us denote all the unobserved customer-product pairs as:

$$P = (U \times I) \setminus S.$$

The objective of an effective RS is to find a ranking $\hat{r}_u$ of all products in I for each customer u in U. This can be formulated as a bijective function:

$$\hat{r}_u : I \to \{1, 2, \ldots, |I|\} \forall u \in U.$$

Now, $\hat{r}_u(i)$ is the rank of product i for a customer u. The ranking function is generally modelled through some scoring function $\hat{y}_u(i)$, where ranking for each u is performed by computing scores for all i and sorting by scores. The formal link between $\hat{r}$ and $\hat{y}$ can be defined as:

$$\hat{r}_u(i) := |\{j : \hat{y}_u(j) \geq \hat{y}_u(i)\}|. \tag{2.1}$$

Equation (2.1) can be explained using the rank one product as an example; there is only one item (itself) that is as large or larger.

Finally, $\hat{y}$ is parametrised by a set of model parameters $\vec{\Theta}$, which in turn uniquely define the ranking $\hat{r}$. Thus, the goal of a recommendation algorithm is to find the $\vec{\Theta}$ that give us a set of rankings that are closest to customers' actual preferences. The following section details some of the models that attempt to achieve this goal.

2.7 Recommendation Techniques

Two main paradigms for generating recommendations have emerged from the research on RS's, namely, content-based filtering (CBF) algorithms and collaborative filtering (CF) algorithms (Lops et al., 2011). CBF techniques try to recommend products similar to those with which a customer has interacted in the past. CF algorithms attempt to match similar customers and cross-recommend products with which they have interacted. A third strategy is to combine these techniques with powerful hybrid methods. The existence of diverse recommendation techniques suggests that the effective RS provides extensible software interfaces with which the data scientist can easily implement new algorithms.

2.7.1 Content-Based Filtering

CBF algorithms typically build a model, or profile, of a customer based on attributes of the products with which the customer has interacted. Then, recommendations are generated by matching the profile with products which have a similar profile. This process can be deconstructed into three steps (Lops et al., 2011). First, the attributes of products are organised to form some structured representation. Next, the customer profile is constructed by aggregating profiles of the products with which they have interacted. Finally, relevant products are matched to aggregated customer profiles to generate recommendations.

The structure of product profiles is largely dependent on attributes already available to the system. In section 2.5 we discussed some of the possible product attributes. Many of these attributes already possess structure, for example "price" is a structured attribute because it represents a clearly defined idea and has a known type. Structured attributes can be included "as is" in the product profile. On the other hand, when attributes have no structure, a pre-processing step is required to extract structured concepts. For example, descriptions of products are unstructured "blobs" of text. However, these descriptions may share common themes and ideas, which could be extracted into structured sets of attributes. One simple way of structuring product descriptions is to use Term Frequency-Inverse Document Frequency (TF-IDF) analyses (Lops et al., 2011). More advanced techniques include Latent Dirichlet Allocation (LDA) (Hu et al., 2014), and Neural Network-based approaches like Word2Vec (Musto et al., 2015).

Customer profiles are generated through a process known as "profile learning" (Lops et al., 2011). The idea is to create a representative profile of the customer by combining profiles of products with which the customer has interacted. One strategy for "learning" a customer profile is to train a supervised model to predict the customer's preference, given a product profile. Then, to generate recommendations for a customer, preference predictions are made for each product in the catalogue, and products with the highest preference are returned to the customer.

CBF has an important drawback. There is a limit to the number and types of features that can be associated with products. As a result, the RS cannot make suitable recommendations if the product attributes do not fully discriminate products that a customer prefers from the ones they dislike (Lops et al., 2011). For example, a customer might be interested in films starring a certain actor. Should the product attributes not include a field relating to actors, the RS could not recognise the customer's interest in that actor. As a result, films starring that actor would most likely not be recommended.

2.7.2 Collaborative Filtering

CF offers a more flexible approach. CF methods produce recommendations based on patterns of customer behaviour without the need for collecting specific information about either products or customers. There are two main approaches to CF, namely neighbourhood methods and latent factor (LF) models (Koren et al., 2009).

Neighbourhood Methods

Neighbourhood methods focus on the relationships between products or between customers. Customer-oriented approaches estimate unknown customer preferences based on known preferences of like-minded customers. In product-oriented approaches preferences are estimated using known preferences of the same customer for similar products. Product-oriented approaches have a number of advantages over the customer-oriented approaches. First, they are more suited to explain how certain predictions are made, a desirable attribute for any recommendation system (Ricci et al., 2015). The reason for this is that customers are familiar with products with which they have previously interacted, but are unlikely to know like-minded customers. In many cases, product-oriented approaches are also more efficient than customer-oriented approaches. This is because, typically, the number of products in a data set is much smaller than the number of customers, which allows precomputation of all product similarities for retrieval as needed (Bell and Koren, 2007).

Central to many product-oriented approaches is a similarity function specifying the degree

of similarity between products. For this discussion, we denote the similarity between products pairs i and j as t_{ij}. Most approaches consider the explicit feedback case where customers have assigned ratings, s_{ui}, to all pairs in S, but no ratings to pairs in P. Thus the goal is to predict the rating that u would assign to j. The inference proceeds as follows; first we identify the k products rated by u which are most similar to j, denoted by $T_u^k(j)$. The prediction is then a weighted average of the ratings for neighbouring items (this approach is known as k-nearest neighbours (kNN)):

$$\hat{y}_u(j) = \frac{\sum_{i \in T_u^k(j)} t_{ij} s_{ui}}{\sum_{i \in T_u^k(j)} t_{ij}}.$$

The t_{ij} have a central role to play in this scheme as they are used for both selected the k-nearest neighbours and for weighting the above average.

Frequently, these similarities are based on the Pearson correlation coefficient, ρ_{ij}, which measures the tendency of customers to rate products similarly. To define the Pearson correlation coefficient, let us introduce U^{ij} to denote the set of customers that have purchased products i and j:

$$U^{ij} := \{u \in U : (u,i) \in S \wedge (u,j) \in S\}.$$

Then, the Pearson correlation coefficient between two products is defined as follows:

$$\rho_{ij} = \sum_{u \in U^{ij}} \frac{(s_{ui} - \mu_i)(s_{uj} - \mu_j)}{\sigma_i \sigma_j}, \tag{2.2}$$

where μ_i, μ_j, and σ_i, σ_j are the empircal means and empirical standard deviations, respectively, of the ratings for products i and j. Since the interaction of customers with products is sparse, it is expected that some products only share a few common customers. Computation of equation (2.2) is based only on common customer support. Thus, similarities based on a greater customer support can be considered more reliable. To incorporate this idea, Koren (2008) propose a "shrunk" correlation coefficient as the similarity metric:

$$s_{ui} = \frac{|U^{ij}|}{|U^{ij}| + \lambda} \rho_{ij}.$$

According to Koren (2008), a typical value for λ is 100.

Enhancements to this approach include correcting for biases in the average ratings of different customers and products. For example, some customers tend to rate products higher than others (Koren, 2008). However, modifications of this type are less relevant to implicit feedback. When working with implicit feedback data, ratings are replaced by the frequencies with which products are consumed by the same customer, such as the number of times a product has been bought. Frequencies for customers might have a very different scale depending on their level of activity, and thus it is less clear how to calculate similarities. A deeper flaw is that traditional

neighbourhood methods do not allow us to express the level of confidence we have in observed customer preference (Hu et al., 2008). In order to address this specific issue, Hu et al. (2008) have introduced LF models for dealing with implicit data.

Implicit Matrix Factorisation

LF models comprise an alternative approach to CF. The key insight of LF models is that a vectorised representation of customers and products can be transformed to the same latent space, inferred only from observed preference data (Barbieri et al., 2014). Models of this type played a central role in the winning entry of the Netflix Prize Competition. Particularly successful was the subclass of LF models, named matrix factorisation models (MF), induced by the Singular Value Decomposition (SVD) of the matrix made up of the ratings assigned to customer-product pairs in S (Koren and Bell, 2011). Consequently, Hu et al. (2008) develop their method, called implicit matrix factorisation (IMF), from this perspective, with the goal of solving some of the issues facing implicit feedback data.

Hu et al. (2008) tackle the case of recommending television shows. Pairs in S are assigned values s_{ui} based on the number of times u fully watched show i. For example, if u watches 70 % of show i, s_{ui} is set to 0.7, and if u watches the show twice, s_{ui} is set to 2. For the setting of e-retail, Hu et al. (2008) suggest that s_{ui} indicate the number of times u purchases i, but it could also indicate the number of times u viewed i.

Two sets of variables, q_{ui} and c_{ui}, are introduced to formalise the notion of confidence that we have in the preferences that s_{ui} measure. The q_{ui} are introduced to indicate that we believe u to have a preference for i, and are derived by binarising the sets S and P:

$$q_{ui} = \begin{cases} 1 & \text{if } (u,i) \in S, \\ 0 & \text{if } (u,i) \in P. \end{cases}$$

In other words, if a customer interacts with a product, we believe that there is some preference for that product, whereas if a customer does not consume a product, we assume no preference for that product.

However, there are greatly varying levels of confidence in our belief of a customer's preference for a product. Beyond not liking the product, there might be multiple reasons the customer does not interact with it. For example, they might not know that the product exists. Thus, inherent in our belief that $q_{ui} = 0$ is a low level of confidence. On the other hand, there is a high level of confidence that the customer has preference for the product if they regularly interact

with it. The set of variables c_{ui} are designed to capture this notion:

$$c_{ui} = \begin{cases} 1 + \alpha s_{ui} & \text{if } (u,i) \in S, \\ 1 & \text{if } (u,i) \in P. \end{cases}$$

Thus, there is some minimal confidence in our belief that $q_{ui} = 0$. As we observe more interaction between u and i, our confidence that $q_{ui} = 1$ increases. The rate at which our confidence increases is controlled by the hyperparameter α.

The goal of IMF is to find customer factors and product factors that "factor" the user preferences while simultaneously accounting for varying confidence levels in those preferences. More specifically, we wish to find customer factors, $\vec{x_u} \in \mathbb{R}^f$ and product factors, $\vec{w_i} \in \mathbb{R}^f$ such that q_{ui} is as close to $\vec{x_u}^T \vec{w_i}$ as possible, and the contribution of factors to the loss function is weighted by our confidence, c_{ui}. Therefore, factors are computed by minimising the following cost function with respect to the product factors and customer factors:

$$C_{IMF} = \sum_{u,i} c_{ui} \left(q_{ui} - \vec{x_u}^T \vec{w_i} \right)^2 + \lambda \left(\sum_u \vec{x_u}^T \vec{x_u} + \sum_i \vec{w_i}^T \vec{w_i} \right). \tag{2.3}$$

The second term in equation (2.3) is necessary for regularising the model to prevent overfitting. The level of regularisation is controlled by the hyperparameter λ.

An alternating least squares (ALS) procedure is used as an efficient optimisation process. It is derived by observing that when customer factors or product factors are fixed, the cost function in equation (2.3) becomes quadratic, so its global minimum is readily computed by differentiation. First, we fix the user factors $\vec{x_u}$ and find the derivative of C_{IMF} with respect to

each item factor, $\vec{w_i}$:

$$\begin{aligned}
\frac{\partial C_{IMF}}{\partial \vec{w_i}} &= \sum_u -2c_{ui}\left(q_{ui} - \vec{x_u}^T\vec{w_i}\right)\vec{x_u} + 2\lambda\vec{w_i} \\
&= \sum_u -2c_{ui}q_{ui}\vec{x_u} + 2c_{ui}\vec{x_u}\vec{x_u}^T\vec{w_i} + 2\lambda\vec{w_i} \\
&= -2\begin{bmatrix}\vec{x_{u_1}} & \vec{x_{u_2}} & \cdots & \vec{x_{u_M}}\end{bmatrix}\begin{bmatrix}c_{u_1,i} & 0 & \dots & 0 \\ 0 & c_{u_2,i} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c_{u_M,i}\end{bmatrix}\begin{bmatrix}q_{u_1,i} \\ q_{u_2,i} \\ \vdots \\ q_{u_M,i}\end{bmatrix} \\
&\quad + 2\begin{bmatrix}\vec{x_{u_1}} & \vec{x_{u_2}} & \cdots & \vec{x_{u_M}}\end{bmatrix}\begin{bmatrix}c_{u_1,i} & 0 & \dots & 0 \\ 0 & c_{u_2,i} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c_{u_M,i}\end{bmatrix}\begin{bmatrix}\vec{x_{u_1}}^T \\ \vec{x_{u_2}}^T \\ \vdots \\ \vec{x_{u_M}}^T\end{bmatrix}\vec{w_i} + 2\lambda\vec{w_i}.
\end{aligned}$$

Let us define, $X = \begin{bmatrix}\vec{x_{u_1}}^T \\ \vec{x_{u_2}}^T \\ \vdots \\ \vec{x_{u_M}}^T\end{bmatrix}$, $C^i = \begin{bmatrix}c_{u_1,i} & 0 & \dots & 0 \\ 0 & c_{u_2,i} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c_{u_M,i}\end{bmatrix}$, and $\vec{q^i} = \begin{bmatrix}q_{u_1,i} \\ q_{u_2,i} \\ \vdots \\ q_{u_M,i}\end{bmatrix}$. Thus,

$$\frac{\partial C_{IMF}}{\partial \vec{w_i}} = -2X^TC^i\vec{q^i} + 2X^TC^iX\vec{w_i} + 2\lambda\vec{w_i}.$$

Now, setting $\frac{\partial C_{IMF}}{\partial \vec{w_i}} = 0$ and solving for $\vec{w_i}$ we get:

$$\begin{aligned}
&-2X^TC^i\vec{q^i} + 2X^TC^iX\vec{w_i} + 2\lambda\vec{w_i} = 0 \\
&\implies \left(X^TC^iX + \lambda I\right)\vec{w_i} = X^TC^i\vec{q^i} \\
&\implies \vec{w_i} = \left(X^TC^iX + \lambda I\right)^{-1}X^TC^i\vec{q^i}.
\end{aligned}$$

To minimise C_{IMF} with respect to each $\vec{x_u}$, we fix the product factors $\vec{w_i}$ and follow the same argument as above:

$$\vec{x_u} = \left(W^TC^uW + \lambda I\right)^{-1}W^TC^u\vec{q^u},$$

where $W = \begin{bmatrix}\vec{w_{i_1}}^T \\ \vec{w_{i_2}}^T \\ \vdots \\ \vec{w_{i_N}}^T\end{bmatrix}$, $C^u = \begin{bmatrix}c_{u,i_1} & 0 & \dots & 0 \\ 0 & c_{u,i_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c_{u,i_N}\end{bmatrix}$, and $\vec{q^u} = \begin{bmatrix}q_{u,i_1} \\ q_{u,i_2} \\ \vdots \\ q_{u,i_N}\end{bmatrix}$.

The factors will stabilise after a few “sweeps” of fixing customer factors and updating each

product factor, then fixing product factors and updating each customer factor.

The computational bottleneck during each sweep is computing the matrix product $X^T C^i X$ when updating product factors, and $W^T C^u W$ when updating the customer factors. The naive calculation of these products is $O(f^2 M)$ and $O(f^2 N)$ (for each of the N products and M customers), respectively. If we assume the matrix inversion is $O(f^3)$, a complete update of product vectors is $O(f^2 NM + f^3 N)$, while a complete update of customer factors is $O(f^2 NM + f^3 M)$. This has poor scaling properties. Ideally, we would like the algorithm to scale linearly with the size of the preference data, S, but this algorithm is polynomial in N and M.

By exploiting the structure of the variables, Hu et al. (2008) reduce the complexity of the update to $O(f^2|S| + f^3 N)$ for product factors and $O(f^2|S| + f^3 M)$ for customer factors. In the context of an update of the customer factors, the argument is as follows; we can improve the computational bottleneck mentioned above by observing that,

$$W^T C^u W = W^T W - W^T W + W^T C^u W = W^T W + W^T (C^u - I) W.$$

Here, $W^T W$ is independent of u, so we can compute this product with $O(f^2 N)$ before updating each of the customer factors. Note also that $C^u - I$ has only $|I^u|$ non-zero entries, where I^u is the set of products bought by u, which is typically much smaller than I:

$$I^u := \{i \in I : (i, u) \in S\}. \tag{2.4}$$

This means that the computation of $W^T W + W^T (C^u - I) W$ can be $O(f^2 |I^u|)$. If we again assume $O(f^3)$ for the matrix inversion, a single x_u can be computed in $O(f^2|I^u| + f^3)$, including computing $W^T C^u \vec{q^u}$ in $O(f|I^u|)$. This is repeated for all M customers to get the total of $O(f^2|S| + f^3 M)$. We can use the same technique for the product factors to update them in time $O(f^2|S| + f^3 N)$. Importantly, this algorithm is linear in the size of the input data, S. The full algorithm as presented in Hu et al. (2008) is shown in algorithm 1.

Algorithm 1 ALS procedure from Hu, et al.

Input: Initial estimates of $\vec{x_u}$ and $\vec{w}_i$.
Output: Stable estimates of $\vec{x_u}$ and $\vec{w}_i$.

1: **for** 1,...,nsweeps **do**
2: $\quad Z \leftarrow X^T X$
3: $\quad$ **for** i in I **do**
4: $\quad\quad w_i \leftarrow [Z + X^T (C^i - I) X + \lambda I]^{-1} X^T C^i \vec{q^i}$
5: $\quad$ **end for**
6: $\quad H \leftarrow W^T W$
7: $\quad$ **for** u in U **do**
8: $\quad\quad \vec{x_u} \leftarrow [H + W^T (C^u - I) W + \lambda I]^{-1} W^T C^u \vec{q^u}$
9: $\quad$ **end for**
10: **end for**

Recall that in order to find the ranking $\hat{r}_u$, we need a function which will score the products for each customer (see equation (2.1)). In the case of IMF, the scoring function is defined as:

$$\hat{y}_u(i) = \vec{x_u}^T \vec{w_i}.$$

A major drawback of this method is that all elements (in P) that the model is required to rank in the future are presented to the learning algorithm as negative feedback during training. Specifically, the model is optimised to predict a value of one for elements in S and zero for elements in P. Thus, a model that has the ability to fit the training data exactly cannot rank at all as it predicts only zeros. The only reason the model has any ranking ability is that it uses regularisation to prevent overfitting (Rendle et al., 2009). To alleviate this issue, Rendle et al. (2009) introduce Bayesian personalised ranking (BPR), which we described in the next section.

Bayesian Personalised Ranking

BPR is a more natural interpretation of the recommendation problem. While Hu et al. (2008) focus on scoring individual products, then ranking, BPR optimises directly for correctly ranking pairs of products. The idea for each customer is to discriminate between products that have been selected and those products that have not yet been selected (Ahmed et al., 2013).

To derive their optimisation criterion, Rendle et al. (2009) formalise the task of a RS. They argue that the task of a RS is is to provide a customer with a personalised total order $>_u \subset I \times I$. The following statements hold for $>_u$:

1. *antisymmetry*, if $i >_u j$ and $j >_u i$ then $i = j$;
2. *transitivity*, if $i >_u j$ and $j >_u k$ then $i >_u k$; and
3. *totality*, $i >_u j$ or $j >_u i$.

Rendle et al. (2009) further argue that we can reconstruct parts of $>_u$ from the observations in S. This is done by enforcing the assumption that a product i is preferred by a customer u over another product j if i, but not j has been selected by u, i.e. $i >_u j$. A dataset, D_S, can then be constructed that is an observed subset of $>_u$:

$$D_S \coloneqq \{(u, i, j) : i \in I^u \wedge j \in I \setminus I^u\}, \tag{2.5}$$

where I^u has been defined in equation (2.4).

The Bayesian treatment of finding the correct personalised ranking over all products is to maximise the following posterior probability with respect to the parameter vector $\vec{\Theta}$ (all $\vec{x_u}$ and

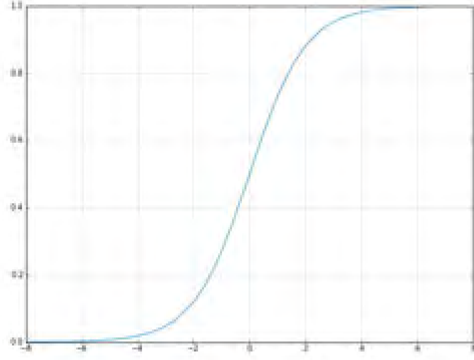


Figure 2.5: Logistic sigmoid loss function, $\sigma(x) = \frac{1}{1+\exp(-x)}$. Very positive values of x result in $\sigma(x)$ being closer to one, while very negative values of x result in $\sigma(x)$ being closer to zero. At $x = 0$, $\sigma(x)$ is 0.5.

$\vec{w_i}$):

$$p\left(\vec{\Theta}\middle|D_S\right) \propto p\left(D_S\middle|\vec{\Theta}\right) p\left(\vec{\Theta}\right).$$

In order to achieve a total order, we further define the likelihood as follows:

$$p\left(D_S\middle|\vec{\Theta}\right) = \prod_{(u,i,j)\in D_S} \sigma\left(\hat{y}_u(i) - \hat{y}_u(j)\right), \tag{2.6}$$

where $\sigma(x)$ is the logistic sigmoid (see figure 2.5):

$$\sigma(x) = \frac{1}{1+\exp(-x)}. \tag{2.7}$$

In addition, we know from section 2.6 that $\hat{y}_u(i)$ is the function which scores product i for u, parametrised by $\vec{\Theta}$. Implicit in equation (2.6) is that customers act independently, and the ordering of any pair of items for a specific customer is independent of the ordering of any other pair.

With equation (2.6), Rendle et al. (2009) have defined the probability that a u actually prefers i to j as $\sigma(\hat{y}_u(i) - \hat{y}_u(j))$. This definition of probability makes sense. Consider the case when i actually ranks above j in the preference raking of u; the model should assign to i a higher score than j, and hence $\hat{y}_u(i) - \hat{y}_u(j)$ would be positive. The shape of the logistic sigmoid in figure 2.5 tells us that a positive difference results in a probability of correctly deducing u's preference that is greater than 0.5, which is what we would expect. Furthermore, the larger the difference, the higher the probability that we correctly deduce u's relationship with i and j.

Moving on to the prior probability, we assume that every entry in $\vec{\Theta} \in \mathbb{R}^k$ is independently

sampled from a normal distribution $\mathbf{N}\left(0, \frac{1}{\lambda}\right)$, which results in:

$$
\begin{aligned}
p(\vec{\Theta}) &= \prod_{i=1}^{k} \left(2\pi \frac{1}{\lambda}\right)^{-\frac{1}{2}} \exp\left(-\frac{\lambda}{2}\theta_i^2\right) \\
&= \left(2\pi \frac{1}{\lambda}\right)^{-\frac{k}{2}} \exp\left(-\frac{\lambda}{2}\sum_{i}^{k}\theta_i^2\right).
\end{aligned}
$$

Thus, the full posterior distribution becomes:

$$
p\left(\vec{\Theta}\middle|D_S\right) \propto \prod_{(u,i,j)\in D_S} \sigma\left(\hat{y}_u\left(i\right) - \hat{y}_u\left(j\right)\right) \exp\left(-\frac{\lambda}{2}\vec{\Theta}^T\vec{\Theta}\right). \tag{2.8}
$$

The parameters in $\vec{\Theta}$ are found by maximising equation (2.8) with respect to $\vec{\Theta}$. In practice, it is easier to minimise the negative log-posterior, so our cost function, C_{BPR}, becomes:

$$
\begin{aligned}
C_{\text{BPR}} &= -\log p\left(\vec{\Theta}\middle|D_S\right) \\
&= \sum_{u\in U}\sum_{i\in I^u}\sum_{j\in I\setminus I^u} -\log\sigma(\hat{y}_u(i) - \hat{y}_u(j)) + \frac{\lambda}{2}\vec{\Theta}^T\vec{\Theta}
\end{aligned}
$$

Note that the hyperparameter λ is arbitrary, so we drop the multiplicative constant of $\frac{1}{2}$ in all further discussions involving C_{BPR}.

Given that C_{BPR} is differentiable with respect to $\vec{\Theta}$, gradient descent based algorithms are a natural choice. There are two popular flavours of gradient descent: full gradient descent or stochastic gradient descent. In the first case, the entire data set is processed to compute a gradient. A small step is then taken toward the minimum with learning rate η:

$$
\vec{\Theta} \leftarrow \vec{\Theta} - \eta\frac{\partial C_{\text{BPR}}}{\partial\vec{\Theta}}.
$$

Rendle et al. (2009) note that in practice full gradient descent is not feasible as the training data contains $O(|S||I|)$ triples. Thus, a stochastic gradient descent (SGD) algorithm is proposed. In this scheme, we uniformly sample a random triple $(u, i, j) \in D_S$, and take a step in the direction of the minimum of the negative log-posterior. The idea is that after many such updates, the parameters of the model will converge to stable estimates. We derive the update step by differentiating a point-wise error function, c_{BPR}, with respect to $\vec{\Theta}$:

$$
\begin{aligned}
c_{\text{BPR}} &\propto -\log\sigma\left(\hat{y}_u\left(i\right) - \hat{y}_u\left(j\right)\right) + \lambda\vec{\Theta}^T\vec{\Theta} \\
\frac{\partial c_{\text{BPR}}}{\partial\vec{\Theta}} &\propto -\left[1 - \sigma\left(\hat{y}_u\left(i\right) - \hat{y}_u\left(j\right)\right)\right] \cdot \frac{\partial}{\partial\vec{\Theta}}\left(\hat{y}_u\left(i\right) - \hat{y}_u\left(j\right)\right) + \lambda\vec{\Theta}.
\end{aligned}
$$

Then, the step is taken in the opposite direction of the gradient:

$$\vec{\Theta} \leftarrow \vec{\Theta} - \eta \frac{\partial c_{\text{BPR}}}{\partial \vec{\Theta}}. \tag{2.9}$$

BPR is a generic optimisation criteria in that it can be applied to a range of model classes. Rendle et al. (2009) apply it to neighbourhood models and LF models. As LF models are of particular interest to us, we continue our discussion of BPR using this model as a concrete example.

As in IMF, our scoring function, $\hat{y}_u(i)$, is defined as:

$$\hat{y}_u(i) = \vec{x_u}^T \vec{w_i}.$$

The parameters of the model, $\vec{\Theta}$, become the $\vec{x_u}$'s, $\vec{w_i}$'s and the $\vec{w_j}$'s. Thus, the update step from equation (2.9) becomes:

$$\begin{aligned}
\vec{x_u} &\leftarrow \vec{x_u} - \eta \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] (\vec{w_i} - \vec{w_j}) + \lambda \vec{x_u} \right) \\
\vec{w_i} &\leftarrow \vec{w_i} - \eta \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] \vec{x_u} + \lambda \vec{w_i} \right) \\
\vec{w_j} &\leftarrow \vec{w_j} - \eta \left(\left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] \vec{x_u} + \lambda \vec{w_j} \right).
\end{aligned}$$

One extension suggested by Zhang and Jordan (2015) is that each variable has its own learning rate, η_u, η_i and η_j. We adopt this methodology going forward. The full BPR algorithm is shown in algorithm 2.

Algorithm 2 BPR procedure from Rendle et al. (2009)

Input: Initial estimates of $\{X, W\}$
Output: Stable estimates of $\{X, W\}$
1: **repeat**
2: draw (u, i, j) from D_S
3: $\vec{x_u} \leftarrow \vec{x_u} - \eta_u \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] (\vec{w_i} - \vec{w_j}) + \lambda \vec{x_u} \right)$
4: $\vec{w_i} \leftarrow \vec{w_i} - \eta_i \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] \vec{x_u} + \lambda \vec{x_i} \right)$
5: $\vec{w_j} \leftarrow \vec{w_j} - \eta_j \left(\left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] \vec{x_u} + \lambda \vec{x_j} \right)$
6: **until** convergence

Rendle et al. (2009) evaluate their BPR MF model on two data sets; an e-retail dataset and the Netflix dataset. For both datasets, BPR results in better performance compared to IMF on the Area Under the Receiver Operating Characteristic Curve (AUC), which we describe in detail in section 2.8.1.

While BPR improves on IMF, it does not address the “cold start” problem common to basic CF models. Cold start refers to the ability of a model to recommend products that have not

yet been purchased and recommend to customers that have not yet made a purchase. CBF models can recommend new products since they rely only on the attributes of the product. CF models rely completely on interaction data to build their recommendations. As a result, if no interactions are available for a given customer or product, the model cannot generate recommendations for the customer, and cannot recommend the product. In an effort to address the cold problem, amongst others, researchers have developed hybrid methods that combine CBF and CF models to incorporate the advantages of both models. In the following section, we an example of a hybrid method that incorporates the taxonomy of a catalogue of products.

2.7.3 Hybrid Methods

Hybrid methods for generating recommendations have, in recent years, consistently outperformed pure CF models (Kanagal et al., 2012; Ahmed et al., 2013; Zhang et al., 2014; Agarwal and Chen, 2009). In the development of our prototype, we have focused on first supporting a number of simple models, namely IMF and BPR. However, we remain mindful of more complex models which the system should support in the future. As a result, in this section, we give an example in the form of the Taxonomy-Aware Latent Factor Model (TF).

Taxonomy-Aware Latent Factor Model

TF tries to directly address two challenges facing LF models; namely, sparsity and cold start. The sparsity problem often manifests in e-retail, where the number of products is very large and each customer purchases a only a few of them. This prevents a simple model, for example IMF, from learning the features that are requisite for effective recommendation (Kanagal et al., 2012). The cold start problem is also a feature of the e-retail setting, where new items are continuously released.

Kanagal et al. (2012) attempt to resolve the sparsity and cold-start problems by including the existing taxonomy of products in their model. Typically, products in an e-retail catalogue are organised in a category tree, or taxonomy. Take, for example, an iPhone; the parent category of an iPhone could be “Smart Phones”, above that “Phones”, and above that “Electronics”, which might be a child of the root category.

TF exploits the taxonomy by introducing factors for every node, in addition to the usual product and customer factors. The algorithm learns product factors such that sibling products have similar factors to their parent. This allows products with very few customer interactions to benefit from siblings with larger numbers of interactions.

In order to model taxonomy, Kanagal et al. (2012) introduce a constraint on the product factors. Namely, a product factor is effectively the sum of itself, and all of the nodes above it

in the taxonomy. To formalise this notion, we introduce some additional notation;

- $\vec{g_i}$ is the effective latent factor corresponding to product i,
- D is the number of levels in the taxonomy, and
- $p_m(i)$ is the mth node above i in the taxonomy. For example, $p_0(i) = i$.

Then, the effective product factors are defined as:

$$\vec{g_i} = \sum_{m=0}^{D} \vec{w}_{p_m(i)}.$$

The scoring function for TF uses the effective product factors instead of the actual product factors:

$$\hat{y}_u(i) = \vec{x_u}^T \vec{g_i}.$$

TF uses the BPR criterion for optimisation. The update equation defined in equation (2.9) expands to:

$$\begin{aligned}
\vec{x_u} &\leftarrow \vec{x_u} - \eta \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{g_i} - \vec{x_u}^T \vec{g_j} \right) \right] (\vec{g_i} - \vec{g_j}) + \lambda \vec{x_u} \right) \\
\vec{w}_{p_m(i)} &\leftarrow \vec{w}_{p_m(i)} - \eta \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{g_i} - \vec{x_u}^T \vec{g_j} \right) \right] \vec{v_u} + \lambda \vec{g_i} \right) \\
\vec{w}_{p_m(j)} &\leftarrow \vec{w}_{p_m(j)} - \eta \left(\left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{g_j} \right) \right] \vec{v_u} + \lambda \vec{g_j} \right).
\end{aligned}$$

Algorithm 3 shows the full algorithm for TF. Note the similarity between this algorithm and algorithm 2. One notable difference is that for every (u, i, j) tripled sampled, we iterate through the ancestors of i and j, updating each of their factors. As a result, the training process for TF is computationally more onerous than BPR.

Algorithm 3 TF procedure from Kanagal et al. (2012)

Input: Initial estimate of $\{X, W\}$
Output: Stable estimate of $\{X, W\}$

1: **repeat**
2: draw (u, i, j) from D_S
3: $\vec{x_u} \leftarrow \vec{x_u} - \eta \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{g_i} - \vec{x_u}^T \vec{g_j} \right) \right] (\vec{g_i} - \vec{g_j}) + \lambda \vec{x_u} \right)$
4: **for** m `in` $0 \ldots D$ **do**
5: $\vec{w}_{p_m(i)} \leftarrow \vec{w}_{p_m(i)} - \eta \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{g_i} - \vec{x_u}^T \vec{g_j} \right) \right] \vec{v_u} + \lambda \vec{g_i} \right)$
6: $\vec{w}_{p_m(j)} \leftarrow \vec{w}_{p_m(j)} - \eta \left(\left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{g_j} \right) \right] \vec{v_u} + \lambda \vec{g_j} \right)$
7: **end for**
8: **until** convergence

Enforcing consistency over the taxonomy ameliorates the cold-start problem. New products

are initially assigned the factors of their immediate parents. As a result, they are available for recommendations, even though they have not yet been purchased by any customers.

Kanagal et al. (2012) find that TF outperforms BPR on an e-retail dataset. TF results in a better AUC than BPR.

A few extensions have been proposed for TF. In addition to modelling product factors over the taxonomy, (Ahmed et al., 2013) model customer preferences for additional product attributes over the taxonomy, including price and brand, using a hierarchical Bayesian approach. (Zhang et al., 2014) take it a step further by learning the category tree itself through a nested Chinese Restaurant Process (nCRP) (Blei et al., 2010).

There are two considerations for the design of a RS that arise from our discussion in this section. Firstly, the diversity of the available approaches dictates that the effective RS provides extensible software interfaces for implementing new methods.

Secondly, we must carefully consider the impact of advanced models on the model serving process. In this section, we have described models that learn factor representations on the level of individual customers and products. As a result, the parameter space of these models is large. In fact, on platforms that service many customers with catalogue sizes in the tens of millions of products, the parameters of these models are unlikely to fit into the memory of a single machine. In addition, calculating a ranking for a customer involves a multiplication of a large matrix of product factors with the customer factor. This is a computationally daunting task. As a result, for scalable serving of recommendations, RS's need to leverage distributed computational and data storage resources.

In the next section, we describe the methods available to the RS for evaluating recommendations.

2.8 Evaluating Recommendations

We have shown above that there are a number of techniques for generating recommendations available to the designer of the RS. How, then, do we choose one technique over another? In this section, we attempt to answer that question. Specifically, we look at two approaches that have been proposed in the literature for solving this problem. The first is offline evaluation. In this method, a number of models are compared in a process using a set of evaluation metrics in an offline setting. The second is online evaluation. In this setting, groups of actual customers are exposed to recommendations generated from different methods. In the context of e-retail, customer experiments measure how different recommendation models influence customer purchasing behaviour (Shani and Gunawardana, 2011). For example, if customers presented with

recommendations from one model buy more products than customers presented with recommendations from another model, then we can conclude that one model is better than another, all else being equal. Thus, the online experiment typically provides a stronger indication of true effectiveness of recommendations than an offline evaluation (Picault et al., 2011).

2.8.1 Offline Experiments

We argue that there are two scenarios in which offline experiments are useful. First, during the search for a good set of parameters for a model, and second in filtering out inappropriate models. In the first case, the goal is to find a set of set of parameters that provide reasonable performance for a model. This process is known in the literature model cross-validation (Demšar, 2006).

In the second case, the goal is to find a small set of candidate algorithms which are "good enough" to test online. As an example, consider the scenario where a new recommendation algorithm is developed and needs to be compared to an existing "state-of-the-art" algorithm. The first step is to run an extensive offline comparison of the state-of-the-art algorithm to the new model. If the new model performs considerably worse than the state-of-the-art, it is probably not worth pursuing with an online experiment (Shani and Gunawardana, 2011).

Data Splitting

Shani and Gunawardana (2011) have given a comprehensive review of the offline evaluation process of recommendation algorithms. We start with a log of transactions, which might be of the format described in section 2.5. This data is then split into three distinct sets, namely, a "training" dataset, a "validation" dataset and a "test" dataset.

The training dataset is used to build the recommendations according to whatever model is under test. We have described various training algorithms in section 2.7.

The validation dataset is used to find a good set of hyperparameters for the model using cross-validation. There are a few methods for performing cross-validation. We describe the method we have implemented for our prototype. For every candidate set of parameters that the data scientist wants to test, a model is trained. Each of these models are then measured against the validation dataset. For every customer in the validation set, we generate the ranking $\hat{r_u}$ using the model, and compare that ranking to the actual customer interactions in the validation set. The model that generates the most optimal ranking is selected to move forward to the next stage.

The purpose of the test set is to measure how well models that performed optimally in the cross-validation stage generalise. For example, consider the situation where we have tuned the

parameters of a BPR and an IMF model, and we would like to reason about which of these models is better. Evaluating a model with a validation or test dataset (also known as a hold-out dataset) gives us an estimate of the actual predictive power of the model. When reporting estimates, however, we cannot claim that the model with the lowest error on a validation set is the optimal model. Since we have selected the models that perform most optimally on the validation set, we have in a sense "fit" the model to the validation set. Thus, these results tend to overestimate the actual performance of the model. As a result, we report estimates of the model performance on an independent test set.

Shani and Gunawardana (2011) specify a number of protocols that can be used to split a data set into training, validation and test sets.

- Sample a validation time and a test time. Place all transactions prior to the validation time into the training set, all transactions that occur between the validation and test time into the validation set, and all the transactions that occur after the test time into the test set. This simulates a situation which is similar to "real-life", where a recommendation algorithm is built at a specific point in time and is required to predict future transactions.
- Another approach is for each customer, to randomly sample a validation time and a test time. Then split the transactions as above, but on a per-customer basis. This protocol does not maintain time consistency across customers, and assumes that the sequence in which products are interacted with are important, not the absolute time at which interactions are made.
- Alternatively, we could ignore time altogether. In this approach, for each customer, we randomly sample a number of products to be placed into the training set, validation set, and test set. This assumes that the temporal aspect of customer interactions is unimportant.

An important feature that should be supported by a RS arises from this discussion – that of "model management" Crankshaw et al. (2014). The goal of model management is to ensure that models used to serve recommendations to customers are performing optimally. There are two aspects to model management; first, maintaining model "freshness", and second, maintaining model effectiveness.

Model freshness refers to recency of the information with which a model has been trained. Customers expect that their needs are adapted to in real-time (Huang et al., 2015). As a result, the RS respond quickly to new information generated through customer behaviour. Ideally, the system updates models incrementally, as new data arrives. Alternatively, the system should provide an automatic mechanism which periodically rebuilds models either after a specified time period, or as performance of the model falls below a threshold.

To maintain model effectiveness, the RS should provide "debuggable" cross-validation procedures. These procedures automatically perform cross-validation and select the best set of parameters for a model. Debuggable refers to the ability to inspect the system's choice of one set of parameters over another. For example, the system could generate a series of plots that describe the performance of the model using some evaluation metric. The data scientist is then able to see exactly why one model has been chosen over another. In addition, the system should automatically, and continually, test different types of models against one another, and make experiment reports available to the data scientist. This would greatly assist in the decision to move ahead with testing a new model in an online experiment. Within this framework for automatic cross-validation and model testing, the RS should also provide extensible software interfaces for implementing different model splitting strategies; and, as we see in the following section, extensible software interfaces for implementing additional evaluation metrics.

2.8.2 Evaluation Metrics

In the previous section, we described the procedure for performing offline experiments. In this section we consider some of the evaluation metrics we can use to compare models during cross-validation and testing. Table 2.2 shows the possible outcomes of our predictions for products in the hold-out dataset. A True Positive (TP) occurs when we recommend a product that the customer ends up buying. A False Negative (FN) occurs when we fail to recommend a product that the customer ends up buying. A False Positive (FP) occurs when we recommend a product that the customer has not bought, and a True Negative (TN) occurs when we do not recommend a product that the customer does not end up buying.

	Recommended	Not Recommended
Bought	True Positive (TP)	False Negative (FN)
Not Bought	False Positive (FP)	True Negative (TN)

Table 2.2: Possible outcomes when recommending a list of products.

Shani and Gunawardana (2011) note that in this prediction scheme, we are assuming that products not bought by a customer are uninteresting to them. This assumption is almost certainly false; the set of products not bought by a customer usually contains products that may be of interest to the customer, but the customers is unaware of their existence. The impact of this assumption is that the number of recommended products that are not of interest to the customer is over-estimated, i.e., the number of FPs.

There are a number of useful quantities which we can compute using the values in table 2.2. Consider the situation in which we generate a personalised ranking for a customer. We then select the first n products into a list and compare the list to hold-out products for the customer.

Three important quantities arise from this situation, namely, precision, recall and false positive rate (FPR).

Precision is defined as the fraction of relevant products within the list. More formally:

$$\text{precision}\,(n) = \frac{TP}{TP + FP}.$$

Note that $TP+FP = n$, the length of the list. Ideally, we would like this quantity to be as close to one as possible. In this case, relevant products occupy the first n positions of the ranking. For example, say we have five products in the hold-out set, and $n = 4$. Ideally, the algorithm generates a ranking where the first four positions of the list are hold-out products. This would result in a precision of one.

To calculate recall, or true positive rate (TPR), we calculate the fraction of all hold-out products in the list. TPR is defined as:

$$\text{TPR}\,(n) = \frac{TP}{TP + FN},$$

where $TP + FN$ is the total number of products in the hold-out set. Ideally, TPR is also close to one. We would like as many of the available hold-out products to be in the list as possible.

FPR describes the proportion of all non-relevant products present in the list. It is defined as:

$$\text{FPR}\,(n) = \frac{FP}{FP + TN},$$

where $FP + TN$ is the number of products not in the hold-out dataset. We want the FPR to be close to zero. In this situation, the number of unbought products in the list is as small as possible.

In practice, there is trade-off between these quantities (Shani and Gunawardana, 2011). For example, to improve TPR, we could increase n. As an increasing number of products are incorporated into the list, TPR approaches one. However, this also increase FPR and reduces precision, as the number of irrelevant products included in the list also increases. Thus, for a given recommendation algorithm, it is useful to explore how these quantities trade-off against one another at varying values of n.

Area Under the Receiver Operative Characteristic Curve

Often, n is not decided upon beforehand (Shani and Gunawardana, 2011). We might, therefore, like to get an understanding of how an algorithm performs at varying values of n. The primary means of doing this comparison is to compute curves that compare precision to TPR, or TPR to FPR. The latter is called the Receiver Operating Characteristic (ROC) curve, and is commonly

used in information retrieval tasks. Figure 2.6 gives an example of this curve. The area under this curve is known as the Area Under the ROC Curve (AUC), and is coloured blue in figure 2.6. It has the useful property that it represents the probability of a randomly chosen hold-out product ranking above any other product.

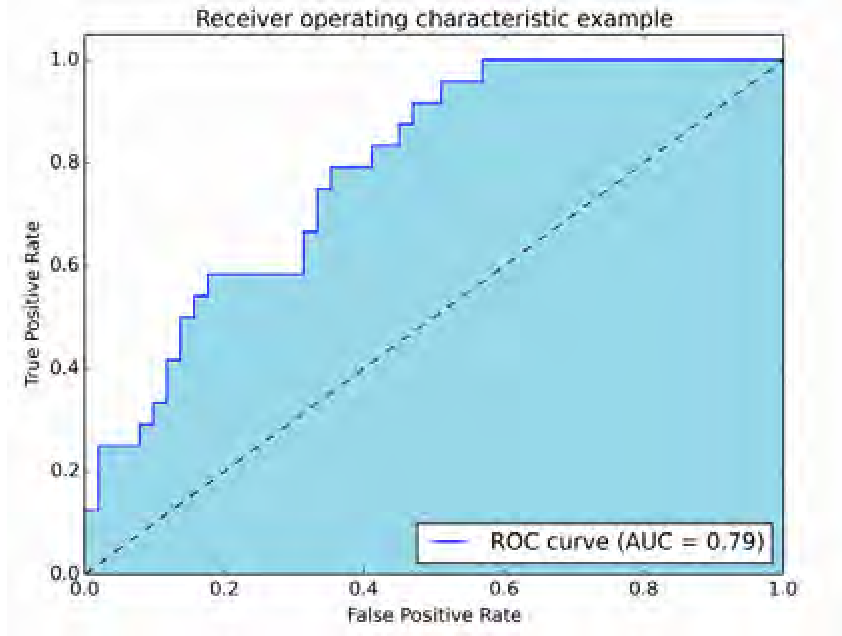


Figure 2.6: Example ROC curve (source: Dernoncourt (2015)).

To get an intuition of why this is true, we introduce some additional notation:

- Let S_{eval} be the evaluation dataset defined by some data splitting protocol. This could be either a validation dataset or test dataset.
- Then, let E^u be the set of products that appear in the evaluation dataset for customer u:

$$E^u \coloneqq \{i \in I : (u, i) \in S_{\text{eval}}\} .$$

  Further, in this section, we reserve special indexing letters i and k for products that appear in E^u, and j for products that appear in $I \setminus E^u$.

Recall that we have N products in total. Now consider that we arrange these products according to a ranking $\hat{r}_u$ generated for a customer u, from 1 to N. Ideally, we would like all the i's to be near the head of this arrangement, as these are the products we hope the recommendation algorithm will suggest to u. As the length of the recommendation list, n, varies from 1 to N, we get that the number of i's in the recommendation list approaches $|E^u|$. This corresponds to moving from the bottom-left to the top-right of the ROC curve. Moving the value of n around traces out the curve. Thus, TPR and FPR are a function of n. The graph moves either a point upward or a point to the right depending on whether the product incorporated next by a change

in n is an i or j, respectively. The amount by which it moves up will be $\frac{1}{|E^u|}$ since there are $|E^u|$ i's, and the amount by which it moves right will be $\frac{1}{N-|E^u|}$, since there are $N - |E^u|$ possible j's.

Now, consider picking at random a j from the arrangement above. Note that a particular j is chosen with probability $\frac{1}{N-|E^u|}$. The proportion of i's above this j in the list is the TPR at the value of n represented by the index of this j in the arrangement, or $\hat{r}_u(j)$. This is also the probability of a random i being above this j. Thus, the overall probability of a random i ranking above a random j is $\frac{1}{N-|E^u|}$ times the TPR at $\hat{r}_u(j)$, summed over all possible j's. This, in turn, is just the width of one "step" on the x-axis of the graph, multiplied by the TPR at that point. Summing over all steps on the x-axis, we get the area under the ROC curve. More formally:

$$\begin{aligned}\mathrm{AUC}_u &= \frac{1}{N - |E^u|} \sum_{j \in I \setminus E^u} \mathrm{TPR}\left(\hat{r}_u(j)\right) \\ &= \frac{1}{N - |E^u|} \sum_{j \in I \setminus E^u} \frac{\mathrm{TP}\left(\hat{r}_u(j)\right)}{|E^u|}.\end{aligned}$$

$TP\left(\hat{r}_u(j)\right)$ is just the number of i's that rank above j:

$$TP\left(\hat{r}_u(j)\right) = \sum_{i \in E^u} \delta\left(\hat{r}_u(i) < \hat{r}_u(j)\right),$$

where δ is the indicator function:

$$\delta(x) = \begin{cases} 1 & \text{if } x \text{ is true,} \\ 0 & \text{else.} \end{cases}$$

From equation (2.1), $\hat{r}_u(i) < \hat{r}_u(j)$ if $\hat{y}_u(i) > \hat{y}_u(j)$. Therefore,

$$TP\left(\hat{r}_u(j)\right) = \sum_{i \in E^u} \delta\left(\hat{y}_u(i) > \hat{y}_u(j)\right),$$

and AUC_u becomes:

$$\mathrm{AUC}_u = \frac{1}{|E^u|\left(N - |E^u|\right)} \sum_{j \in I \setminus E^u} \sum_{i \in E^u} \delta\left(\hat{y}_u(i) > \hat{y}_u(j)\right). \tag{2.10}$$

The AUC is used extensively in the literature to compare the performance of recommendation models. Rendle et al. (2009) use it as the basis for their BPR optimisation criteria. Kanagal et al. (2012), Ahmed et al. (2013) and Zhang et al. (2014) also use the AUC metric to evaluate their approaches. In these papers, the authors choose a representation of the AUC that is averaged over all customers. In this scheme, the AUC is computed for each customer, then averaged to

get a final “average” AUC value, which is used to compare recommendation algorithms.

Mean Reciprocal Rank

Another metric used in evaluating recommendation algorithms is the Mean Reciprocal Rank, or MRR. Shi et al. (2012b) use it to develop their collaborative less-is-more filtering (CLiMF) algorithm. It arises from the reciprocal rank (RR), which measures how early the first hold-out product occurs in a ranking. The MRR is then the average of the RR across all customers. The RR for a single customer is defined as:

$$RR_u = \sum_{i \in E^u} \frac{1}{\hat{r}_u(i)} \prod_{k \in E^u} \left[1 - \delta\left(\hat{r}_u(k) < \hat{r}_u(i)\right)\right] \tag{2.11}$$

In other words, equation (2.11) is one over the rank of the top-ranking product from the hold-out set. The MRR is useful in scenarios where customers are provided with a few but valuable recommendations, which includes some of the e-retail use-cases described in section 2.4.

Apart from AUC and MRR, there are numerous other metrics used to evaluate the accuracy of recommendations, including, Mean Average Precision (MAP) (Shi et al., 2012a) and Normalised Discounted Cumulative Gain (NDCG) (Shani and Gunawardana, 2011). Thus, the effective RS should provide, as part of its model management component, extensible software interfaces with which the data scientist can implement different evaluation strategies.

2.8.3 Online Experiments

The evaluation methodology that provides the strongest evidence as to the true effectiveness is online evaluation. According to Shani and Gunawardana (2011) many real-world RS’s employ an online testing system for comparing recommendation algorithms based on their performance with real customers. In these systems, a small percentage of traffic is directed to different recommendation algorithms. The customers’ interactions with the recommendations are then recorded. These interactions are used to compute which algorithm best optimised real business metrics. For example, in e-retail, one might choose the recommendation algorithm that leads to the highest number of purchases.

Shani and Gunawardana (2011) mention a few points to consider when running online experiments. First, it is important that customers are randomly sampled for redirection to a specific algorithm, so that comparisons between alternatives are fair. Second, more than one “thing” should not be under test at the same time. For example changes to the way in which recommendations are presented to the customer (the UI) should not be tested at the same time as the underlying algorithms. The reason for this is that we may not be able to tease out effects

of the new UI from effects of the new algorithm. Thus, when testing one of the elements that comprise the recommendation experience, the others should remain fixed.

It is preferable to run the online experiment after an extensive offline study has indicated that a new recommendation approach shows promise. The main reason for this is that online experiments can be risky. For example, a recommendation algorithm that provides poor recommendations may cause customers who are shown these recommendations to become despondent with the platform as a whole. As a result, the experiment could have a negative effect on the business. Thus, it is best to run the evaluation knowing that the new recommendation algorithm has potential to improve on the current algorithms.

Although online testing is a critical part of evaluating recommendation algorithms, the testing systems need not be part of a RS. Many large web companies have implemented separate systems for performing online experiments, including LinkedIn (Xu et al., 2015), Google (Tang et al., 2010) and Microsoft (Kohavi et al., 2013). The use-cases for experiment systems extend beyond recommendations. For example, an e-retail platform might want to experiment with the colour of their "Add to cart" button by asking the question, "does a green button improve add to cart conversion?" This is a toy example and the likely answer is no, but it demonstrates the utility of experimentation beyond recommendations.

In the next section, we summarise the characteristics that we have developed throughout this chapter, and discuss how three RS's described in the literature satisfy these characteristics.

2.9 Software Architecture

In the previous sections, we have developed the characteristics necessary for an effective RS. These characteristics are summarised as follows:

- Accuracy; recommendations should be useful to the customer (section 2.4).
- Low latency; customer requests should be served within the window of interactivity (section 2.4).
- Durability; the system should be robust to internal failures (section 2.4).
- Scalability; the system should exhibit graceful scaling properties in the face of increasingly large input datasets and model parameter sets (section 2.5 and section 2.7).
- Model management; models should remain up to date and effective (section 2.8).
- Extensibility; software interfaces that allow the data scientist to implement new techniques are an important feature of a RS. In particular, extensible interfaces should be available for; model building, data splitting, and calculating and summarising evaluation metrics.

In order to understand the core software components of a RS that enable the above characteristics, we look at three RS's described in literature and implemented in practice.

First, we describe Oryx[8], an open source machine learning platform based on the lambda architecture[9]. In this system, models are periodically rebuilt using a scalable computation engine, Apache Spark (Zaharia et al., 2012), from a master dataset stored in a distributed filesystem called the Hadoop Distributed File System (HDFS) (Shvachko et al., 2010). Models are incrementally updated as new customer interaction data is generated by the application. A distributed message bus called Apache Kafka (Kreps et al., 2011) passes data and model updates around the system. A separate application subscribes to model updates, and stores models in-memory for serving recommendation requests.

Second, we describe Velox (Crankshaw et al., 2014), also an open source machine learning platform with recommender capabilities. Like Oryx, this system periodically rebuilds models in their entirety and incrementally updates models as more information arrives. Velox distinguishes itself through advanced model management methodology, and scalable model serving application.

Finally, we describe TencentRec (Huang et al., 2015), a system that relies completely on incremental model updates. In this system, there is no batch processing capability; models are built completely in a streaming fashion.

2.9.1 Oryx

Oryx is a realisation of a the lambda architecture for large scale machine learning (ML). It has been built as a general framework for ML applications, but includes an application for CF. We focus on this part of the Oryx framework. Oryx contains the three side-by-side "layers" of the lambda architecture (figure 2.7):

1. Batch layer; recomputes models as a function of all historical data. This is a long running operation, which by default runs every few hours.
2. Speed layer; computes and produces incremental model updates as a function of the current model and a stream of new customer interactions. These incremental updates are intended to be "best effort" updates to the latest model. Updates happen on the order of seconds.
3. Serving layer; receives models and model updates, and implements an Application Programming Interface (API) which exposes recommendations from the model.

[8] `http://oryx.io/`
[9] `http://lambda-architecture.net/`

In addition, the Oryx layers communicate over a transport layer, which is implemented by Apache Kafka, a distributed publish-subscribe messaging system.

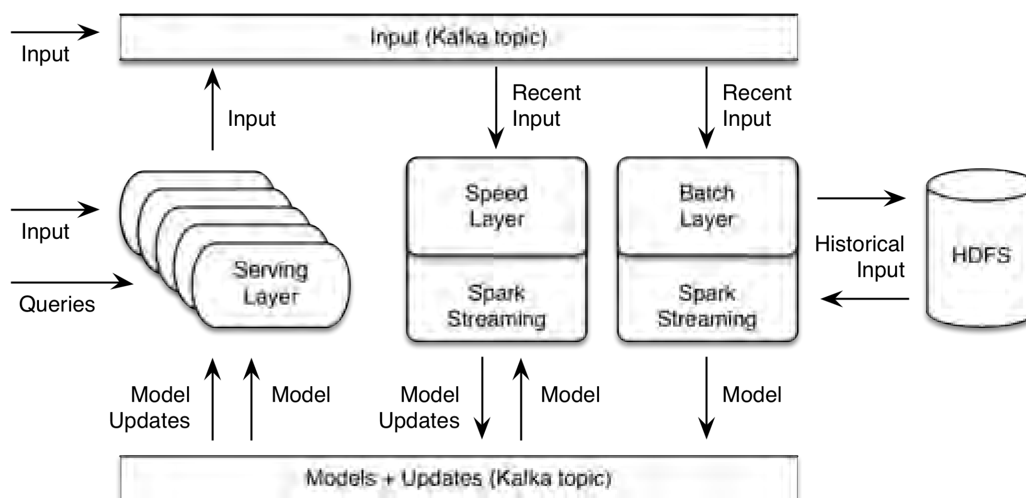


Figure 2.7: Oryx architecture (source: Owen (2015)).

The backbone of Oryx is a scalable, fault tolerant messaging system. Apache Kafka is a distributed commit log designed for publish-subscribe messaging. Kafka maintains feeds of messages in categories called "topics". "Producers" send messages to topics, and "consumers" subscribe to topics and process the feed of published message. Kafka is run as a cluster of one or more servers called "brokers". Kafka has scalability and fault tolerance built-in; a topic is separated into logs called "partitions" that are distributed amongst the brokers. Each partition is replicated and has a leader broker. Producers send messages to the leader of a partition. When a failure occurs and a partition leader is lost, a new leader is elected for that partition from one of the replicas.

As shown in figure 2.7, Oryx ingests data through the "Input" Kafka topic. Users have two options for adding data to this topic. They can either push data directly to the topic using a Kafka client, or they can send data to the `ingest` API endpoint in the serving layer. The batch layer continuously reads data off the input topic, and appends the data to a historical data set stored in HDFS. We describe HDFS in detail in section 3.3. There is no danger of data loss in Oryx because underlying data storage systems (HDFS and Kafka) are fault tolerant.

For scalable and fault tolerant computation of models and model increments in the batch and speed layers, Oryx uses Apache Spark. Oryx uses a specific component of Spark called Spark Streaming. Spark Streaming receives live input data from Kafka and divides the data into batches, which are then processed by the Spark engine to generate a final stream of results in batches (Zaharia et al., 2013). Spark Streaming applications are long-running, and typically collect and process batches of data in short loops of a few seconds. This is the case for the speed layer of Oryx, where small model updates are performed. For the batch layer, where

entire model rebuilds are performed, the period of the Spark Streaming application is set set to a few hours. Spark is described in more detail in section 3.4.

Oryx comes with built-in support for the IMF model we have described in section 2.7. In fact, in the batch layer, Oryx uses the Spark implementation of IMF, which comes packaged with Spark. This implementation of IMF in Spark is described in detail in section 3.8.1. For model updates in the speed layer, Oryx implements a custom update function that approximates updates to the customer and product factors.

Oryx implements basic model management in the form of automatic cross-validation. A user specifies in the configuration different hyperparameter values to try. When the batch retraining phase starts, Oryx builds a model on a training data set for each combination of parameter values and chooses the model with the lowest prediction error on a test data set. Training is done over all historical data, plus a configurable proportion of new data not seen in the last batch retraining phase. Testing is done over the remainder of the new data. Oryx uses the AUC to determine which model performs best. The best performing model is then published to the Kafka “Models + Update” topic in figure 2.7.

Transient failures in the computation are transparently handled by Spark. If the batch layer has a catastrophic crash, it can simply be restarted and it will continue processing the data in Kafka from where it left off. This gives the batch layer a simple failure recovery mechanism.

The speed layer also has a graceful failure recovery mechanism. When the speed layer starts, it begins reading the update topic for all models and model update messages. As soon as it has a valid model in-memory, it publishes model updates to Kafka. Since it also uses Spark for computation, transient failures in computation are handled by Spark. If the entire process dies and is then restarted, it will begin reading from the latest position on both the input topic and model topic. This means data that arrives before the speed layer has a valid model in-memory is ignored. Because the role of the speed layer is to provide an approximate update to the latest model, this behaviour is not unexpected. Due to the simplicity of this solution, it is desirable.

The serving layer in Oryx is stateless. On startup, it reads all models and model updates from the update topic, and begins responding to requests for recommendations as soon as a model has been stored in its memory. Although this allows recommendations to be served from models that are out-of-date, the serving layer will catch up quickly to the latest model. The simplicity of this scheme allows the serving layer to be scaled outward if load on the RS is high.

The serving of LF models involves operations on a large matrix, and is therefore challenging to scale. To alleviate this pressure, Oryx uses Locality Sensitive Hashing (LSH) to speed up the search for recommendations. It does this by pruning the number of candidate product factors according to some sample rate. As product features are created or updated, the serving layer hashes them into partitions. When a request for a recommendation is received, Oryx only

considers products from a sample of the partitions that is less than, or equal to, the specified sample rate. Then, all the products in each of the candidate partitions are ranked according to $\vec{x}_u^T \vec{w}_i$. This computation is done in parallel by a set of threads. Finally, the rankings from each partition are merged to get a list of recommended products. The number of partitions is chosen by Oryx so that the cores available to the serving layer are completely utilised in the computation of the recommendations from each of the partitions (in fact, Oryx allows the number of considered partitions to exceed the number of cores by one). For example, if there are three cores available to the serving layer, and we wish to only consider 75% of the products, then Oryx would hash the product factors into four partitions, and consider three partitions for recommendations. This scheme decreases processing time roughly linearly; a sample of 75% gives a speedup of 1.33× (Owen, 2015). Thus, greater speedup can be achieved through lower sample rates. However, setting sample rates too low adversely affects recommendation quality.

2.9.2 Velox

In the RS research community, there has been very little work on how recommendation models are actually deployed, served, and managed. Crankshaw et al. (2014) have attempted to fill this gap with Velox. Velox provides a framework for applications that allow users to access large ML models at low latency. In Crankshaw et al. (2014), their specific focus is the IMF model provided by Spark. Velox achieves this with two primary architectural components, shown in figure 2.8. First, the model manager orchestrates the computation and maintenance of the model. Second, the model predictor implements a low-latency prediction interface for users of the RS.

In order to ingest data into the system, Velox exposes an `observe` API call (see figure 2.8). To insert data into Velox, an application calls observe, providing a customer ID, product ID and some level of interaction. In order to use the observation for offline model retraining, it is written to Velox's data store, which by default is Tachyon (Li et al., 2014a). Tachyon is an open-source, fault-tolerant, memory-optimised distributed storage system often used as a backing store for Apache Spark. In addition, the observation is used to trigger an online update of the recommendation model. This simultaneously supports low latency learning and personalisation of sophisticated models.

The offline model retraining phase leverages the batch computation capabilities of Apache Spark. In this phase, both customer factors and product factors are recomputed from scratch using the built-in Spark IMF algorithm. Velox passes all available historical data to the model, and the result is a new set of product and customer factors.

The online learning phase runs continuously to update only the customer factors. As observations arrive from the `observe` interface, Velox updates customer factors according to a

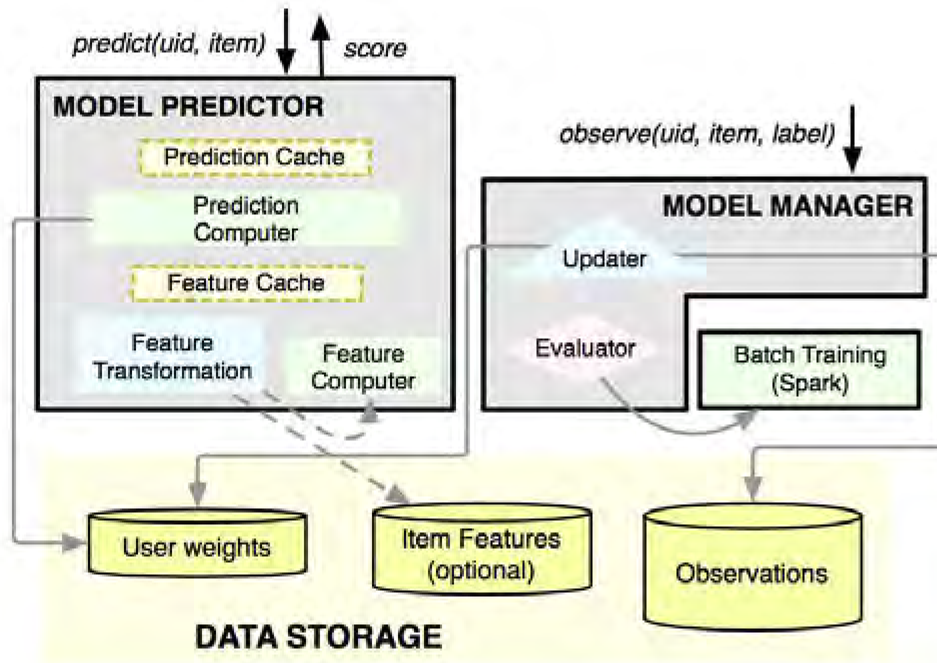


Figure 2.8: Velox architecture (source (Crankshaw et al., 2014)).

user-specified error function. By updating the customer factors online and the product factors offline, Velox provides an approximation to retraining the entire model for every observation. This introduces potential inaccuracy into the model, as product factors should also change in the presence of new observations. Crankshaw et al. (2014) find that although complete retraining has a larger impact on model performance, incrementally updating only customer factors still yields significant benefit in maintaining model performance. Thus, incremental model updates in combination with an error-based model maintenance strategy, which we explain next, ensures that Velox models are always performant.

Model maintenance is an explicit concern in Velox. Velox will perform model retraining when performance degrades below a certain configurable threshold. To assess model performance, Velox maintains per-customer aggregates of errors associated with a model. In addition, when retraining a model using Spark, Velox runs a cross-validation step to assess the generalisation performance of the model. This allows Velox to choose an optimal set of parameters. If a performance regression occurs, Velox allows the user to rollback to a previous version of the model while the regression is investigated.

Velox exposes two interfaces for serving recommendations. The first is `predict`, which takes as parameters a customer ID and product ID and returns the point prediction for the level of interaction between the customer and product. Since we are interested in serving lists of recommendations to customers, this interface is not of much use. The second API call, `topK` is more what we are looking for. It takes as input a customer ID and a list of product IDs. It returns the same list of product IDs, but ordered from most relevant product for the given

customer to the least.

To serve recommendations at scale and with low latency, Velox implements two strategies. The first exploits partitioning of customer factors across a Velox cluster. Velox stores customer factors in-memory using Tachyon. Each machine contains a partition of the customer factors. When a request arrives at the prediction service, Velox routes it to the machine that contains that customer's factors. This ensures that lookups of customer factors can be resolved locally, which reduces latency and network load on the cluster. In addition, it provides a natural load balancing mechanism for distributing serving across a Velox cluster. The second strategy involves caching of product factors.

Product factors are also distributed across the cluster. Thus, when making predictions for a given customer, data transfer may need to occur from a remote machine containing the require product features. While the number of products in the system might be large, product popularity is often heavily skewed (Crankshaw et al., 2014), thus many items are not frequently accessed, while a small subset of items are accessed very often. Velox caches these "hot items" on each machine to improve computation.

2.9.3 TencentRec

Huang et al. (2015) take a pure-streaming approach in the design of their RS, TencentRec. Like Oryx and Velox, TencentRec is a platform on which various ML algorithms can be implemented and served. We focus specifically on their implementation of an item-based kNN CF algorithm, which powers recommendations on a few of their online properties, including YiXun[10], a popular Chinese e-retail website.

TencentRec is comprised of three components, as shown in figure 2.9:

1. TDAccess: A Kafka-like publish-subscribe messaging system for ingesting data into TencentRec.
2. TDProcess: A stream-processing application based on the Apache Storm[11] computation engine. This component contains functionality for performing real-time CF.
3. TDStore: A distributed key-value storage engine for storing state required for CF.

These components enable TencentRec to respond to customer interactions and update recommendations in less than one second. In addition, TencentRec operates at massive scale; it deals with 10 billion customer interactions and more than one PB of data each day.

[10] http://www.yixun.com/
[11] http://storm.apache.org/

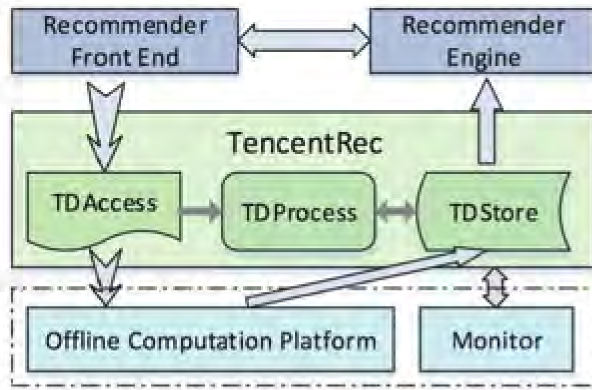


Figure 2.9: TencentRec architecture (source (Huang et al., 2015)).

Huang et al. (2015) have chosen Storm, an alternative to Spark Streaming, for generating their streaming CF model. Storm is an open-source, distributed, real-time computation system for processing streams of data. It operates in clusters of nodes called "Nimbus" nodes and "Supervisor" nodes. The Nimbus node is the cluster manager — it distributes code around the cluster, and assigns work to the supervisor nodes. Supervisor nodes manage "workers", which are processes that do the computational heavy-lifting. The Nimbus and supervisor nodes are fault tolerant in that they are stateless. If they fail, they can simply be restarted without concern over erroneous state. The same applies to the workers, whose computation is assumed to be stateless. This makes it easy to add and remove supervisors to scale the system up and down.

In order to provide scalable and fault tolerant data ingestion, Huang et al. (2015) developed TDAccess. TDAccess is remarkably similar to Apache Kafka, with the same partitioned data model. In TDAccess, producers send data to a cluster of "data servers", who store data from producers in many partitions across the cluster. Consumers pull data in parallel, with the level parallelism being equivalent to the number of partitions.

While TDAccess and Storm provide scalability and fault tolerance for data ingestion and computation, there is a requirement for fault tolerant and scalable state storage. Recommendation algorithms such as kNN CF must keep track of historical customer behaviours in order to compute recommendations. Huang et al. (2015) have developed Tencent Data Store (TDStore) for this purpose. TDStore is a distributed key-value storage system that stores state data used in recommendation computation. TDStore is made up of a cluster of data servers, each of which are assigned a keyspace. Each server is the host of a keyspace and also a backup of another keyspace. Thus, if a particular server fails, its backup becomes the server for its keyspace. In this way, TDStore provides robust support for stateful operations in Storm.

TencentRec implements an adaption of the kNN CF algorithm described in section 2.7 for generating recommendations. Huang et al. (2015) make two adjustments to the algorithm to

support their use-case. First, they adapt the similarity measure to include implicit ratings, and second, they propose an incremental update scheme for item similarity scores. As customer interactions arrive in TDAccess, they flow through the Storm processing pipeline, incrementally updating the relevant similarity statistics stored in TDStore. In order to compute recommendations, applications similarity scores from TDStore and compute recommendations. Thus, the designers of TencentRec have chosen not to implement a serving layer as a part of their RS.

2.10 Summary

e-Retail in SA is a sector with enourmous potential once systemic issues around internet infrastructure have been resolved, and greater levels of trust have been established with potential customers. Key to the development of the sector is the ability of e-retailers to accurately identify customer needs and personalise interaction with the platform. Although e-retail platforms have the advantage of wider choice, allowing customers to effectively navigate an online catalogue is a challenge. Utilising an effective RS alleviates this issue and is in some ways fundamental to the acceptance of the technology by future customers. As a result, an effective RS brings with it significant economic benefits. For example, in 2006, Amazon attributed 35 % of their revenue to sales generated through their RS. In that year, Amazon's revenue was over $10 billion, which means that the RS accounted for around $3 billion, a significant amount of value. Thus, a strong argument can be made for the economic advantages of a RS.

Nevertheless a RS is itself a technical challenge. In this chapter, approaches to building a RS have been outlined and reviewed. In particular, the various options with respect to recommendation techniques, the evaluation of recommendations and software architecture have been summarised. Arising from this discussion have been six characteristics of an effective RS, namely; accuracy, low latency, durability, scalability, model management, and extenisbility. This information has been a critical input in the design and implementation of the prototype, which we describe in the following chapter.

Chapter 3

System Description

Our RS is made up of six frameworks: (1) data ingestion, (2) data preparation, (3) model building, (4) model evaluation, (5) model publishing, and (6) model serving. In this chapter, we attempt to give a detailed description and rationale for each of these frameworks.

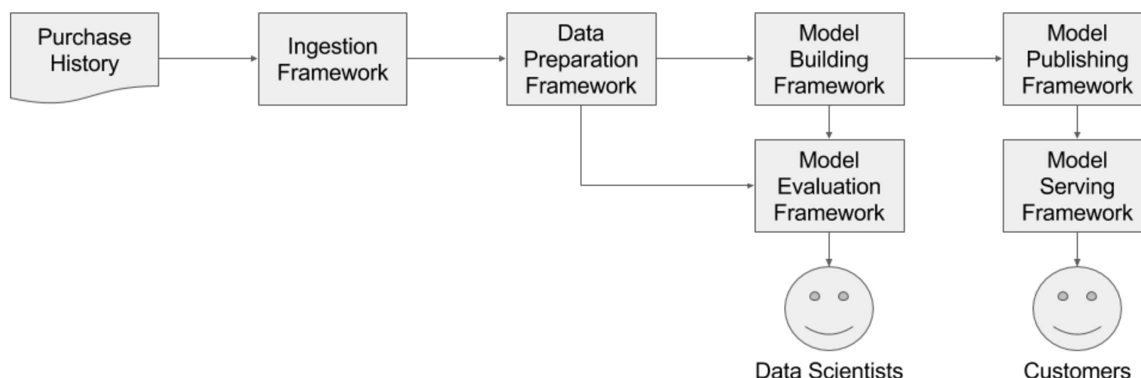


Figure 3.1: Flow of data through the recommender system.

Figure 3.1 shows the flow of data through the system (from left to right). The RS takes as input customer transaction data. In section 3.1 we describe the various characteristics of this input data.

In section 3.2, we describe the outputs of the RS. The primary purpose of a RS is to provide a set of products personalised to the tastes of specific customers. In addition, we have argued that an effective RS should also support the model management activities of the data scientist, i.e. choosing a model with which recommendations are generated. This is usually achieved through the comparison of model performance as measured by some evaluation metric. Thus, in section 3.2, we first describe the format of the recommendations, and second describe the output of model evaluation, which aids model management activities.

In chapter 2, we developed the characteristics of an effective RS. Including the model

management capabilities already mentioned above, they are:

- accuracy,
- low latency,
- scalability,
- durability,
- model management, and
- extensibility.

In our system, we have attempted to meet these requirements through the application of a distributed filesystem, scalable computation engine, and flexible workflow system. In sections 3.3 to 3.5 we explain and motivate the use of the Hadoop Distributed Filesystem (HDFS), Apache Spark and Luigi as the distributed filesystem, computation engine and workflow system, respectively.

From section 3.6, we move on to the implementation details of the various components that comprise the RS. First, we discuss the ingestion framework, which provides a flexible software interface for adding new sources of data. In addition, we detail our specific implementation of the ingestion framework for this research.

Once transaction data has been ingested into the system, we filter and augment it in various ways. Filtering removes bad or unnecessary data, while data augmentation adds fields to the data useful in downstream tasks. One example of a downstream task is splitting data into training, validation and test sets for model building and evaluation. Data filtering, augmentation and splitting fall under the data preparation framework. In section 3.7 we describe how the data preparation framework provides a flexible set of software interfaces for implementing different filtering, augmentation and splitting strategies, and give details about the strategies we have chosen for this dissertation.

Next, the model building framework makes available data from the data preparation framework for building recommendation models. In section 3.8 we describe the software interface provided by this framework. In addition, we present the LF models that we have chosen to implement for this dissertation, namely IMF and BPR.

In section 3.9, we describe the model evaluation framework. The purpose of this framework is to provide software interfaces for model evaluation and consumable summaries of these evaluations (often in the form of visualisations). We focus on our implementation of the AUC for LF models, and our choice of visualisation for that metric.

Once models have been built and evaluated, the most promising models are published to be served to customers. The model publishing framework provides an interface with which programmers can make their models available for serving. In section 3.10, we describe this interface and the model publishing strategy we have employed for the LF models relevant to this dissertation.

Finally, in section 3.11 we present the model serving framework. Here we describe the recommendations service, which allows users of the RS to query the most recent recommendation models for product suggestions for specific customers.

3.1 System Input

The RS assumes that there exists some data store which contains the current state of all transactions on the platform. The format of the transactions is a structured set of records where each record represents a purchase of a single product. A purchase is defined by a set of identifiers, a timestamp and the state. The identifiers specify the customer and product, and the timestamp specifies the time at which the order was placed. The state could conceptually be one of “ordered”, “shipped”, “cancelled”, etc. Table 3.1 gives a few examples of what these transactions could look like.

Customer ID	Product ID	Order Date	State
123456	654321	2015–01–01 08:01:23	ordered
483450	752394	2015–01–01 08:02:20	shipped

Table 3.1: Example of transactions expected by the recommender system.

3.2 System Output

The RS exposes two main sets of outputs intended for two different audiences. The outputs of primary importance are the recommendations, which are lists of products personalised for specific customers. These recommendations are usually requested from some recommendation model. Often, the user of the RS must choose from very many models, and for each model select a promising set of parameters from a large search space. How does the user make this choice? The second set of outputs are intended to be an aid for this decision. More specifically, the second set of outputs are easily digestible summaries of offline model performance.

3.2.1 Product Recommendations

Product recommendations are exposed to users through an application programming interface (API) implemented by the recommendations service. A user of the RS retrieves recommendations by creating an API request that specifies a model and a customer identifier. In response, the RS generates recommendations for the specified customer using the specified model. The format of the response is a list of product identifiers. This list is ranked; the first identifier in the list represents the product that is most relevant, and the last identifier the product that is least relevant to the specified customer. The RS could also respond with an empty list, in which case it has no knowledge of the specified customer. See section 3.11 for a detailed description of the recommendations service.

3.2.2 Model Evaluation

The RS provides templates for various offline model evaluation summaries. In section 2.8, we described some of the offline metrics with which recommendation models can be evaluated. Often, these metrics measure the performance of models on a per-customer basis. The result is a set of very many values that without an appropriate summary are impossible for the data scientist to digest in their entirety. Therefore, the purpose of the model evaluation summaries is to assist the data scientist in digesting comparative metrics to make decisions about which models or model parameters show promise. Visualisations provide useful interfaces which allow the data scientist to explore representations of the distribution of an evaluation metric over the population of customers. In section 3.9.1, we provide specifics about the templates for evaluation metrics and visualisations that we have decided to implement for this dissertation.

3.3 Hadoop Distributed File System

In addition supporting the model management activities of the data scientist, the RS should transparently scale with the size of its input data. For example, consider the situation where an e-retail company is doubling their transaction volume year-on-year for two years — a scenario not uncommon among hyper-growth web-based startup companies (Agrawal, 2015). Consider further the non-optimal, but simple data ingestion strategy of consuming a snapshot of the entire dataset every day for the purposes of model building. Now, assume this trend continues for the foreseeable future. After six months the total amount of data generated just through data ingestion will be approximately 220 times the initial data size. After a year, the storage requirement will have grown to approximately 525 times the size of the initial data; and after two years, 1500 times the initial data size. Traditional databases are not designed to scale with this kind of growth (Bailis, 2015). Systems that can scale with this data are therefore necessary

to support the long-term functioning of the RS. Inspired by its use in Oryx (Owen, 2015), we have chosen HDFS as the scalable storage engine underlying our RS.

HDFS is a Java-based file system that provides fault tolerant and scalable data storage (Hortonworks, 2016). HDFS is designed to run on a cluster of commodity computers. It scales by logically "joining" the storage capacity of each of these computers and presenting to the user a unified access layer. The design of HDFS has been based on a number of assumptions, namely (Hadoop Development Team, 2015):

- hardware failure is expected,
- applications need a write-once-read-many access model for files,
- applications that use HDFS for storage need high throughput access to their datasets,
- applications that run on HDFS have large datasets, and
- computation is much more efficient if it is executed near in network terms to the data on which it operates.

These assumptions allow the design of HDFS to support massive scale, high rates of data throughput and robust fault detection and recovery.

HDFS has a leader/follower architecture. As shown in figure 3.2, an HDFS cluster consists of a single NameNode – a leader node that manages the filesystem namespace (commonly known as the directory structure), and a number of DataNodes – follower nodes which manage attached storage. There are no hard requirements for HDFS node sizes, however, table 3.2 shows the recommended node size for a pilot deployment of HDFS (Hortonworks, 2016). In the enterprise server world these are relatively small machines, also known as commodity machines.

CPU	Memory	Disk
2 × Quad Core	12 to 24 gigabyte (GB)	4–6 drives of 2 terabyte (TB) each

Table 3.2: Typical machine size for a pilot HDFS deployment.

Hardware failures in clusters of commodity computers are bound to occur (Vishwanath and Nagappan, 2010). An HDFS deployment may consist of hundreds or thousands of server machines, each with a non-negligible probability of failure. This means that some component of HDFS is always non-functional. Therefore, detection of faults and quick recovery from them is a core goal of HDFS. To achieve this, a file on HDFS is split into a number of blocks, which are replicated with some user-defined factor and spread amongst the DataNodes (see arrow marked "Replication" in figure 3.2) (Hadoop Development Team, 2015). The NameNode decides on the mapping of blocks to DataNodes, and monitors the number of replicas for each

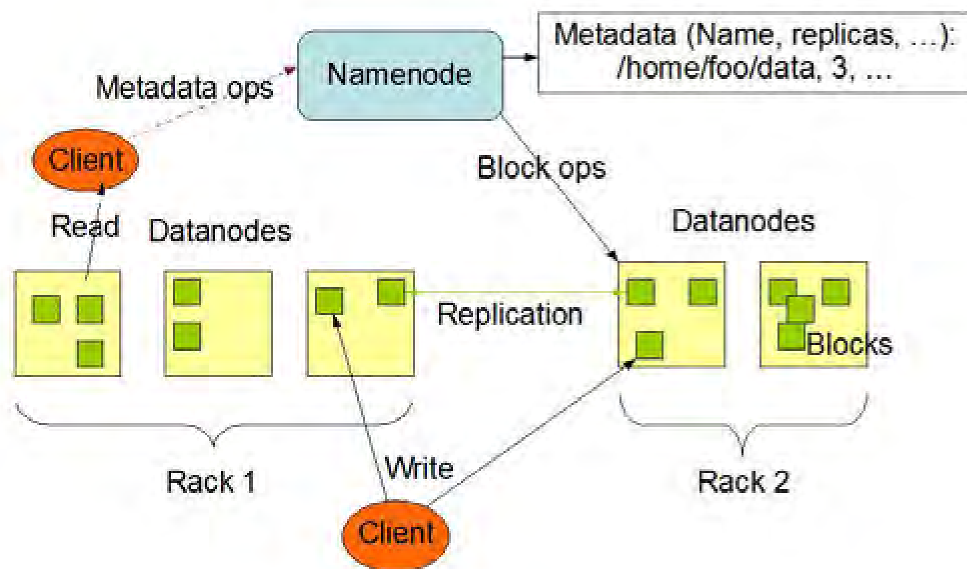


Figure 3.2: HDFS architecture (source: Hadoop Development Team (2015)).

block. When a block is lost due to DataNode failure or disk failure, the NameNode instructs a healthy DataNode to create another replica of the block. In this way, data is stored reliably even in the presence of failure.

A replicated data model introduces data coherency complexity (Bailis, 2015). HDFS gets around this issue by requiring that all blocks be immutable; they are written once and after that cannot be modified (Hadoop Development Team, 2015). Thus, HDFS only has to be concerned about consistency between replicas when a block is created and never again. This also improves read throughput, as there is no need to stall for synchronisation between replicas on block updates. We can be sure that a client reading from any replica in the cluster is getting the most up-to-date information.

HDFS has specifically made the trade-off between low-latency data access and high throughput in favour of high throughput. It has been designed for applications that stream large amounts of data in a batch fashion on a cluster that can scale to hundreds or thousands of nodes. Toward this end, HDFS has been tuned for extremely large datasets, where a typical file is gigabytes to terabytes in size. In order to prevent these massive files from continually shuffling across the network, HDFS is suited for applications that are capable of parallelising execution across the cluster. This model of execution requires a paradigm-shift in the thinking of programmers who move over from traditional modes of data storage. Over the years, one specific paradigm of distributed computing has become dominant in the world of HDFS, namely MapReduce (Dean and Ghemawat, 2004). In fact, MapReduce has been designed by the same group of programmers who maintain HDFS, and many of the features of HDFS have been designed to complement the design of MapReduce.

MapReduce exploits a restricted programming model to automatically parallelise user programs and provide transparent fault-tolerance for those programs. A MapReduce application takes as input a set of key-value pairs, and produces a set of output key-value pairs. The programmer expresses a MapReduce operation through two functions; `map` and `reduce`. The `map` function takes an input pair and produces a set of intermediate key-value pairs. The `reduce` function accepts an intermediate key and set of values for that key, and merges those values to form a new, typically smaller, set of values. The `reduce` function usually produces the final output of the program. For example, this could be the counts of specific words in a very large corpus of documents.

In practice, a MapReduce program proceeds by first partitioning the input data into M partitions. Each of the M partitions are processed in parallel across the cluster by M workers that apply the `map` function to each record in each partition. The intermediate keys are then written into one of R partitions by the map workers. The `reduce` function is then applied to every intermediate key by R reduce workers across the cluster. Each reduce worker produces as output a single file. Thus, after successful completion, the output of the MapReduce program is available in R output files on HDFS.

In the computing environment for which HDFS and MapReduce has been designed, network bandwidth is a relatively scarce resource (Dean and Ghemawat, 2004). MapReduce conserves network bandwidth by taking advantage of the partitioning scheme of HDFS. Typically, a mapper task will be launched for each partition of a file on HDFS, and MapReduce attempts to schedule the mapper worker corresponding to a particular partition on one of the HDFS nodes that contains a replica of that partition. Failing that, it attempts to schedule a mapper on a machine close to a replica of the input data. Dean and Ghemawat (2004) report that when running large MapReduce operations on a significant fraction of workers in a cluster, most of the input data is read locally and consumes no network bandwidth.

The co-located nature of computation and data defines the major difference between MapReduce and traditional computational paradigms. Consider the case where we have a centralised relational database system backing the RS. The database would usually be running on a machine with one large disk and anywhere between four and 16 processors (Turner, 2011). In order to pull data from the system, a programmer would issue a query in Structured Query Language (SQL). Since the query is limited to SQL, and recommendation models are complex enough that they would be difficult to express solely in SQL (Armbrust et al., 2015), this query might get the programmer only some of the way toward building a recommendation model. Next, the programmer can continue processing data in a sequential manner, or, if the programmer finds themselves in a High Performance Computing (HPC) environment, processing can continue in another distributed computing framework like Open MPI[1]. Since HPC setups are usually lim-

[1] `https://www.open-mpi.org/`

ited to research institutions, and we find ourselves in an enterprise environment, the choice an a MapReduce-like computational engine is necessary.

However, MapReduce is not the computational engine we have chosen for our RS. Apache Spark has evolved as an extension of MapReduce, and like Oryx and Velox, this is the engine we choose for our distributed computation.

3.4 Apache Spark

Apache Spark is general-purpose cluster computing engine with libraries for streaming, graph processing and machine learning (Armbrust et al., 2015). It offers a functional programming interface in Java[2], Scala[3], Python[4] and R[5], where users manipulate distributed in-memory collections called resilient distributed datasets (RDDs). RDDs form the basis for a higher-level abstraction called the DataFrame, which allows relational processing over internal Spark RDDs and external data sources. In this section, we show how Spark supports both the machine learning and relational processing activities of the recommender system with RDDs and DataFrames.

Cluster computing on commodity hardware has emerged as one of the most promising paradigms for large-scale data analytics. Early systems such as MapReduce have been widely adopted because they allow users to write parallel computations using a set of high-level operators, without having to worry about distribution and fault-tolerance (Zaharia et al., 2012). Although these early systems provide useful abstractions for a cluster's computational resources, they do not provide abstractions for leveraging distributed memory. This makes them inefficient for analytic applications that reuse intermediate results, a pattern common in many iterative machine learning algorithms. In MapReduce, for example, the only way to iterate between computations is to write intermediate results to an external stable storage system such as HDFS. The cost of this is prohibitive – application execution times become dominated by data replication, data I/O and object serialisation.

Spark has introduced a novel abstraction called resilient distributed datasets (RDDs) that enables data reuse in a broad range of applications. RDDs are fault-tolerant data structures that are distributed across the memory of a cluster. They allow users to explicitly persist intermediate results in memory and manipulate them with a fully-featured API (Zaharia et al., 2012). Each RDD is a collection of Java or Python objects that are partitioned across a cluster. RDDs can be processed through operations such as map, filter and reduce, which take functions as arguments. Spark serialises these functions then ships them to workers across the cluster,

[2] `https://java.com/en/`
[3] `http://www.scala-lang.org/`
[4] `https://www.python.org/`
[5] `https://www.r-project.org/`

who apply them to their entire local partitions (Armbrust et al., 2015). Additionally, these coarse-grained transformations allow an RDD to efficiently provide fault-tolerance by logging its lineage (all the transformations required to build it) rather than replicating the actual data. When a partition is lost due to machine failure or otherwise, the RDD has enough information about how it was created to recompute just that partition. Thus, lost data can quite often be quickly reconstructed, avoiding costly data replication.

As an example of the expressiveness of the Spark RDD API, the example in listing 1 shows how one could perform a word count using Python over a file or set of files stored on HDFS, then save the result back to HDFS. This code snippet is known as a "driver program". As shown in figure 3.3, the driver program connects to a cluster of workers and is responsible for tracking the lineage of RDDs, which are stored as partitions in the RAM of workers.

```
1 from pyspark import SparkContext
2 sc = SparkContext()
3 text_file = sc.textFile("hdfs://...")
4 words = text_file.flatMap(lambda line: line.split())
5 word_count = (words.map(lambda word: (word, 1))
6                    .reduceByKey(lambda a, b: a+b))
7 word_count.saveAsTextFile("hdfs://...")
```

Listing 1: Word count example using the Spark RDD API. Adapted from Zaharia et al. (2012).

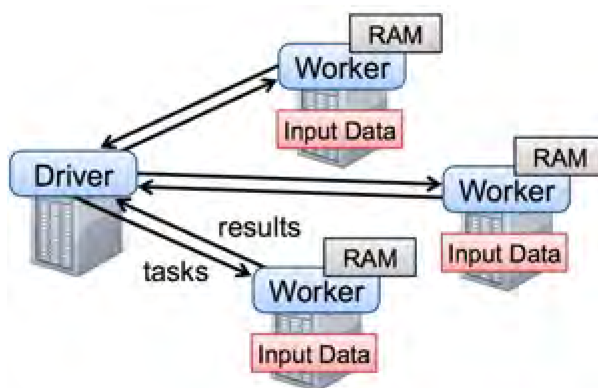


Figure 3.3: Spark runtime architecture (source Zaharia et al. (2012)).

A Spark driver starts by defining one or more RDDs in stable storage. In the case of listing 1, it is a set of files on HDFS. It is useful to deploy Spark workers alongside HDFS DataNodes, as workers can then read data straight from their local disks into memory, without using the network. HDFS also provides a natural partitioning scheme for RDDs; each block of each file on HDFS is mapped to a single RDD partition. Next, the programmer can define a set of "transformations" and "actions" over the RDDs. Transformations are operations that create a new dataset from an existing one. Examples of transformations that appear in listing 1 are `flatMap`, `map` and `reduceByKey`:

- `flatMap` operates here on an RDD whose elements are individual lines of text. The lambda function splits these lines into a list of individual words. Conceptually we now have a collection of lists of words. `flatMap` flattens each of these lists so that the resulting RDD has elements which are individual words.
- `map` applies a function to each element of an RDD. Here, it maps each word of the `words` RDD to a pair of the word and the number 1.
- `reduceByKey` is called on the `(word, 1)` key-pairs and applies the given reduce function to the values of each key-pair. In this case, the values are integers, and the reduce function simply sums them up. Because the initial value for each word is one, this field becomes a counter for its corresponding word.

The resulting RDD contains `(word, number)` pairs, where `number` represents the count of `word` occurring in the original HDFS files.

Our `word_count` RDD is then used in what is known as an "action", which is an operation that either returns a value to the driver program, or saves data to an external storage system. In this case, the action is `saveAsTextFile`, which saves the `(word, number)` RDD to a set of HDFS files. Other examples of actions include `count` (which returns the number of elements in the dataset) and collect (which returns the elements of the dataset to the driver program) (Zaharia et al., 2012).

In order to support iterative analytics (including data exploration and machine learning), Spark exposes the `persist` method to indicate that an RDD is to be reused in future computation. If a programmer calls this method on an RDD, the first time it is computed in an action, it is kept in memory on the workers. Spark keeps persisted RDDs entirely in memory by default, but can spill them to disk if there is not enough available RAM.

An important note about the API is that RDDs are computed lazily. Spark will wait for an action to be performed on an RDD before launching a computation. This allows Spark to perform some simple query optimisation before computation begins, such as pipelining operations (Armbrust et al., 2015). For example, in listing 1, the engine will pipeline reading from the HDFS file with performing the `flatMap` and `map` operations so that the intermediate `words` RDD never gets materialised. Although these optimisations are extremely useful, they are also limited because Spark has no insight into the structure of the data, which are arbitrary Python objects; nor the behaviour of the functions passed into the transformations, which contain arbitrary code (Armbrust et al., 2015). This has been part of the motivation for the creators of Spark to develop the DataFrame API, which we describe next.

The DataFrame API provides two important contributions to the Spark ecosystem. First, it provides a relational interface already familiar to many data analysts. Second, it provides

a powerful optimisation engine called Catalyst, which makes it easy to add new data sources, optimisation rules and data types to the DataFrame API (Armbrust et al., 2015).

Data applications such as RSs require a mixture of processing techniques, data sources and data sinks. Programmers often prefer to write their data transformations in these systems using a declarative language such as SQL, or a domain specific language (DSL) such as the ones provided in R DataFrames (R Core Team, 2013) and Python Pandas (McKinney, 2010). The applications may then demand that this data be loaded into a semi- or unstructured format for further processing or serving to users. Relational queries are often not suited to these situations and custom procedural algorithms are required. In addition, programmers often need to perform advanced processing such as building models using machine learning algorithms, which are difficult to express in relational terms. Thus, data pipelines such as those found in RSs require a mixture of relational queries and procedural algorithms. By adding the DataFrame API to Spark, its creators have given users the ability to freely mix the two paradigms.

In the Spark world, a DataFrame is a collection of rows with the same schema, distributed throughout the Spark cluster. They are built on top of traditional RDDs, but have the additional capability to track their schema and support various relational operations that lead to optimised computation. DataFrames can be constructed from a table in an external data source such as Parquet[6] files on HDFS, or from an existing RDD of Python (or Java or Scala) objects (Armbrust et al., 2015). They can also be written to a variety of data systems once processing has been completed. DataFrames are manipulated with a set of relational operators similar to the R DataFrame API (R Core Team, 2013). These operators include projections with `select`, filters with `where`, joins with `join` and aggregations with grouped data created with `groupBy`. They all take as arguments a limited set of expressions, which allow Spark to capture the structure of the operations. For example, the snippet in listing 2 performs exactly the same word count as in listing 1, except uses the DataFrame API.

```
1 from pyspark import SparkContext
2 sc = SparkContext()
3 sqc = SQLContext(sc)
4 text_file = sc.textFile("hdfs://...")
5 words = text_file.flatMap(lambda line: line.split()).toDF("words")
6 word_count = words.groupBy(words["words"]).count()
7 word_count.write.parquet("hdfs://...")
```

Listing 2: Python word count example using the Spark DataFrames API.

The `words` DataFrame is created with the `toDF` function which maps RDD elements to a DataFrame column with the name `words`. The expression `words["words"]` represents the `words` column of the DataFrame. Column expressions can be additionally manipulated

[6] https://parquet.apache.org/

with operators that return new columns expressions, including; comparison operators such as == for equality testing and > for greater than testing, arithmetic operators such as + and −, and aggregation operators such as `count("words")`. In listing 2, we group by the word and perform a count aggregation over these words. The resulting DataFrame is written to HDFS files in Parquet format with the DataFrame output API. An example of the data that this Parquet file is expected to contain can be seen in table 3.3.

words	count
Spark	10
HDFS	5
Apache	6
⋮	⋮

Table 3.3: Example output DataFrame for the snippet in listing 2.

Like RDDs, DataFrames are lazily evaluated by Spark. Each DataFrame represents a logical plan to achieve some computation, but no computation occurs until the user calls an output operation, such as the `write` operation in listing 2. With DataFrames, Spark has detailed knowledge of the schema of the the DataFrame (each column has a known type, as opposed to arbitrary objects in an RDD); and the operations on these columns are transparent to the engine (only a limited set of operators are supported, as opposed to arbitrary code in RDD transformations). This allows Spark to build up a very detailed set of instructions, which it represents internally as a tree, and pass these instructions to Catalyst for optimisation before computation starts.

Catalyst is Spark's extensible optimiser based on functional programming constructs in the Scala programming language. The purpose of Catalyst is to perform rule- and cost-based optimisation of the computation of a specific DataFrame. The internals of Catalyst are not relevant to this discussion, and we refer the interested reader to chapter 4 of (Armbrust et al., 2015). However, it is worth mentioning that Catalyst allows external developers to extend the optimiser by adding data source specific rules that can push filtering or aggregation into external storage systems that support it. This makes it easy to add data sources and sinks to the DataFrames API. For example, a Java database connector (JDBC) for relation database management systems (RDBMSs) has been added that scans ranges of a table from an RDBMS in parallel and pushes filters into the RDBMS to minimise communication (Armbrust et al., 2015). A data source also exposes network locality information. For HDFS, this allows Spark to optimise which machines read which partitions of the data.

Another significant feature enabled by DataFrames and Catalyst is performance parity between the different language APIs. In the past, there has been a significant performance gap between RDD routines written natively in the Java Virtual Machine (JVM) with Scala and Java

versus the same routines written in Python, with the JVM holding the advantage. The reasons are twofold; first, Spark is written in Scala and therefore applications written in its native language are more efficient, as objects need not be continually passed between the JVM and Python. Second, vanilla Python is known to be slower for performing CPU-bound computations than the JVM (Armbrust et al., 2015). The DataFrame API, and Catalyst in particular, brings equality to the different language APIs. Only the logical plan for the DataFrame is generated in the chosen API language (e.g. Python), after which it is passed to Catalyst which optimises the plan and generates the JVM bytecode used in computation (Armbrust et al., 2015). Furthermore, DataFrames makes it easy for the Spark developers to support their machine learning routines in all languages.

Perhaps the most attractive reason to use Spark in a data system such as a RS is that Spark is supplied with a rich library of machine learning algorithms called MLlib. It includes, amongst others (Spark Development Team, 2016):

- logistic regression and linear support vector machine (SVM),
- linear regression with L_1, L_2 and elastic-net regularisation,
- singular value decomposition (SVD), QR decomposition and principle component analysis,
- random forest and gradient-boosted trees,
- topic modelling via latent Dirichlet allocation (LDA), and most importantly for our use-case,
- recommendations via IMF.

DataFrames have also been integrated with the algorithms of MLlib. This means that a programmer can provide as input to an algorithm a DataFrame. Typically, the output of an algorithm will be a model whose parameters could also be stored in one or more DataFrames. Support for DataFrames makes it easy to perform transformations on a data source, pass that data to an advanced machine learning algorithm, then use that model for prediction on some other data. Apart from DataFrames' convenience for the user, they have also been useful for exposing the algorithms in MLlib to all of Spark's supported languages (Armbrust et al., 2015). In the past, each of the algorithms took a bespoke set of objects as input, for example, labelled points for classification, or triples of customer, product and rating for recommendation. In order to pass data between the runtimes (from Python to Scala, for example) each of these data structures had to be implemented in all the languages. Using DataFrames everywhere means that conversions for types need only occur in the DataFrames API, where they already exists. Therefore, as more algorithms are added, feature parity between programming languages becomes less of an issue when considering Spark as part of an analytics framework.

3.5 Luigi

The activities of a RS can often be broken down into a number of tasks. For example, model building and publishing can be broken down as follows:

1. Extract data from a data source.
2. Transform the data to match the input format of the model.
3. Train the model parameters.
4. Copy the model parameters to a data store from which it can be served to customers.

These tasks make up what is known as a "data pipeline"; a set of (possibly) dependent tasks that operate on a set of data with the desired result of transforming the data from one form to another. In this case, the transformation is from a set of transaction data to a set of model parameters. Our RS is comprised of a number of tasks operating in a number of pipelines. Therefore, we employ a system explicitly capable of managing these pipelines. Specifically, we have chosen Luigi for this purpose.

Luigi is a open-source Python library for building data pipelines (Bernhardsson and Freider., 2015). Luigi pipelines automatically handle dependency resolution, task execution, and failure modes. Of additional advantage to our use-case is that Luigi comes with built-in support for HDFS and Spark. In this section, we describe the features and functionality of Luigi useful for the recommender system.

Central to Luigi are "tasks", "parameters" and "targets". A task is defined by a set of parameters, while targets define the desired output of a task. In practice, each of these ideas are implemented as Python classes. A target represents anything that can be tested for existence (e.g. a file of HDFS, a file on the local filesystem, or a table in a database). When we implement a task by subclassing Luigi's `Task` class, we define the output `Target` with the `output` method of the `Task` class. Specifically, the `Target` is returned when the `output` method of the `Task` is called. The task is deemed to be complete when this target exists. Luigi `Targets` provide an interface which allows atomic writes to their underlying location. This could be a file on HDFS, or on a local filesystem, for example. Should a task fail mid-way through writing the result, the target will not materialise the required output, and thus Luigi will not deem the task complete. It will then retry the task, or notify a human operator of the failure. This defines Luigi's simple task resolution and failure management mechanisms.

Dependencies between tasks are defined using the `requires` method. For example, if Task A appears in the collection returned by the `requires` method of Task B, Luigi will not execute Task B until the output target of Task A has been satisfied. A task accesses the outputs of

its dependencies through the its `input` method. For example, Task B could get access to the output of Task A by calling its own `input` method.

Figure 3.4 gives a diagrammatic representation of the said example. Task A is returned by the `requires` method of Task B, so Task A will be executed first by Luigi. The file that Task A creates as output is made available to Task B through Task B's `input` method. Finally, Task B does some work with the file on local storage and creates a set of files on HDFS as output.

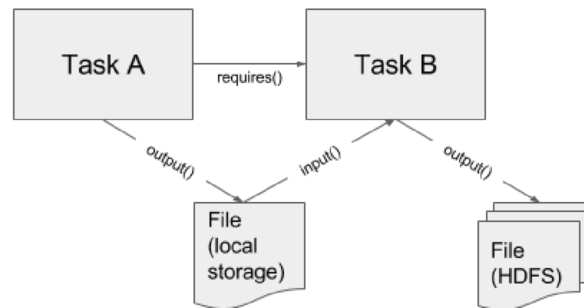


Figure 3.4: Example Luigi pipeline.

Any Luigi tasks that do work are required to implement the `run` method. This method contains the actual code that Luigi runs when the task is available for execution (after its dependencies have been satisfied). Referring to figure 3.4, Task B implements a `run` method that writes everything from the local file output by Task A to a set of file on HDFS. This is indicated by the HDFS files referred to by the `output` arrow originating from Task B.

Adding to the example, let's say that Task B is parametrised by the date on which it is run. This date uniquely identifies an instantiation of Task B, which means that Luigi will only schedule one task with a specific date at any time; even if two Task Bs are submitted to Luigi's scheduler. A task's parametrisation should be reflected in the output of the task so that Luigi can uniquely resolve the execution of that task. Thus, in our example, Task B should include the date in the name of the file on HDFS.

Luigi provides a set of built-in parameter types which take care of parsing string parameters to correct internal representations (for example dates, integers, or booleans). This is useful when defining parameter values in configuration files or on the command line. Figure 3.5 brings these concepts together in an implementation of Task B. The code captures the following concepts:

1. All Luigi tasks are a subclass of the Luigi `Task` class.
2. Parameters are defined as class variables. In this case, we have defined a class variable of type `DateParameter`, with the default value of today. This allows us to specify a string in the Luigi configuration file and have it automatically converted to a Python date object.

3. The `requires` method shows a dependency definition. In this case, we have one dependency (Task A) in the list, but we could also have multiple dependencies in the list, or we could replace the list with a dictionary of name-task pairs for convenient referencing of dependencies.

4. The `output` method shows how a target is usually defined. The path passed in to the `Target` constructor is also parametrised by the date parameter, which has been parsed by Luigi and is available to the task as a Python date object.

5. The `run` method performs the business logic of the task using the inputs and outputs provided by the Luigi framework. In this case, the input is a file on local disk and the output is a file on HDFS. Luigi provides convenience methods for manipulating input and output targets, such as the `open` method seen here. This function is atomic, meaning that any writes to the file will only be written if the task is successful, with the corollary that if an error occurs during writing of the output, all writes will be discarded.

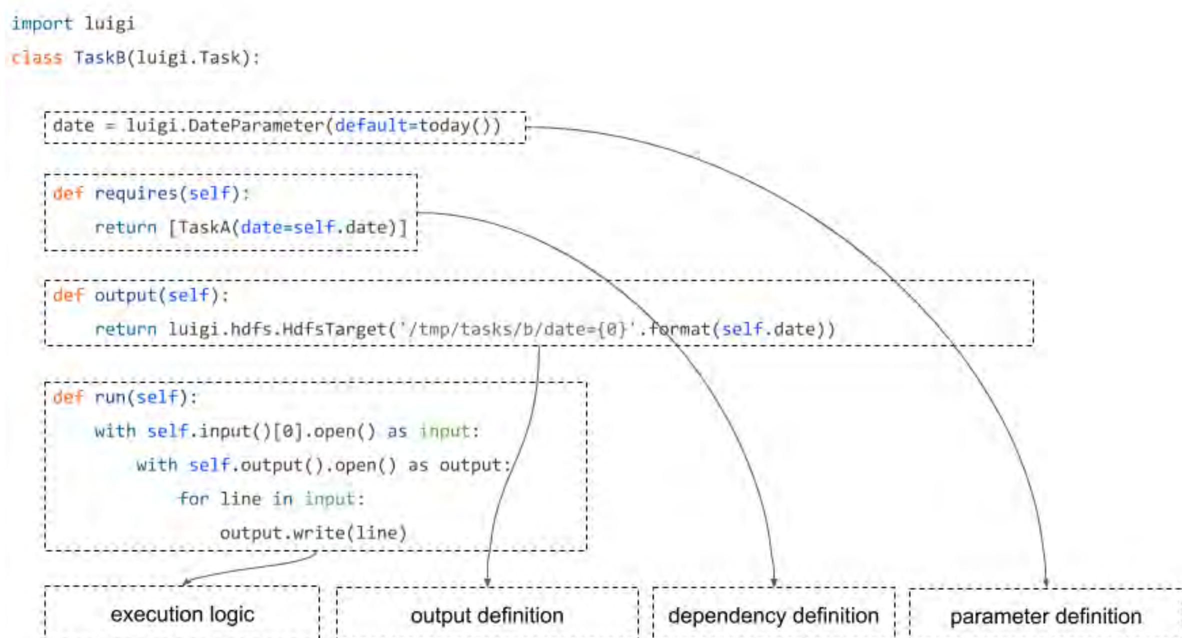


Figure 3.5: Definition of an example Luigi task.

Like the example above, the ingestion framework is a Luigi data pipeline made up of a set of tasks. In the next section, we explain the ingestion pipeline in more detail.

3.6 Data Ingestion

The purpose of the ingestion framework is to make available to the data preparation framework the entire set of successfully completed customer transactions. Toward this end, the inges-

tion framework provides two interfaces for Luigi tasks which allow the programmer to quickly implement tasks for extracting data from different data sources.

The first task interface specifies a Spark-compatible data source. This task does no actual work; its purpose it to check that the data source exists and is accessible. Luigi provides a useful task type for this purpose called an `ExternalTask`. This task does not implement the `run` method, and specifies a single output target which can be any Luigi target. The task will not be marked as complete until the target exists. Therefore, it is "external" in the sense that it is only complete when an external system has created its output target.

The second task interface allows the programmer to specify a procedure for extracting data from the data source and storing it on HDFS. It is dependent on any task the implements the above interface. It uses the Spark DataFrame API to load data from the data source, then write it out to a location on HDFS (provided by the user). Luigi supports the programmer in writing Spark tasks through the `SparkTask` class, which handles packaging the task code and submitting it to the Spark cluster. Additionally, Luigi can check if the task has been successfully completed with its built-in support for HDFS. This makes Luigi a good fit for combining tasks that rely on Spark and HDFS.

For this dissertation, we have implemented these two tasks to read data from a MySQL[7] database table and write it out to HDFS. They are chained together in an application called the "ingestion pipeline", which triggers the second task using Luigi's simple triggering infrastructure. This consists of a command line application that takes as arguments the name of a task and its parameters. The application then starts Luigi and runs the pipeline.

3.6.1 Ingestion Pipeline

The ingestion pipeline is an implementation of the two interfaces described above. Figure 3.6 shows the relationship between these tasks. Our implementation of the first task interface checks for the existence of a table in a MySQL database with the same structure as in table 3.1. The implementation of the second task interface simply provides the path on HDFS to which we want to write the data. We can periodically run this pipeline by triggering the second task using Luigi's triggering system.

Notice that we use a different strategy for data ingestion compared to Oryx, Velox and TencentRec. These three systems use a streaming ingestion strategy, while we use a batch ingestion approach. Oryx and Velox provide an API call which a programmer uses to ingest individual customer-product interactions into the system. In addition, Oryx allows programmers to add records directly onto its event bus, Kafka. This is the same strategy employed by

[7] `https://www.mysql.com/`

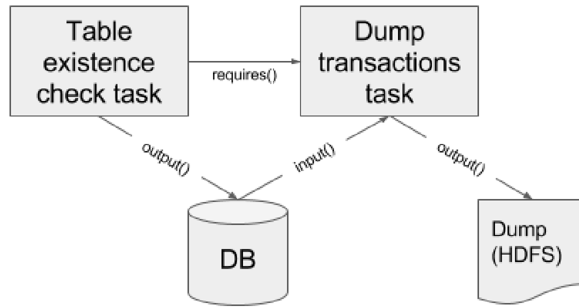


Figure 3.6: Luigi tasks making up the ingestion framework.

TencentRec, except TencentRec uses TDAccess. In our approach, we periodically load data into the RS in large batches. In fact, for the prototype described in this dissertation, at every batch interval we load the entire purchase history of all customers. Unfortunately, our environment precludes a streaming ingestion system. As a result, batch ingestion is the only choice.

3.7 Data Preparation

The data preparation framework consists of four Luigi task interfaces, shown in the dotted box of figure 3.7. The first task takes as input the data on HDFS generated by the ingestion framework, and filters out unwanted data, then augments the data with information relevant to tasks further downstream. The next three task interfaces are concerned with splitting the data into training, validation and test data, in preparation for model building and evaluation.

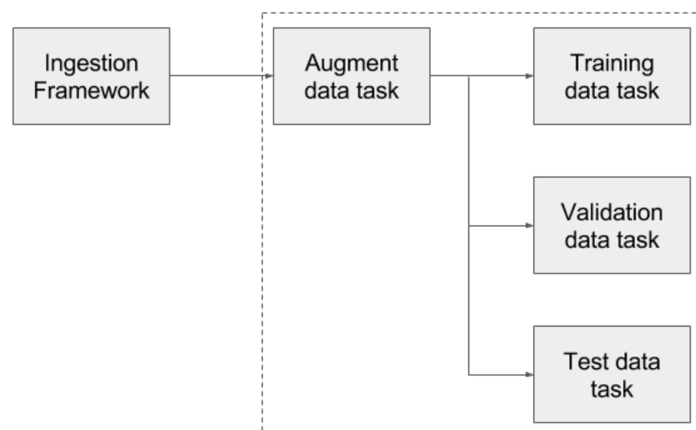


Figure 3.7: Luigi tasks making up the data preparation framework.

3.7.1 Prefiltering and Data Augmentation

The prefiltering and augmentation interface consists of one Luigi task which takes as input the data on HDFS generated by the ingestion framework (see section 3.6). The objective of the task

is to filter out unwanted data and augment the data with information relevant to tasks further down the various pipelines. The task performs the following filtering steps before augmenting the filtered data:

1. Filter out purchases that occurred before a specified date. For example, we may only want to build models on data that is less than a year old.

2. Filter out purchases of products that have been bought by fewer than a specified number of customers.

These filtering conditions imply that the task be parametrised by a date before which purchases are deemed irrelevant, and the number of unique customers to have purchased a product before that product can be included in the recommendation models. These parameters are configurable and therefore should be tuned for best model performance. After the data has been pre-filtered, it is augmented with information about the products and customers appearing in each transaction. Thus, in addition to the fields appearing in table 3.1, the following fields are added to each transaction:

- The date on which the customer first made a purchase.
- The date on which the product was first purchased.
- The date on which the customer first purchased the product.
- The proportion of transactions that occurred before this transaction.

These fields provide enough information for the data splitting interfaces to split the data into training, validation, and test sets. They also support the various model tasks in transforming input data to meet their individual requirements.

The computational heavy-lifting of computing the filters and augmentations is left to Spark, and specifically the DataFrame API. This makes it trivial to add new filters and augmentations should the need arise. The programmer need only be familiar with common SQL concepts and syntax. The resulting filtered and augmented dataset is written to a set of files on HDFS for consumption by the data splitting component of the framework, which we describe next.

3.7.2 Data Splitting

The purpose of the data splitting interfaces is to make separate sets of training and evaluation data available for model building and model evaluation, respectively. In general, models are built from the full filtered and augmented dataset or a training dataset, and are evaluated on a

validation dataset or test dataset. The Luigi tasks in this component of the framework output are concerned with splitting the full filtered and augmented transaction dataset into training, validation and test datasets.

We discussed in section 2.8 that training, validation and test datasets can be constructed in many different ways. For example, we could split the full pre-filtered and augmented data based on a global proportion. In an example of this scheme, the first 80% of transactions go into the training set, the next 10% go into the validation set, and the remaining 10% go into the test set. In another scheme, we could partition each customers transactions into the different datasets. For example, the first half of each customers transactions could go into the training set, the second 25% could go into the validation set and the remaining 25% could go into the test set. In fact, there are infinitely many variations on these schemes. Extensibility in this respect is supported in our RS through the use of the Luigi `TaskParameter`, which allows Luigi tasks to take other tasks as parameters. This is helpful in our case because tasks downstream from the data splitting can define a generic interface for the tasks which generate their input data, into which we can plug different training, validation, and test data splitting strategies. For example, to train a model with a new strategy for creating training data, one would implement the strategy in a new task, then simply use the new task as the training task parameter for a downstream model task, without having to modify the model task. Additionally, this allows us to pass in the filter and augmentation task to downstream model building tasks, which will result in a model trained on the entire dataset.

As an initial attempt, we have implemented in our RS the splitting strategy described above based on a global proportion. The three tasks depend on the pre-filtering and augmentation task. They are parametrised by a proportion and the current date. The training data task first selects all transactions from the pre-filtered and augmented dataset that appear before the proportion. Then, these transactions are further augmented with the number of distinct products that each customer has purchased in that dataset. The result is saved to a set of files on HDFS.

The validation data task does something similar, but selects the first half of the remaining transactions from the pre-filtered and augmented dataset. Then, the task filters this data so that no repeat transactions appear, i.e. it filters out all customer-product transactions that have appeared in the training dataset. The filtered result is materialised to a set of files on HDFS. The test data split task is exactly the same as the validation task, but operates on the last of the data remaining from the pre-filtered and augmented dataset. As an example of result of the above procedure, we refer to table 3.4.

Table 3.4a gives an example of a full filtered and augmented transaction dataset (for brevity, we show only those fields relevant to the discussion). In this example, training, validation and test splitting tasks have been given a proportion parameter of 0.5. As a result, half of the

	Customer ID	Product ID	Date	...	Customer First Purchase of Product	Proportion
1	1	2	2015–01–01 08:01:23	...	2015–01–01 08:01:23	0
2	2	1	2015–01–01 08:02:20	...	2015–01–01 08:02:20	0.17
3	1	3	2015–01–01 08:15:00	...	2015–01–01 08:15:00	0.33
4	1	2	2015–01–01 08:30:25	...	2015–01–01 08:01:23	0.6
5	1	1	2015–01–01 08:45:15	...	2015–01–01 08:45:15	0.67
6	2	3	2015–01–01 09:30:52	...	2015–01–01 09:30:52	0.83

(a) Pre-filtered and augmented dataset.

	Customer ID	Product ID	Date	...	Customer First Purchase of Product	...
1	1	2	2015–01–01 08:01:23	...	2015–01–01 08:01:23	...
2	2	1	2015–01–01 08:02:20	...	2015–01–01 08:02:20	...
3	1	3	2015–01–01 08:15:00	...	2015–01–01 08:15:00	...

(b) Training dataset.

	Customer ID	Product ID	Date	...	Customer First Purchase of Product	...
5	1	1	2015–01–01 08:45:15	...	2015–01–01 08:45:15	...

(c) Validation dataset.

	Customer ID	Product ID	Date	...	Customer First Purchase of Product	...
6	2	3	2015–01–01 09:30:52	...	2015–01–01 09:30:52	...

(d) Test dataset.

Table 3.4: Training, validation and test data splitting strategy.

transactions (all those with a proportion less than 0.5) have been placed in the training set, seen in table 3.4b. The next quarter of the transactions have been placed in the validation set, seen in table 3.4c. Note that transaction four in the full dataset is the same as the first transaction in that dataset. As a result, even though transaction four should appear in the validation set, it has been filtered out, as it already appears in the training set. The final transaction (with a proportion greater than 0.75) has been placed in the test set (table 3.4d).

Of the three RSs reviewed in section 2.9, Oryx provides the most thorough treatment of data preparation and augmentation. During batch layer recomputation, Oryx splits data into training and test sets using a global time-based split, like our RS. Since Oryx has not been designed to compare models of different types, it has no concept of a validation set. This is where our system differs; we have explicitly designed for model comparison across model types. Therefore, we provide a validation set for tuning parameters within a specific model class, and a test set for comparing model performance across model classes.

Once we have sets of training, validation and test data, we can build, evaluate and compare

recommendation models. Two sets of tasks work together to achieve this goal, namely the model building tasks and the model evaluation tasks. Next, we take a deep-dive into the various model building tasks we have implemented in the RS.

3.8 Model Building

The model building framework is made up of a single Luigi task interface. Figure 3.8 shows how this task fits into our discussions thus far; it is dependent on tasks from the data preparation framework. In particular, the model building task expects to take as input either the training data task, or the data filtering and augmentation task. This allows the model to be trained over the full dataset, in the case where the model is to be published, or on a training dataset, in situations where we wish to perform model cross-validation or model testing. For the purposes of this dissertation, we have chosen to implement two models, namely, IMF and BPR. We described the formulation of these models in detail in section 2.7. Here, we focus on the implementation of the training process for these models, with specific emphasis on how their implementation supports increasingly large volumes of data.

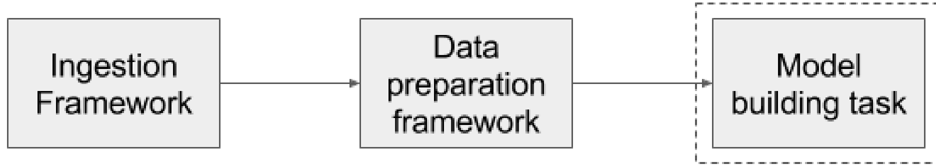


Figure 3.8: Model building tasks fit into the workflow after the data preparation framework.

Before detailing each of the models, it is worth providing a recap of the recommendation framework introduced in section 2.6:

- $U = \{u_1, \ldots, u_M\}$ is a set of M customers and $I = \{i_1, \ldots, i_N\}$ a set of N products,
- we use indexing letters u, v for customers and i, j for products,
- $S \subseteq U \times I$ is the set of observed customer-product transactions,
- $P = (U \times I) \setminus S$ is the set of unobserved (but possible) customer-product transactions,
- $I^u = \{i \in I : (i, u) \in S\}$ is the set of items bought by customer u, and
- $D_S = \{(u, i, j) \,|i \in I^u \wedge j \in I \setminus I^u\}$ is the set of triples (u, i, j) where i has been bought by u, but j has not.

The objective of a recommendation model is to find a ranking $\hat{r}_u$ of all products in I for each customer u in U. This can be formulated as $\hat{r}_u : I \rightarrow \{1, 2, \ldots, |I|\} \forall u \in U$, where $\hat{r}_u(i)$ is the

rank of item i for a user u. The ranking function is generally modelled through some scoring function $\hat{y}_u(i)$, where ranking for each u is done by computing scores for all i and sorting by scores. The formal link between $\hat{r}$ and $\hat{y}$ can be defined as:

$$\hat{r}_u(i) := |\{j : \hat{y}_u(j) \geq \hat{y}_u(i)\}|$$

Finally, $\hat{y}$ is parametrised by a set of model parameters $\vec{\Theta}$, which in turn uniquely define the ranking $\hat{r}$.

3.8.1 Implicit Matrix Factorisation

Recall from section 2.7 that the IMF algorithm attempts to estimate a set of customer-factors X, and product-factors W. Pairs in S are assigned values s_{ui} based on some level of interaction between u and i. For e-retail, Hu et al. (2008) suggest that s_{ui} indicate the number of times u purchases i. This is an arbitrary choice, however, and there are multiple other representations we could choose. For example, we could incorporate the number of times a u has clicked on an i, or the amount of time a u has spent looking at i. Additionally, we might choose to weight more recent customer purchases higher than older ones as a customer's taste my change slightly over time (Johnson, 2014).

Two sets of variables, q_{ui} and c_{ui} are introduced to capture some notion of confidence in the preferences that s_{ui} measure. The q_{ui} are introduced to indicate that customer u has a preference for i, and are derived by binarising S and P:

$$q_{ui} = \begin{cases} 1 & \text{if } (u,i) \in S \\ 0 & \text{if } (u,i) \in P \end{cases}$$

As s_{ui} grows, we have a stronger indication that u has a preference for i. The set of variables c_{ui} are designed to capture this notion:

$$c_{ui} = \begin{cases} 1 + \alpha s_{ui} & \text{if } (u,i) \in S \\ 1 & \text{if } (u,i) \in P \end{cases}$$

The goal is then to find customer factors, $\vec{x_u} \in \mathbb{R}^f$ and product factors, $\vec{w_i} \in \mathbb{R}^f$ such that q_{ui} is as close to $\vec{x_u}^T \vec{w_i}$ as possible, and the contribution of factors is weighted by c_{ui}. Therefore, factors are computed by minimising the following cost function:

$$\min_{\vec{x_*}, \vec{w_*}} \sum_{u,i} c_{ui} \left(q_{ui} - \vec{x_u}^T \vec{w_i} \right)^2 + \lambda \left(\sum_u \vec{x_u}^T \vec{x_u} + \sum_i \vec{w_i}^T \vec{w_i} \right)$$

An ALS procedure is used as an efficient optimisation process. It is derived by observing

that when customer factors or product factors are fixed, the cost function with respect to the non-fixed set of factors becomes quadratic, so its global minimum is readily computed by differentiation. We only state the result here, for a full derivation of the procedure please refer to section 2.7. To update customer factors when product factors are fixed, we must solve for $\vec{x_u}$ in the following system of equations:

$$\left(W^T W + W^T \left(C^u - I \right) W + \lambda I\right) \vec{x_u} - W^T C^u \vec{q^u} = 0, \tag{3.1}$$

where C^u is the diagonal matrix with all c_{ui}'s on the diagonal, and $\vec{q^u}$ is the vector of q_{ui} for all i. Note that the computation of $W^T W$ can be reused across updates of customer factors, so we only need to compute it once per iteration. Also, to update each customer factor we only need the factors of the products rated by the customer, as $(C^u - I)$ and $\vec{q^u}$ are zero everywhere except at products with which u has interacted. This is important for parallelisation of the algorithm. Factors can be updated in parallel if we ensure that each of the product factors rated by a customer is available at the same node on which we are updating the customer factor.

Similarly, to update the product factors when customer factors are fixed, we solve for each product factor in the follow system of equations:

$$\left(X^T X + X^T \left(C^i - I \right) X + \lambda I\right) \vec{w_i} - X^T C^i \vec{q^i} = 0,$$

The idea is that the factors will stabilise after a few "sweeps" of fixing user factors, and updating each item factor, then fixing item factors and updating each user factor.

The sequential version of the algorithm described in Hu et al. (2008) has a computational complexity which is linear in the size of the input data, S. This is a reasonable scaling property, but we can do better by parallelising the update of factors. Parallelisation also allows the algorithm to "scale-out". When the input data becomes too large to store in one machine, we can partition the dataset out onto multiple machines. This is a good use-case for Spark and in fact Spark comes with a built-in scalable implementation of IMF, which we describe next.

The Spark implementation of IMF expects as input a DataFrame with at least the following three columns; customer ID, product ID and rating. The rating is represented by the s_{ui} variables in the description above. As discussed in section 3.7.2, the input data for the model building tasks has a customer ID and product ID, but nothing equivalent to s_{ui}. Therefore, we synthesise a rating column for each customer-product pair before passing the data to the Spark model for training. To do this, we follow the advice of Hu et al. (2008) and create a rating column by counting the number of times u purchases i. In practice, this involves grouping the data by customer ID and product ID, then counting the number of transactions in each group. We then pass this grouped DataFrame to the Spark IMF algorithm for training. In addition to the input data, the Spark model takes six parameters, which we expose as parameters for the

Luigi wrapper task as well:

- `rank`: the number of elements in the factors (f in the description of IMF above).
- `numUserBlocks`: number of customer blocks to create.
- `numItemBlocks`: number of product blocks to create.
- `maxIter`: number of sweeps of updating factors to perform.
- `regParam`: regularisation parameter (λ in the description of IMF above).
- `alpha`: value by which interactions are scaled to create confidence variables (α in the description of IMF above).

The algorithm starts by partitioning the counted transactions data into four RDDs, namely *customer-inlinks*, *customer-outlinks*, *product-inlinks* and *product-outlinks*. We can imagine the inlink RDDs as views of the counted transaction data. The customer-inlink RDD contains the data partitioned by customer ID, while the product-inlink RDD contains the data partitioned by product ID. This concept is reflected in figure 3.9. In this example we have six products and six customers partitioned into three customer-inlink blocks (orange, cyan and magenta), and three item-inlink blocks (purple, blue and green). In actual fact, the inlink is an RDD of tree-like data structures. To explain how these data structures are generated, we use the example of a customer-inlink block. First, the input data is partitioned by customer ID into `numUserBlocks` partitions. This means that we are guaranteed that any customer's transactions will appear wholly in one partition. The data in a partition is gathered into a single block and the block is assigned unique ID. As shown in figure 3.10, the block ID is the highest level in the tree-like data structure (for explanatory purposes, the block ID is a string, but in practice it's actually an integer). The next level of the tree contains the customer IDs that are present in the block, and the final level contains an array of tuples that identify a transaction. Figure 3.10 calls the fields in the tuple *product block ID*, *product local ID* and *rating*. The product block ID identifies the product block in which the product in the transaction resides. Product local ID refers to the index of the product in the product block. Finally, the rating is the s_{ui} we have defined above. Note that the actual product ID is not stored here, only the index within a block. This is necessary to index into blocks of received product factors when the time comes to update customer factors. Product inlink blocks are constructed in exactly the same manner, but with considering all the above from the perspective of product IDs.

The outlink blocks are also partitioned by customer ID or product ID. The purpose of the outlink block is to identify the the exact factors that are required to be sent to inlink blocks for updating of factors. We use the example of a product outlink block in figure 3.11 to aid the discussion. Like the inlink RDDs, the outlink RDDs contain one tree-like data structure

product ID

customer ID	0	1	2	3	4	5
0	1	-	-	-	-	1
1	-	1	-	-	1	-
2	1	-	-	1	-	-
3	-	-	1	-	1	-
4	-	-	-	1	-	-
5	-	-	1	-	1	-

Figure 3.9: Example transaction matrix.

Figure 3.10: Customer inlink block example.

per partition. At the top level, the *product block ID* identifies the block to which the data structure belongs. For a product outlink block, we wish to identify customer inlink blocks to which product factors need to be sent, and for each of those customer inlink blocks, we wish to identify the exact product factors that are required. As a result, the second level identifies the customer blocks which contain ratings of products included in the top-level product block. Referring back to figure 3.11 we see that the two customer blocks appearing at the second level are orange and cyan. This makes sense when we look at the customer blocks that have ratings in the purple product block in figure 3.9 – only the orange and the cyan customer blocks have ratings in the purple product block. The final level of the product outlink block specifies exactly which of the products in the top-level product block are required for each of the mid-level customer blocks. Figure 3.11 shows that these are product local IDs (index of products within the top-level block) stored as an array. Customer outlink blocks are constructed in exactly the same manner, but swapping the roles of products and customers.

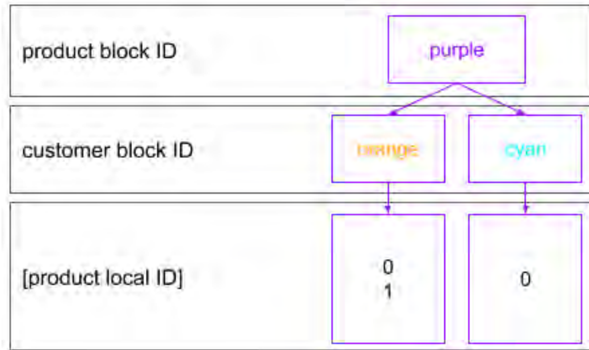


Figure 3.11: Product outlink block example.

Once the four RDDs have been constructed, they are stored in the memory of the cluster, as seen in figure 3.12. The outlink and inlink blocks for a given set of product IDs or customer IDs are partitioned with the same scheme, and as a result they will always land up in the memory of the same machines. This is an important feature of the algorithm because it avoids shuffling data across the network when computing product or customer factors.

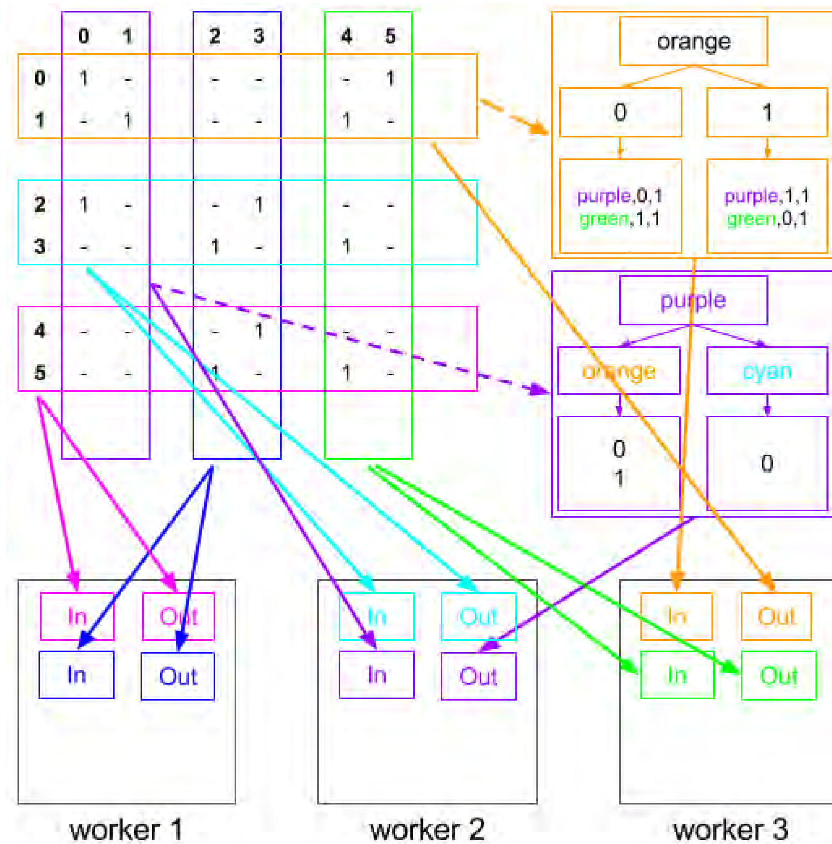


Figure 3.12: An example of how blocks could be distributed throughout a Spark cluster with three worker nodes.

The next step in the algorithm is to initialise the customer and product factor RDDs. This is achieved through a map transformation of the inlink blocks. For each block in the inlink

RDD, and each ID in the block, a non-negative random unit vector of length `rank` is generated. This results in blocks consisting of a single key-value pair, where the key is the block ID and the value is a matrix of factors. The matrix has the same number of rows as IDs in the block and `rank` columns. Figure 3.13 gives an indication of how the factor blocks are distributed across the cluster. Factors corresponding to a set of IDs in a specific inlink block end up on the same machines as the inlink block.

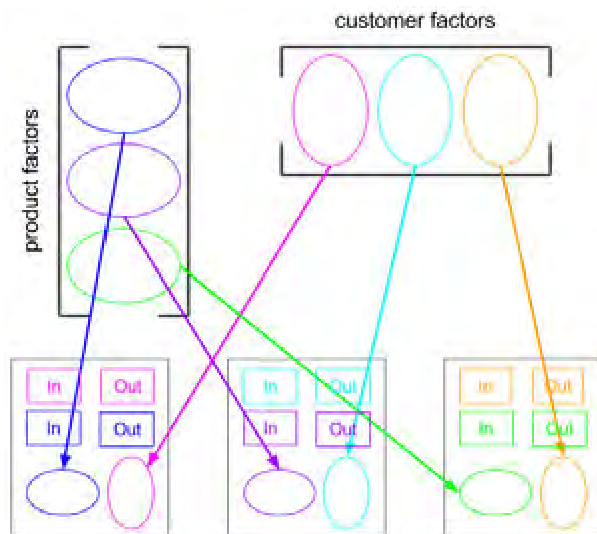


Figure 3.13: Factor blocks are distributed alongside inlink and outlink blocks.

Now the iterative update of customer, then product factors can begin. Using the Spark RDD API, the following steps are performed in parallel across the cluster to perform an update of customer factors:

1. The matrix product $W^T W$ is computed and distributed throughout the cluster.

2. The product outlink blocks are joined with the product factors on product block ID. As shown in figure 3.14, this happens in parallel across the cluster. No data needs to be shuffled across the network as the outlink and factor blocks are partitioned in the same manner. The result is an RDD which contains the outlink information and factors. Recall that the product outlink block identifies the customer inlink blocks that depend on factors from the product factor block, and exactly which factors are required. This information is used to pick out the required factors into a new key-value RDD. The key is the customer block ID and the value is a tuple of product block ID, and array of product factors.

3. A `groupByKey` operation is run on the result above, which shuffles the required factors to their corresponding customer inlink blocks, identified by the customer block IDs. This process is represented in figure 3.14 for the orange customer inlink block. In parallel, the factors required from the purple and green product blocks are picked out in the join step (represented by the smaller ovals). Next, these factors are shuffled to their destination in

the group-by-key step. For the purple product factors this means moving data across the network, while for the green no data needs to move across the network as the green factor block resides in the same machine as the orange customer inlink block.

4. To solve for the new customer factors, we join the shuffled product factors with the customer inlink block on customer block ID. Now, for each customer ID within the block we have enough information to solve the system of equations in equation (3.1).

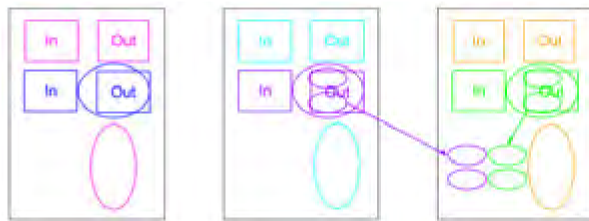


Figure 3.14: Shuffling product factors required by the orange customer inlink block.

To aid solving equation (3.1), Spark programmers have created a construct called `NormalEquation`. This object represents a normal equation to solve the following weighted least squares problem:

$$\min_{\vec{x}} \sum_i c_i \left(\vec{a_i}^T \vec{x} - b_i \right)^2 + \lambda \vec{x}^T \vec{x}.$$

To solve this problem, we take the derivative of the above with respect to $\vec{x}$ and set it to zero, which gives us the following "normal equations":

$$\sum_i c_i \left(\vec{a_i} \vec{a_i}^T \vec{x} - b_i \vec{a_i} \right) + \lambda \vec{x} = 0. \tag{3.2}$$

The Spark `NormalEquation` object represents this set of equations. It provides operations to add an observation, merge with another normal equation and solve using the Cholesky decomposition. The add operation takes an observation $\vec{a_i}$ and b_i, and optionally a weight c_i as input, and appends the observation to Spark's internal representation of equation (3.2). The merge operation takes as input another normal equation object and adds their internal representations together.

The internal representation of equation (3.2) consists of two arrays. The first array keeps track of the upper-triangle of $\sum_i c_i \vec{a_i} \vec{a_i}^T$, since $\vec{a_i} \vec{a_i}^T$ is symmetric. If $\vec{a_i} \in \mathbb{R}^f$, then this array has length $\frac{f(f+1)}{2}$. The reason for this is clear if we imagine an $f \times f$ symmetric matrix whose rows are indexed by $l \in \{1, \ldots, f\}$ and columns indexed by $m \in \{1, \ldots, f\}$. To represent the entire matrix, we need only store its upper-triangle, i.e. those elements whose indices have $m \geq l$. Thus, in each column, there are m which need to be stored. As a result, to find the

total number of elements that need to be stored, we perform the following sum:

$$\sum_{m=1}^{f} m = \frac{f\,(f+1)}{2}.$$

The second array keeps track of $\sum_i c_i b_i \vec{a_i}$. It has length f. When a new observation, $\vec{a_j}$, b_j and c_j, is added to the normal equation, Spark performs two efficient linear algebra operations. The first adds the array representing the upper-triangle of $c_j \vec{a_j} \vec{a_j}^T$ to the array representing the running sum $\sum_i c_i \vec{a_i} \vec{a_i}^T$, and the second adds the new $c_j b_j \vec{a_j}$ to the running sum $\sum_i c_i b_i \vec{a_i}$.

The requirement for a merge operation is that the input normal equation object has the same dimensionality as the current object. If this is true, Spark will add the internal representations of the objects together. This can also be completed efficiently with two algebraic operations. In the first, the two upper-triangles of the running sums $\sum_i c_i \vec{a_i} \vec{a_i}^T$ are added together, and in the second the two arrays representing the running sums $\sum_i c_i b_i \vec{a_i}$ are added together.

In step one of the parallel update process for $\vec{x_u}$ described above, the matrix product $W^T W$ is stored as a normal equation object. To represent $W^T W$, the $\vec{w_i}$ are iteratively added to a normal equation object whose $\vec{a_i}$'s are the product factors, $\vec{w_i}$, c_i's are one and b_i's are all zero. Thus, we get:

$$\sum_{i}^{N} \vec{w_i} \vec{w_i}^T \vec{x_u} + \lambda \vec{x_u} = 0, \qquad (3.3)$$

where λ corresponds to the `regParam` input parameter. This object is then distributed to each of the workers in the cluster.

Recall that to solve for $\vec{x_u}$ in equation (3.1) in parallel, we only need the product factors, $\vec{w_i}$, of those products bought by a customer u. This is exactly the information we have collected in step four above. Thus, for every customer factor held by a worker and in parallel, we create a new normal equation object and iteratively add to it the $\vec{w_i}$'s and the confidence variables c_{ui}.

$$\sum_{i \in I^u} (c_{ui} - 1) \left(\vec{w_i} \vec{w_i}^T \vec{x_u} - \frac{c_{ui}}{c_{ui} - 1} \vec{w_i} \right) + \lambda \vec{x_u} = 0$$
$$\implies \sum_{i \in I^u} (c_{ui} - 1)\, \vec{w_i} \vec{w_i}^T \vec{x_u} - c_{ui} \vec{w_i} + \lambda \vec{x_u} = 0$$

Referring back to equation (3.2), the $\vec{a_i}$'s are $\vec{w_i}$'s and, the b_i's are $\frac{c_{ui}}{c_{ui}-1}$, and the c_i's are $c_{ui} - 1$.

This normal equation object is merged with the normal equation object we distributed earlier

represented by equation (3.3) and we get

$$\sum_{i}^{N} \vec{w_i}\vec{w_i}^T\vec{x_u} + \sum_{i\in I^u} \Big((c_{ui}-1)\,\vec{w_i}\vec{w_i}^T + \lambda \Big)\,\vec{x_u} - c_{ui}\vec{w_i} = 0$$
$$\implies \big(W^TW + W^T(\,C^u - I\,)W + \lambda I\big)\,\vec{x_u} - W^TC^u\vec{q^u} = 0,$$

which is exactly the system of equations we need to solve in equation (3.1).

The same procedure as the above is then followed for updating of product factors. This iterative process of fixing product factors and solving for customer factors in parallel, then fixing customer factors and solving for product factors in parallel is repeated `maxIter` times.

The output of the algorithm is two RDDs. One of product factors comprising tuples of product ID and product factor, and the other of customer factors comprising tuples of customer ID and customer factor. The Luigi wrapper task saves these two RDDs to HDFS as a set of parquet files.

3.8.2 Bayesian Personalised Ranking

Before describing our scalable implementation of BPR, we provide a recap of the formulation of the model as described in section 2.7. The parameters in $\vec{\Theta}$ are found by maximising the posterior distribution:

$$p\left(\vec{\Theta}\middle|D_S\right) \propto \prod_{(u,i,j)\in D_S} \sigma\left(\hat{y}_u\left(i\right) - \hat{y}_u\left(j\right)\right) \exp\left(-\frac{\lambda}{2}\vec{\Theta}^T\vec{\Theta}\right). \tag{3.4}$$

We perform the optimisation by minimising the negative log posterior in equation (3.4) using stochastic gradient descent. In this scheme, we uniformly sample a random triple $(u, i, j) \in D_S$, and take a step in the direction of the minimum of the negative log-posterior. The idea is that after many such updates, the parameters of the model will converge to stable estimates. The update steps are defined as:

$$\vec{x_u} \leftarrow \vec{x_u} - \eta_u \left(- \left[1 - \sigma\left(\vec{x_u}^T\vec{w_i} - \vec{x_u}^T\vec{w_j}\right)\right] (\vec{w_i} - \vec{w_j}) + \lambda\vec{x_u} \right) \tag{3.5}$$
$$\vec{w_i} \leftarrow \vec{w_i} - \eta_i \left(- \left[1 - \sigma\left(\vec{x_u}^T\vec{w_i} - \vec{x_u}^T\vec{w_j}\right)\right] \vec{x_u} + \lambda\vec{w_i} \right) \tag{3.6}$$
$$\vec{w_j} \leftarrow \vec{w_j} - \eta_j \left(\left[1 - \sigma\left(\vec{x_u}^T\vec{w_i} - \vec{x_u}^T\vec{w_j}\right)\right] \vec{x_u} + \lambda\vec{w_j} \right). \tag{3.7}$$

SGD is traditionally implemented in a sequential manner on a single machine. This is not acceptable in our case for two reasons: (1) we expect the size of the data to increase beyond memory capacity of one machine (see section 3.6), and (2) we wish utilise the existing computing infrastructure built for other parts of the RS, namely the Spark cluster. Thus, we attempt to

scale BPR by implementing it on a parallel stochastic learning framework called Splash (System for Parallelising Learning Algorithms with Stochastic Methods) (Zhang and Jordan, 2015).

Sequential stochastic learning algorithms provide many opportunities for distributed implementations. Some implementations use asynchronous parallel updates on parameters stored in lock-free shared-memory (Zhuang et al., 2013; Niu et al., 2011; Liu et al., 2013; Ho et al., 2013). If the time delay of concurrent updates are bounded, the updates preserve correctness of the algorithms (Liu et al., 2013). However, the communication requirements of these algorithms can be overly expensive when implemented on networks of commodity machines. Communication cost can easily dominate computation cost when messages are frequently exchanged across the network. Thus, our use-case is suited more for fully-distributed implementations of stochastic algorithms. However, fully distributed implementations proposed to date have been limited in that they are usually designed for specific algorithms. In contrast, Splash has been proposed as a general, fully-distributed and communication-efficient framework for parallelising stochastic algorithms.

Splash is compelling for our use-case for two reasons. First, Splash has a simple API with which a programmer can develop a sequential stochastic algorithm without thinking about issues of distributed computing. Second, the execution engine which parallelises execution is written on top of Spark, a technology which is already a central part of the recommender system.

The Splash API requires the programmer to implement a slightly stronger version of the base sequential algorithm. More specifically, the algorithm needs to be able to process weighted samples of the data. Many stochastic algorithms, including SGD used by BPR can be generalised to process weighted samples without sacrificing computational efficiency. In addition, weighted samples can be incorporated into the sequential algorithm without thinking about distributed computing, so the programmer is not burdened with much additional complexity.

Like Spark, Splash is written in the Scala programming language, and as such, programmers write applications that use Splash in Scala. Splash extends Spark by providing a data structure called a Parametrised RDD that stores and maintains a distributed dataset. A Splash application starts by creating a parametrised RDD from the usual Spark RDD:

```
val paramRdd = new ParametrizedRDD(rdd)
```

Each parametrised RDD is associated with a vector of shared variables. The shared variables can be thought of as the parameters of the model. They are replicated on every partition of the parametrised RDD, and synchronised in a network-efficient manner by the Splash execution engine. To make a pass over the dataset as in a sequential algorithm, the programmer calls the run method on the parametrised RDD and passes it a processing function as follows:

```
paramRdd.run(process)
```

The processing function adheres to the format:

```
process(elem: any, weight: int, sharedvar: varset,
  localvar: varset),
```

where the arguments are defined as follows:

- `elem`: a single element of the original dataset.
- `weight`: the weight of the element.
- `sharedVar`: the shared variables associated with the partition from which the element originates.
- `localVar`: the local variables associated with the partition. Local variables are provided by Splash, but we don't use them so they are excluded from this discussion.

The execution engine makes one full pass over the dataset each time `run` is called. This means that `process` is called for every element in the parametrised RDD. The framework is communication efficient because inter-node communication to synchronise updates only occurs at the end of a full iteration, as opposed to every time a shared variable is modified.

The intention is that inside the `process` function, one or more shared variables are updated according to the input element. Shared variables are manipulated as key-value pairs. The value of shared variables can be accessed by `sharedVar.get(key)`. Updates are performed by operators such as add and multiply. For example, to add a scalar value to a variable, the programmer would use the expression `sharedVar.add(key, value)`.

The execution engine achieves parallelisation of the algorithm through a distributed process of reweighting and averaging. During execution, every worker sequentially processes its partition of the data to arrive at a local solution. At the end of an iteration, these local solutions are averaged to construct a global update to the shared variable set. Although the averaging step reduces the variance of the local solutions, their bias is large. These local solutions are the result of a large number of threads touching a small subset of samples relative to the entire dataset. Therefore, they have significant bias compared to the update generated by a sequential algorithm running over the full sequence of random samples. The reweighting scheme described above where programmers are asked to implemented a weighted update to the shared variables ensures that each thread processes a weight equal to the total number of samples in the full sequence. This helps individual threads generate a nearly unbiased estimate of the the full

update. In fact, Zhang and Jordan (2015) prove that this scheme achieves the optimal rate of convergence for parallelising SGD for a smooth and strongly convex objective. Additionally, they show experimentally that Splash achieves significant speedup over sequential algorithms for non-convex optimisation tasks.

The SGD updates in equations (3.5) to (3.7) for BPR are easily generalised to handle weighted samples. The update equations become:

$$
\begin{aligned}
\vec{x_u} &\leftarrow \vec{x_u} - \hat{\eta}_u \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] (\vec{w_i} - \vec{w_j}) + \lambda \vec{x_u} \right) \\
\vec{w_i} &\leftarrow \vec{w_i} - \hat{\eta}_i \left(- \left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] \vec{x_u} + \lambda \vec{w_i} \right) \\
\vec{w_j} &\leftarrow \vec{w_j} - \hat{\eta}_j \left(\left[1 - \sigma \left(\vec{x_u}^T \vec{w_i} - \vec{x_u}^T \vec{w_j} \right) \right] \vec{x_u} + \lambda \vec{w_j} \right),
\end{aligned}
$$

where the learning rates $\hat{\eta}_u$, $\hat{\eta}_i$ and $\hat{\eta}_j$ are reweighted by the weight, m, passed to the processing function by the Splash execution engine:

$$
\begin{aligned}
\hat{\eta}_u &\coloneqq \min(1, m\eta_u) \\
\hat{\eta}_i &\coloneqq \min(1, m\eta_i) \\
\hat{\eta}_j &\coloneqq \min(1, m\eta_j).
\end{aligned}
$$

We bound the learning rates by one because SGD in stochastic BPR is known to quickly diverge with step sizes greater than one. The actual implementation of BPR follows from these three modified equations.

Our implementation of BPR is a Scala application that expects as input a DataFrame containing a set of transactions with a customer ID and product ID. The outputs of the data augmentation task and training data task both satisfy this requirement and as a result we can feed the BPR application with either of these datasets. However, the BPR algorithm itself expects to process random samples of $(u, i, j) \in D_S$. Algorithm 4 shows how we generate these samples before training the model. For each $(u, i) \in S$, the code generates m samples of j, where m is a parameter of the BPR application. This parameter originates from the observation in Rendle et al. (2009) that BPR usually converges after a subsample of $m|S|$ update steps.

In addition to m, the Luigi task which wraps this model takes the following extra parameters which it passes to the model:

- `rank`: number of elements in the factors.
- `lambda`: regularisation parameter (λ in the description of BPR above).
- `customerStepSize`: size of step to take when updating customer factors (η_u in the description of BPR above).

Algorithm 4 Generate random triples for consumption by BPR algorithm.

Input: Transactions S.
Output: Data set containing $m|S|$ samples of D_S to be fed to the BPR algorithm.

```
1: ids ← distinct product IDs from S.
2: Broadcast ids to each worker.
3: groups ← Group S by customer ID.
4: d_S ← groups.flatMap (u, group)) ⇒
5:     set ← convert group of product IDs to set.
6:     interactions ← empty list.
7:     while interactions.length < m do
8:         i ← random sample from set.
9:         j ← random sample from ids not in set.
10:        interactions.append(u, i, j)
11:    end while
12:    interactions
```

- `posStepSize`: size of step to take when updating positive product factors (η_i in the description of BPR above).
- `negStepSize`: size of step to take when updating negative product factors (η_j in the description of BPR above).
- `iters`: number of passes Splash takes over the dataset.
- `initPartitions`: initial number of partitions into which the data is split.

The algorithm starts by randomly reshuffling the input RDD (generated by algorithm 4) into `initPartitions` across the workers. This ensures that each worker gets a random subset of the data. Next, a parametrised RDD is created from the reshuffled RDD. This parametrised RDD is responsible for managing the sets of shared variables made up of all the product factors and all the customer factors. In Splash, each partition of the data has its own copy of the shared variables. Thus, for each partition we initialise the entire set of M customer factors and N product factors (in exactly the same manner that Spark does in the IMF algorithm). The scalability of the algorithm is limited by this design choice, however, as we can only handle as many factors that will fit in the memory of one worker. In the future, more careful arranging of the data could avoid this pitfall.

Next, we instruct the Splash execution engine to start processing the data by calling the `run` method of the parametrised RDD we have created above. The algorithm will call the `run` method `iters` time consecutively. This means that the algorithm will make `iters` passes over the data with a round of communication for synchronising shared variables after each pass.

Once the algorithm is finished, the resulting product factors and customer factors are converted back to two standard spark RDDs with exactly the same format as the output of the

IMF algorithm above. Finally, the Scala application writes these factors to a set of files on HDFS so that they can be consumed by downstream tasks.

A drawback of our model building strategy is that we have not implemented low-latency updates. We follow a batch model for updates, where the entire model is updated on a periodic basis. The system cannot update models as new customer interactions are generated by the website, and therefore the RS cannot respond to a customer's changing needs in real-time. This is mainly due to system constraints. At the time that the project was started, there was no mechanism for propagating customer interactions beyond the transactional systems as they occurred. Since then, however, this capability has been added and one of the focuses of future work will be to add low-latency model updates.

3.9 Model Evaluation and Testing

The model management activities of the RS are based on two Luigi task interfaces, which are used in two Luigi pipelines. The interfaces are shown in figure 3.15, labelled as the evaluation task and summary task. These tasks are used to evaluate models from the model building framework on data provided by the data preparation framework, and then provide summaries of these evaluations for the data scientist to use in managing which models are served to customers; a feature which is unique to this RS.

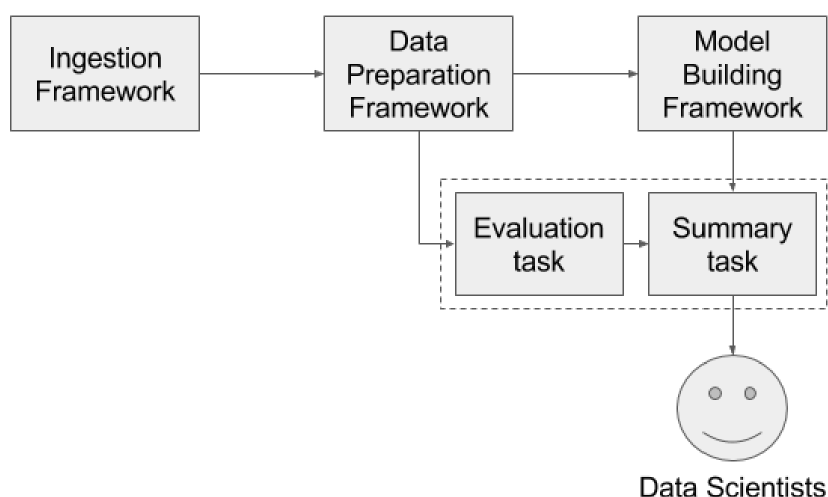


Figure 3.15: Model evaluation and testing framework.

As an aid for the decision to serve one recommendation model over another, model evaluation summaries have been given first-class support in our system. We recognise that model management is a key activity in the lifecycle of a RS. When making a decision to pursue one course of action over another, we should use sound scientific reasoning to make our choice. The decision to use one recommendation model for serving recommendations to customers over an-

other is no different, and the model evaluation summaries provide us evidence for this decision. With their use, we can provide probable cause for changing the set of parameters used to train a model, or for changing the model used in a specific context altogether.

The model evaluation process can be performed in two contexts. First, in cross-validation, where a series of models of the same class are evaluated with different parameters on validation data, with the goal of choosing the most promising set of parameters. This occurs in the cross-validation pipeline. Second, during model testing, where the most promising models of different classes are evaluated against test datasets with the goal of choosing the most promising model class. Model testing is performed in the model testing pipeline.

3.9.1 Model Evaluation

The model evaluation task interface provides abstractions for evaluating a class of models based on some error metric. For this dissertation, we have implemented an evaluation task based on the Area Under the ROC Curve (AUC) metric for the class of LF models to which IMF and BPR belong.

At the very least, an implementation of the evaluation task is dependent on two other tasks, each specified as Luigi `TaskParameters`. These include a model building task and an evaluation data task. The evaluation data task is either a validation data task or test data task, allowing the evaluation to be performed in the context of model cross-validation or model testing, respectively. The specified model building task should train a model that is supported by the evaluation strategy used by the evaluation task. For example, an AUC evaluation task specific to latent factor models should only be given as input a model building task that builds a latent factor model.

In addition, the evaluation data task should be supported by the model under evaluation. For example, neither IMF models nor BPR models support the cold-start scenario, where predictions are required for customers and products that have not been seen during model training. Thus, the evaluation data task cannot present to the model customers or products that do not appear in the training data.

For the purposes of this dissertation, we have implemented an evaluation scheme which computes the AUC metric for the class of latent factor models into which IMF and BPR fall. The derivation of the AUC for the task of ranking has been described in section 2.8. Our implementation computes the AUC for every pair of customer ID and product ID in the evaluation dataset. The result is a DataFrame with three columns; customer ID, product ID and AUC. Loosely, the AUC in this DataFrame is a per-user-per-product AUC. To recap:

- S_{eval} is the evaluation dataset defined by some data splitting strategy. For example, we use the date-based splitting strategy as described in section 3.7.2.

- E^u is the set of products that customer u has bought in the evaluation set:

$$E^u := \{i \in I : (u, i) \in S_{\text{eval}}\}.$$

Then, the per-user-per-product AUC, AUC_{ui} is defined for all $(u, i) \in S_{\text{eval}}$:

$$\text{AUC}_{ui} := \frac{1}{|I \setminus E^u|} \sum_{j \in I \setminus E^u} \delta\left(\hat{y}_u(i) > \hat{y}_u(j)\right) \tag{3.8}$$

It is straightforward to see that AUC_{ui} is in the range $[0, 1]$. A greater value indicates that the product i from the evaluation set ranks above most of the other products, while a low value indicates that the product i ranks below most of the other products. Therefore, a higher value of AUC_{ui} indicates a better ranking quality provided by the model. We would expect AUC_{ui} to be 0.5 for a random ranking, so a model with any significant predictive capacity should do much better than 0.5. Note also that the quantity AUC_u which we described in section 2.8 is simply the average AUC_{ui} for a given u.

AUC_{ui} is a computationally intensive metric to calculate. For every customer in the evaluation data, the entire set of products is scored. Consider the scoring strategy where for a particular customer, we arrange the product factors in a $N \times f$ matrix W such that row i of the matrix contains $\vec{w}_i^T$. Then we perform the matrix product $W\vec{x}_u$ to get a vector of scores, where element i of the vector is $\vec{w}_i^T \vec{x}_u$. Finally, this vector is sorted to get a ranking of the products. The time-complexity of this algorithm is dominated by the sort, which is $O(N \log N)$. We do this for every customer in the evaluation dataset, and the complexity increases to $O(M_{\text{eval}} N \log N)$, where M_{eval} is the number of customers in the evaluation dataset. Often, there are many more customers than products, which means that M_{eval} could be comparable in size to N. Thus the overall complexity of the algorithm is in the realm of $O(N^2 \log N)$. This complexity has poor scaling properties. It would be advantageous if we could keep the complexity in the realm of $O(N \log N)$, which has acceptable scaling properties.

It turns out that by making some assumptions about the size of the product factor matrix W, we can achieve quasi-linear scaling of the algorithm. Specifically, by parallelising the matrix multiplication and sort, we can achieve significant speedup over the $O(N^2 \log N)$ algorithm described above. However, this requires that either the set of product factors or set of customer factors fit comfortably in the memory of all of the machines in the cluster (Farahat, 2015). If we assume that the number of products is much smaller than the number of customers, it is likely that the set of product factors will easily fit in memory for any of the machines in the cluster. Under this assumption, the Luigi task to compute the AUC_{ui}'s proceeds as follows:

1. Create two RDDs from the model output. One RDD of (customer ID, customer factor) pairs and the other of (product ID, product factor) pairs.

2. Gather the product factor RDD to the driver. At the driver, concatenate these factors to form the matrix W.

3. Broadcast this matrix to all workers.

4. Repartition the customer factor RDD by customer ID, and for each customer perform the matrix multiplication and sort operation. Since the product factor matrix has already been broadcast to each of the workers, the matrix multiplication operation does not need to shuffle data across the network. The result is an RDD of (u, L^u) pairs, where L^u is all products I ranked according to $\hat{r}_u$ — including products not within E^u.

5. Combine the evaluation DataFrame by customer ID to get an RDD of (u, E^u) pairs. Repartition the RDD by u so that the data layout matches the data layout of the RDD of (u, L^u) pairs.

6. Join the (u, E^u) pairs with the (u, L^u) to get an RDD of $(u, (E^u, L^u))$ pairs.

7. Run this RDD through the algorithm represented by the pseudo-code of algorithm 5. Convert the result to a DataFrame with a column for each of customer ID, product ID and AUC_{ui}. Write this DataFrame to HDFS.

Algorithm 5 Sampling (u, i, j) triples for training the BPR algorithm.

Input: RDD of $(u, (E^u, L^u))$ pairs.
Output: RDD of (u, i, AUC_{ui}) triplets.

```
1: rdd ← RDD of (u, (E^u, L^u)) pairs.
2: triplets ← rdd.flatMapValues (E^u, L^u) ⇒
3:     aucs ← empty list.
4:     ranks ← rank of all E^u in L^u
5:     for i, rank ∈ E^u, ranks do
6:         ranksAbove ← number of elements in ranks less than rank.
7:         auc ← (rank−ranksAbove) / (|L^u|−|E^u|)
8:         aucs.append(i, auc)
9:     end for
10:    aucs
11: triplets ← triplets.map((u, (i, auc)) ⇒ (u, i, auc))
```

3.9.2 Evaluation Summaries

One of the advantages of having a per-customer-per-product AUC as described above is that we get the flexibility to aggregate the AUC in different ways. For example, to get the quantity, AUC_u described in section 2.8, we could average AUC_{ui} for each customer. The purpose of

the summary tasks is to use this data to produce summaries and visualisations that can assist the data scientist in making informed decisions about which models are most promising. The summary tasks provide a software interface into which programmers can "plug" different aggregation strategies. For example, we implement a summarisation and visualisation technique that shows the distribution of the AUC in various groups of customers. The groups are defined by the number of training examples seen for that customer. Then for each group, we plot the distribution of AUC_u, and provide an associated numeric summary in tabular form. The plots are standard box plots and the summary includes following information:

- *count* of pairs of customer and product per group.
- *mean* AUC_u per group. This is generally what is reported in the literature (Zhang et al., 2014; Kanagal et al., 2012; Ahmed et al., 2013), and is defined as:

$$\mathrm{AUC} = \frac{1}{|S_{\mathrm{eval}}|} \sum_{(u,i) \in S_{\mathrm{eval}}} \mathrm{AUC}_{ui}$$

- *stddev* of AUC_u per group.
- *min* AUC_u per group.
- 2nd *percentile* AUC_u per group.
- 25th *percentile* AUC_u per group.
- 50th *percentile* AUC_u per group.
- 75th *percentile* AUC_u per group.
- 98th *percentile* AUC_u per group.
- *max* AUC_u per group.

This information gives users of the RS a clear picture of how their models perform for different groups of users who display different buying behaviours. We have found it useful for finding optimal models in two specific contexts. In the context of cross-validation, summary information is used to find the best set of parameters for a specific type of model. In the context of model testing, summary information is used to select the most promising model from a variety of model types.

Cross-validation and model testing are implemented as sets of Luigi tasks stitched together in a pipeline. These pipelines are Python applications which set up the top-level Luigi tasks, then run them. In addition to the cross-validation and model testing pipelines, there is the model publishing pipeline. In the following sections, we describe each of these pipelines.

3.9.3 Cross-Validation Pipeline

The cross-validation pipeline evaluates models of the same type across a set of parameters. Figure 3.16 shows the tasks comprising the cross-validation pipeline. The cross-validation task takes parameters that specify the type of summary task, type of evaluation task, type of model building task, type of training data task, type of validation data task, and a dictionary specifying the model parameter search space.

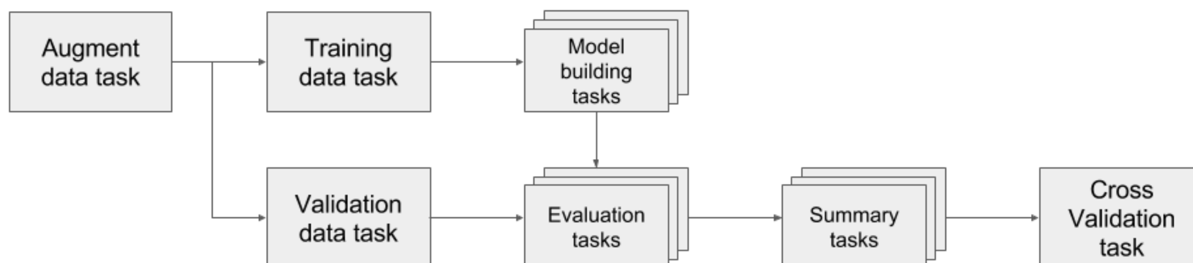


Figure 3.16: Cross-validation pipeline.

The cross-validation task then creates a set of summary tasks by "exploding" the parameter search space. This concept is shown in figure 3.17. The programmer specifies the type of model task and the parameter search space. The parameter search space is a dictionary where each dictionary key is the name of a model parameter and the value is a list of parameter values. In this example, the model type is BPR and the there are four combinations of the given parameters. This results in four summary tasks, each with its own model task and a single set of parameters. The summary tasks then set up the downstream evaluation tasks, which in turn set up the model tasks and the various data splitting tasks. The result of the cross-validation task is a set of model evaluation summaries as described in section 3.9.2.

3.9.4 Model Testing Pipeline

Once a good set of parameters has been found for a specific model type, the model can be compared with other models using a test dataset. The model testing pipeline caters for this use-case. The diagram in figure 3.18 shows the dependency graph for the model testing pipeline. Like the cross-validation task from section 3.9.3 the testing task takes parameters that specify the type of summary task, type of evaluation task, type of model task and type of training data task. Unlike the cross-validation task, however, the testing task only takes one set of model parameters, namely the best performing set of parameters from the cross-validation pipeline. Additionally, instead of a validation data task, it takes a test data task for evaluating models. The testing task then sets up the summary task, which in turn sets up all the downstream tasks. The result is a model summary as described in section 3.9.2.

The explicit support in our RS for multiple models and for model evaluation summaries sets

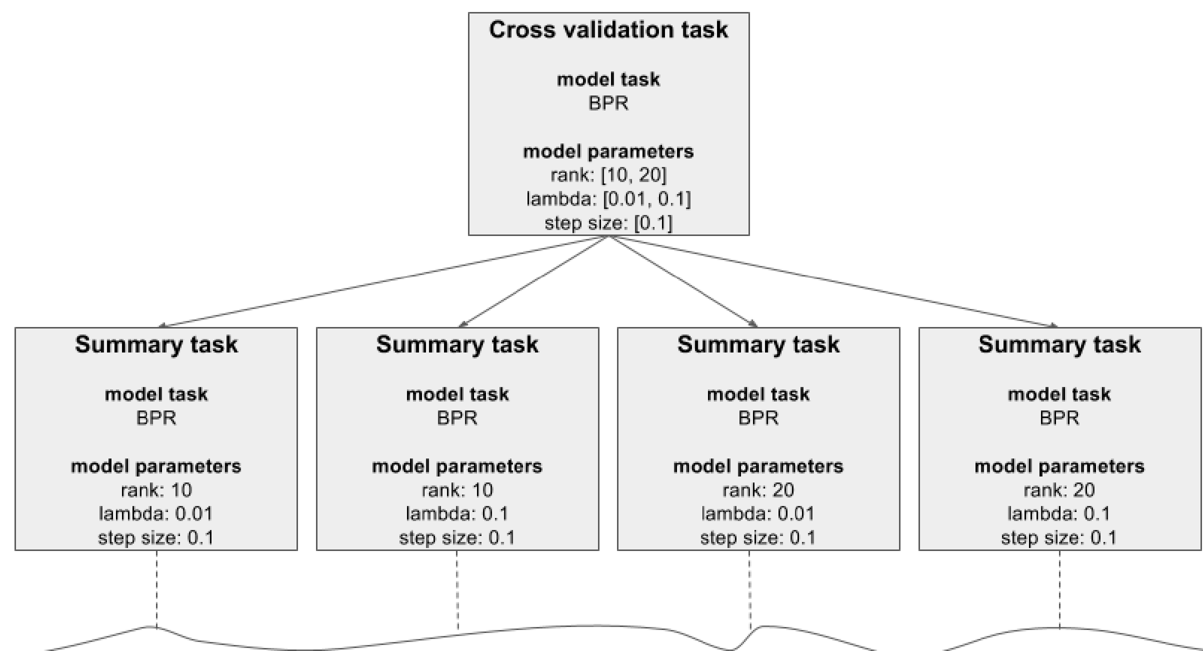


Figure 3.17: The parameter search space is converted into a summary task for each combination of parameters.

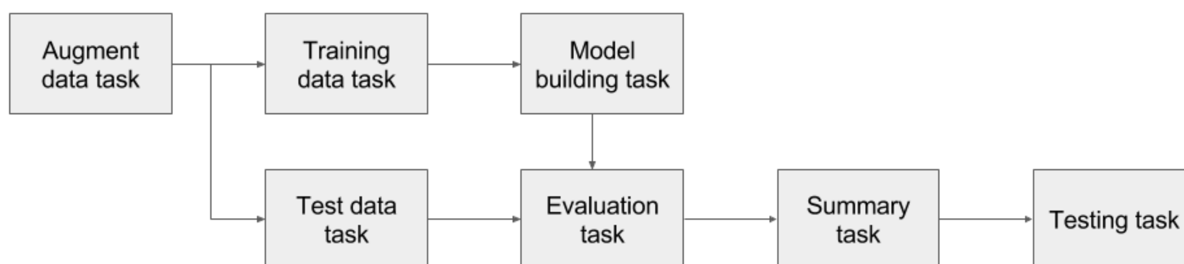


Figure 3.18: Model testing pipeline.

it apart from similar systems described in the literature. Velox and Oryx have explicit support for model management, while model management in TencentRec is implicit.

For Oryx, this comes in the form of a cross-validation step in the batch model retraining layer, which automatically selects model with the best cross-validation score. Note that Oryx has support for only one model, and thus has no notion of the model testing describe in section 3.9.4.

Velox contains a similar cross-validation mechanism to Oryx, and in addition maintains per-customer aggregates of errors. If these errors fall below a certain threshold, the model is retrained. It also has no concept of testing multiple models against one-another.

TencentRec maintains its model implicitly in that every new piece of data triggers an update to the model. However, in TencentRec there is no capability to use historical data to tune model parameters with cross-validation, or to compare different models with model testing.

None of the systems described in section 2.9 recognise the need for consumable summaries of model evaluation. As a result, the data scientist who uses these systems is burdened with performing ad-hoc analysis of model performance outside of the system. In contrast, our design is driven by the argument that these activities are better performed with support from the RS itself.

3.10 Model Publishing

The model publishing framework is comprised of two main applications. The first is the model publishing pipeline, which implements a Luigi task interface for publishing models to a location on HDFS. The second application, called the parameter server, is web service that allows users to download models in their raw form.

The RS provides a single Luigi task interface for publishing models built by the model building framework. This is shown in figure 3.19 as the model publishing task contained in the dotted block. The task depends on a model building task from the model building framework and takes an additional parameter specifying the directory on HDFS to which the built model should be published. The intention for the interface is that it aggregates the built model to one file on HDFS. It then calls an endpoint in the parameter server, which notifies the parameter server where on HDFS the model is stored. The parameter server can then stream the model to users who request it. The specifics of the model aggregation are dependant largely on how one intends to serve the model. Next, we describe the implementation of the model publishing task that we provide for the LF models described in this dissertation.

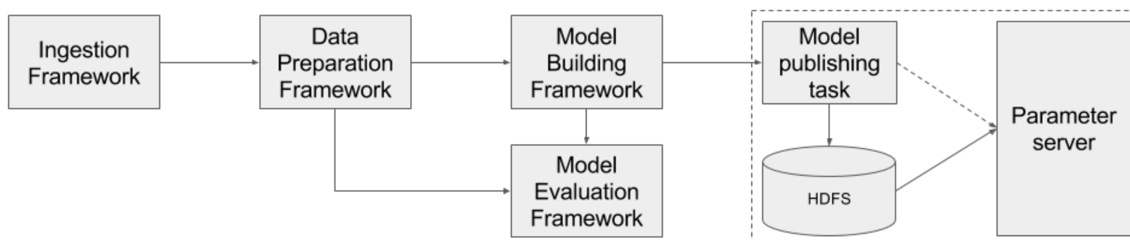


Figure 3.19: Model publishing framework.

3.10.1 Model Publishing Pipeline

The model publishing pipeline that we have implemented is shown in figure 3.20. The pipeline builds a model from the full augmented data, aggregates the model into one file on HDFS, and sends a message to the parameter server informing it of the location of the model on HDFS. Although the form of the aggregated model parameters is largely dependent on the type of model being served, the aggregated models are always stored in a Python "pickle" file. "Pickling" is

the process of serialising a Python object hierarchy into a byte stream. Therefore, a pickle file is one or more Python objects represented as bytes written to disk.

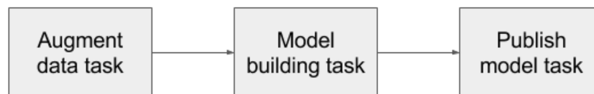


Figure 3.20: Model publishing pipeline.

For the BPR and IMF models the pickle file contains one object for each of the user factors and a number of objects for the matrix of product factors. The product factors are split into blocks where each block represents the matrix of product factors belonging to the same product department. For example, all the factors for Toys end up in the same matrix of product factors. This format suits the recommendation process for these models, where a customer factor, $\vec{x}_u$ is multiplied with the product factor matrix W to find a set of good recommendations.

3.10.2 Parameter Server

The parameter server is a Python web application that streams recommendation models from HDFS to users upon their request. The model publishing pipeline pushes trained models to this application by querying one of its endpoints with the model name and a path on HDFS. The expectation is that this path points to a file on HDFS containing the aggregated model parameters corresponding to the name of the model.

To download model parameters, a user issues a request to the parameter server with the model name as a parameter. The parameter server then streams the model pickle file from HDFS to the user. In our case, the user is usually the model serving framework, which we describe in the following section.

3.11 Model Serving

The purpose of the model serving framework is to serve recommendations to users of the RS. It is comprised of a Python application called the recommendation service. This is represented by the components enclosed by the dotted box in figure 3.21. The functionality of the recommendations service is split into two components. The first is the model importer, which downloads models from the parameter server. The second is the model server, which contains a model-specific endpoints for retrieving recommendations for individual customers.

At the start of this chapter, we described the desirable characteristics desirable of a successful RS. These included scalability, fault tolerance and low latency. These characteristics are especially relevant for the recommendations server. Outage or slowness of the server would

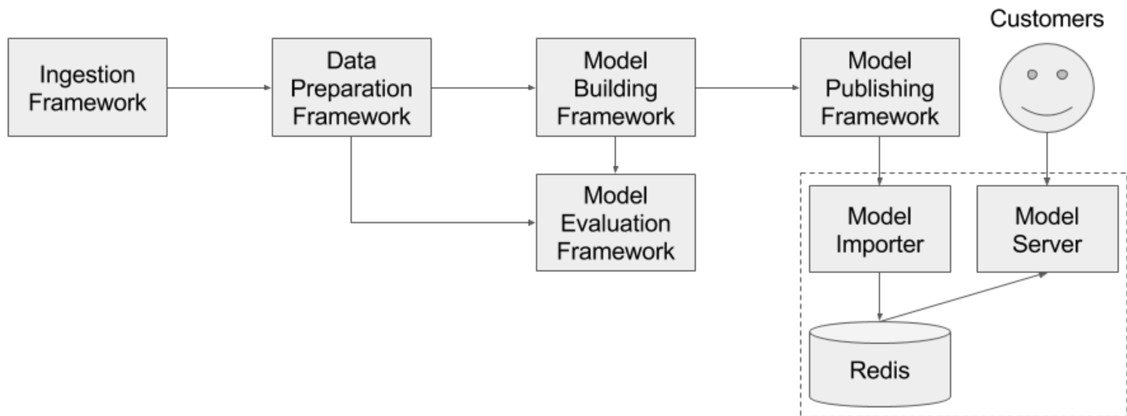


Figure 3.21: Model serving framework.

immediately be noticed by customers who expect a consistent recommendation experience. Thus, a number of design choices have been made for the recommendations service to achieve these goals, including: using a fast, scalable in-memory database called Redis for storing model parameters; parallelising computation of recommendations for speed, and allowing additional recommendation service instances to be added without affecting other running instances. In the remainder of this section, we introduce Redis, then described the functioning of the two components of the recommendations service.

3.11.1 Redis

Redis[8] is an open source, in-memory key-value data store. Known for its performance, it is often used as a database, cache or messaging system. In our application, we use it to store simple byte-strings, but it supports data structures such as hashes, lists, sets, sorted sets, bitmaps, hyperloglogs and geospatial indices. Redis supports scale and fault tolerance with a built-in cluster mode where data is automatically sharded and replicated across multiple Redis nodes. In addition to its speed, scalability and fault tolerance characteristics, Redis also exposes a simple programming interface for Python. In order to achieve its performance, Redis works with an in-memory dataset. In this data model, keys are strings and values are arbitrary byte arrays.

3.11.2 Model Importer

The model importer inserts models from the parameter server into Redis as key-value pairs. First though, the model parameters are downloaded as a pickle file from the parameter server. As the file is downloaded, it is unpickled and the parameters are inserted into Redis. The keys

[8]`http://redis.io/`

for these parameters have the following format:

`modelName.timestamp.paramName.id`,

where each of these fields indicates:

- `modelName`: the name of the model to which this parameter belongs.
- `timestamp`: the time at which the model was published to the parameter server.
- `parameterName`: identifies whether this parameter belongs to a customer or product. For example, if the parameter is a customer factor, the value of this field would be `c`.
- `id`: the customer ID or product block ID.

The values are byte strings representing either customer factors or blocks of product factors. In addition to the key-values for model parameters, there is a master key for each model that stores the last time that model was updated.

The model importer constantly polls the parameter server for updates to model parameters. By checking the master key of a model, the importer can see whether the model stored in Redis is outdated. If it is, it will acquire a lock over the Redis keyspace and download the new model. The lock prevents multiple instances of the recommendations server from writing to the same set of keys at the same time. Once the new model has been inserted into Redis, the master key for the model can be "flipped" to the updated timestamp and the model serving framework will start serving the new model.

3.11.3 Model Server

The model server is responsible for serving recommendations for individual customers. It consists of a Python application that exposes an endpoint for each model. Thus there are two endpoints for the two models presented in this chapter; IMF and BPR. These models have the same model structure and same recommendation strategy, so instead of describing them separately in the context of the model server, we lump them together under the umbrella of latent factor models and rather describe how the model server handles this class of models.

To retrieve recommendations for a specific user from the model server, a user sends a request to one of the latent factor endpoints and specifies the customer ID for which recommendations are required. If the model under question has parameters for that customer, it will return an ordered list of products, where the most relevant product is at the front of the list. Additionally, the user can also specify the number of recommendations to retrieve for a given user. If the customer ID is not known to the model, an empty list will be returned to the user.

For serving recommendations from latent factor models, we chose to store the product factors in the memory of the application. Recall that to compute recommendations for a given user we compute the vector-matrix product $W\vec{x}$, the sort the result. Therefore for every recommendation request, it is necessary to have available one customer factor and all the product vectors. Therefore, the model server reads all the blocks of product factors from Redis into the memory of the application.

When a request is made for recommendations for a specific customer from one of the latent factor models, the corresponding customer factor is read from Redis. Threads are then spawned to compute the matrix product for each block of product factors. The level of parallelism for computation of the matrix products is therefore controlled by the number of blocks into which we split the product fact matrix W. To scale computation to a set of product factors that cannot fit into the memory of one machine, we could conceivably do the computation of the matrix product across machines, although we have not as of yet implemented this feature.

Like the model importer, the model server continuously checks that the model parameters it has stored in memory are up to date. In fact, on every request, the model server first checks if there is a newer model in Redis using the master key for the model. If there are newer parameters in Redis, the model server first completes the request, then reads the new product factors into memory.

To increase the throughput of the model server, we simply add more instances of the Python application and inform the clients of the new instances. Thus, if the load on the model server becomes too onerous for one or two instances to handle, we can simply add more without changing any code.

The design of the Model Server was inspired in large parts by Oryx, whose serving layer has a very similar design to our recommendations service. One important difference is that Oryx uses Kafka to transport models from the speed layer, which updates the model continuously as data arrives, and the batch layer, which performs complete rebuilds of the models. Kafka obviates the need for a parameter server, as the serving layer can read model parameters directly off Kafka, instead of querying a parameter server for updates. This leads to a slightly more elegant solution than the one we have implemented for the prototype described in this dissertation, and in future work, we would like to implement a similar solution.

3.12 Summary

Each of the six frameworks that comprise the RS has been designed to satisfy the characteristics of an effective RS from chapter 2. HDFS (section 3.3) and Spark (section 3.4) provide scalable and fault-tolerant storage and computation engines, respectively, for the following frameworks;

data ingestion (section 3.6), data preparation (section 3.7), model building (section 3.8), model evaluation and testing (section 3.9), and model publishing (section 3.10). Luigi (section 3.5) provides the extensible workflow system that executes activities of these frameworks, which include; data ingestion, model cross-validation and testing, and model publishing. Many aspects of the design have been inspired by various RS's described in the literature. This includes the model serving framework (section 3.11), which is responsible for low-latency serving of the recommendation models. In this framework, Redis provides scalable and robust support for model storage, while a stateless model server has been designed to serve recommendations to customers with low latency.

In the following chapter, we describe the deployment of this prototype into a real-world scenario, namely, for serving recommendations to the customers of Takealot. Through observing and using the RS in this situation, we can provide a discussion about the extent to which the RS satisfies the requirements of an effective RS.

Chapter 4

Takealot Deployment and Discussion

A significant achievement of this research has been the deployment of the RS in service of the Takealot mobile applications. In the first section of this chapter (section 4.1), we detail the deployment and show how the applications have been using recommendations generated by the RS. In section 4.2, we evaluate the RS against the six characteristics of an effective RS developed in chapter 2, which are:

- recommendation accuracy,
- low latency service,
- scalability,
- durability,
- model management, and
- extensibility.

To show how the RS supports model management, we have performed a proof-of-concept offline experiment comparing the performance of IMF to BPR on the AUC metric. Otherwise, we find, for the most part, that the RS meets these requirements. However, there are areas of the system on which future research could improve.

4.1 Deployment

The prototype has been deployed across two physical locations. Latency insensitive components of the system reside in Amazon Elastic Cloud Compute (EC2), while the latency sensitive components have been deployed in the Takealot Cape Town Data Centre (DC).

4.1.1 Amazon Elastic Cloud Compute

Data ingestion, model building, model evaluation, and model publishing occurs in EC2. Thus, the EC2 infrastructure runs the following applications:

- the ingestion pipeline (section 3.6.1),
- the cross-validation pipeline (section 3.9.3),
- the model testing pipeline (section 3.9.4),
- the model publishing pipeline (section 3.10.1), and
- the parameter server (section 3.10.2).

Figure 4.1 shows the physical layout of the machines. Five nodes run the applications in EC2. Three nodes run HDFS and Spark, and the other two nodes run Luigi and the parameter server, respectively.

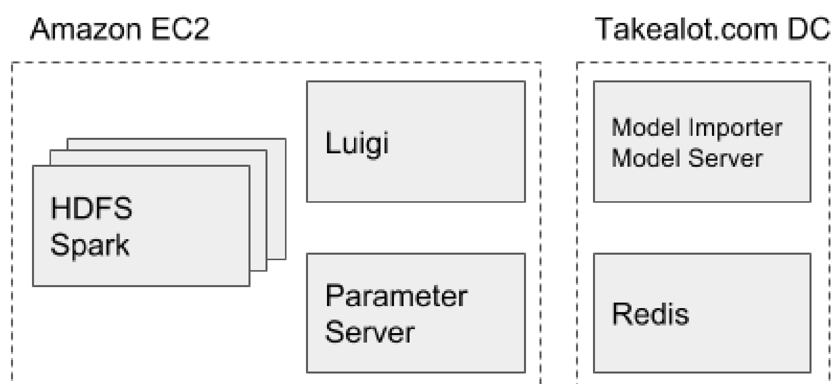


Figure 4.1: Physical layout of the machines across EC2 and the Takealot DC.

The three nodes that run Spark and HDFS co-locate HDFS DataNodes and Spark workers. As described in section 3.4, this allows Spark workers direct access to data stored by the DataNodes, without going over the network. This is important in EC2 because the network has variable performance (DripStat, 2015). Table 4.1 shows the specification of the machines in EC2. The Spark and HDFS nodes are larger compared to the others, with eight CPUs and 30.5 GiB of memory, where 1 GiB is equal to 2 exp 30 bytes. In terms of disk, the data requirements for the prototype have thus far not surpassed the gigabyte scale, so we have 400 GiB disks on each node. This gives us access to a total of 1.2 TiB of disk space, where 1 TiB is equal to 1000 GiB.

The Luigi node manages the various pipelines that comprise the activities of the RS. It primarily runs the Luigi scheduler, which offloads most of the heavy computation to Spark. In

Node	EC2 Instance Type	CPU	Memory (GiB)	Disk (GiB)
Spark/HDFS	r3.xlarge	8	30.5	400
Luigi	m3.large	2	7.5	32
Parameter Server	m3.medium	1	3.75	4

Table 4.1: Specifications for the various nodes in EC2.

addition, it stores a small amount of pipeline task history. As a result, the CPU, memory and disk requirements are significantly lower than that of the Spark and HDFS nodes. The ingestion pipeline and model publishing pipeline are run once a day in an effort to keep models fresh. There is also an inteface that runs in a Jupyter notebook (Pérez and Granger, 2007), which allows the data scientist to manually execute the cross-validation and model testing pipelines, and explore the results.

The final node in table 4.1 is responsible for running the parameter server. It handles occassional requests for new models from the Takealot DC, but the load is not cumbersome. As a result, the specifications of this node are the lightest of the EC2 infrastructure.

We have chosen to deploy these services in EC2 rather than the Takealot DC because EC2 provides a number of advantages that the DC cannot provide:

- Massive scale. As the amount of data ingested into the system grows, there will be a requirement for additional HDFS and Spark nodes. The Takealot DC does not have capacity to add nodes the size of the Spark and HDFS nodes. On the other hand, EC2 is designed specifically to provide this kind of scalability.
- Elasticity in the number of compute nodes. Some jobs may require significantly more resources than others. For example, we have noted in section 3.8.2 that BPR requires more memory than IMF. In this case, we might like to start larger Spark nodes for BPR only, then destroy them after the model has been built. EC2 provides the ability to do this with a simple programming interface that can be manipulated from within Luigi job. The RS currently does not take advantage of this feature, but it will be useful in future.

The main drawback of EC2 is latency. A heuristic latency test[1] reveals the median latency from Cape Town to our EC2 infrastructure is around 600 ms. This is a considerable delay if we expect to respond to a customer during the window of interactivity. As a consequence, we cannot run the services that are responsible for generating recommendations, namely the model server, in EC2. Thus, we have placed the model server in the Takealot DC. Fortunately, the

[1] `https://cloudharmony.com/speedtest-for-aws`

model server is not as computationally intensive as the model building framework, so resources are less of a concern.

4.1.2 Takealot Data Centre

Two nodes comprise the RS infrastructure in the Takealot DC; the model importer and model server node, and the Redis node. These are shown on the right of figure 4.1. The model importer (section 3.11.2) and model server (section 3.11.3) run on the same node. The model server is responsible for generating recommendations. For LF models, this is a fairly computationally intensive task. As a result, this node is given more resources than the Redis node, as shown in table 4.2.

The only requirement of the Redis node is that the models fit in memory. At the scale that Takealot is currently operating, both the LF models which we have implemented for this prototype fit comfortably in the 4 GiB allocated to the Redis node.

Node	CPU	Memory (GiB)	Disk (GiB)
Model Importer/Model Server	4	8	8
Redis	2	4	10

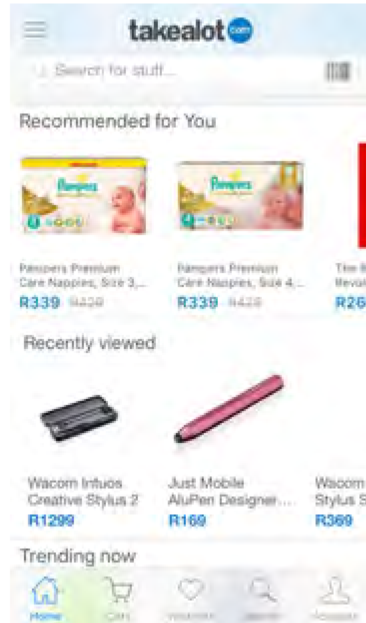
Table 4.2: Specifications for the various nodes in the DC.

Applications access the model server through the Takealot API. When the API receives a request for recommendations, it forwards the request to the model server, which responds with a list of product identifiers. The API augments the list with metadata about each of the products, including, name, price, image locations, etc., and sends it to the application. In the next section, we show how the Takealot Android and iOS applications have implemented recommendations.

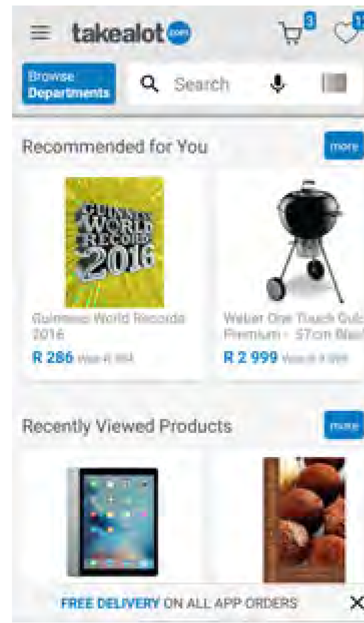
4.1.3 Takealot Mobile Applications

Figure 4.2 shows recommendations on the home screen of the Takealot Android and iOS applications. The experience is similar to the Netflix and Amazon recommendations discussed in section 2.4. The recommendation strips read, “Recommended for You,” indicating to the customer that the following recommendations have been personalised to their tastes.

These applications are good examples of the challenges in mobile, where screen “real-estate” is limited. At any time, the customer sees a maximum of two recommendations at any time. It is therefore critical that these recommendations are accurate. In the next section, we discuss



(a) iOS application.



(b) Android application.

Figure 4.2: Recommendations on the Takealot mobile applications.

the extent to which our prototype has satisfied the characteristics of an effective RS, of which accuracy is one.

4.2 Discussion

In this section, our aim is to evaluate the RS against the six characteristics of an effective RS we have developed in this dissertation.

4.2.1 Accuracy

A major shortcoming in this research has been the absence of an online experiment. In section 2.8 we discussed the importance of the online experiment for testing the accuracy of recommendations. These experiments are shown to be the most informative method for gauging the usefulness of recommendations to customers. Limitations in the application infrastructure at Takealot have thus far prevented the running of such an experiment. Therefore, we cannot provide a definitive measure of the accuracy of our recommendations. In section 4.2.5 we provide a proof-of-concept offline experiment, which gives us a heuristic measure of the accuracy of recommendations provided by the system.

4.2.2 Low Latency

In section 3.11.3 we described how the model server generates recommendations in response to customer requests. Figure 4.3 gives an indication of the variation in the median response time of the model server throughout the day. The median response time is most often below 105 ms, but during peak traffic times (for a small portion of the day), it can rise to around 127 ms.

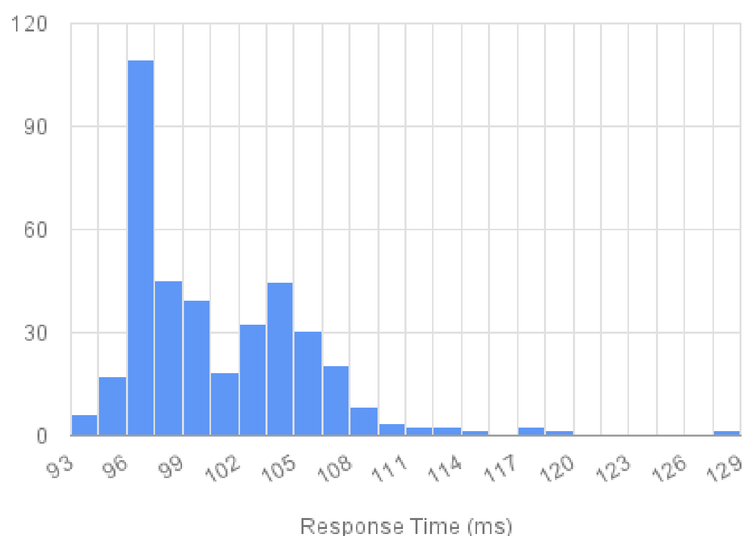


Figure 4.3: Histogram of median latency between 5 AM and 12 AM for the model server.

To give these numbers some perspective, we can compare them to those reported by the designers of Oryx. For a comparable number of features (dimensionality of factors), Owen (2015) reports that the Oryx serving layer responds in 27 ms. This is significantly lower than our system, but within an order of magintude. We suspect this disparity is mainly due to the fact that Oryx was tested on significantly more powerful hardware. More specifically, Oryx was tested on a 32-core architecture, compared to the model server hardware shown in table 4.2.

As the number of products in the system increases, the latency of the model server will increase. In section 3.11.3 we discussed the algorithm used to generate recommendations for a given customer. To recap briefly, the matrix of product factors is multiplied with the customer factor, and the result is sorted. As a result, as more products are added, this algorithm will become slower, and latency will increase. In the current design of the model server, we do nothing to mitigate this issue. There are two possible mitigation strategies which we could implement in future. First, we could take a similar approach to Oryx (section 2.9.1), where the number of products considered for recommendation is reduced. In addition, we could give the model server more powerful hardware. Adding more cores to the machine running the model

server would improve the parallelism of the matrix multiplication, thus improving improving latency. In this case, we would expect to see similar performance to Oryx.

4.2.3 Scalability

Scalability has been a central feature of our discussions throughout this dissertaion. We have utilised HDFS and Spark to provide scalability in the data storage and model building activities of the RS. Nevertheless, our implementation of BPR is one possible scaling bottleneck. The algorithm described in section 3.8.2 requires that all product factors and all customer factors fit in the memory of every Spark worker. Fortunately, at current data sizes, this is not an issue. However, as the number of customers and products increase, it is conceivable that the combined size of the factors grows beyond the memory capabilities of a single machine. To alleviate this issue, we could design an improved data partitioning scheme. For example, we could store all product factors on every machine, but partition customer factors across the cluster. Even with this strategy, the size of the product factors eventually becomes a bottleneck. At this point, we may be forced to adopt a different computational paradigm altogether for building the model. For example, we could use a parameter server model which provides scalable parameter storage at the cost of increased communication (Li et al., 2014b).

The model server is another bottleneck for scalability. With the current design, the application stores all product factors in memory. As the number of products in the system grows, the size of the product factors will eventually exceed the capability of a single machine. There are two possible strategies to mitigate this problem:

1. Increase the amount of memory available to the model server. This is a simple solution, but it is not sustainable, as we cannot indefinitely increase the memory of a machine.
2. Parallelise the execution of the algorithm. In this case, product factors are partitioned across several machines. When a request arrives at a particular node, the node begins computing recommendations of products for which it is responsible, and forwards the request to the other nodes. After completing computation, it collects results from other nodes, and performs the final sorting of products. This result is then returned to the requestor. The drawback of this design is that it is significantly more complicated than the original.

4.2.4 Durability

Fault tolerance has been another primary concern in the design of the RS. HDFS and Spark provide fault tolerance in the data storage and processing layers of the system. In the model

serving layer, Redis provides fault tolerance for data storage. Running multiple instances of the model server provides fault tolerance for model serving; should one instance fail, other instances continue to serve recommendations. In addition, model serving is isolated from failures in other pars of the system. For example, should the model building pipeline fail during computation of a model, the model server continues to serve recommendaitons using an older model. Therefore, the RS provides an appropriate level of durability for the environment in which it operates.

4.2.5 Model Management

In section 2.8 we discussed the two aspects of model management; model freshness and model effectiveness. Model freshness refers to keeping models up-to-date. Model effectiveness involves providing the data scientist with the tools to maintain the effectiveness of models over time. In particular, we have discussed "debuggable" automatic cross-validation and model testing procedures. In this section, we discuss our RS's support for these features.

We mentioned in section 4.1 that models are automtically rebuilt every day. As a result, the time at which a customer purchases a product to the time at which the RS incorporates that action is at maximum a day. This is by no means prohibitive, but ideally, the RS shoud respond to customer actions immediately. As we discussed in section 3.6.1, limitations of the broader environment preclude continuous updates to models.

We have seen that the RS provides extensive support for model cross-validation and testing (see section 3.9.1). The RS does not automate these activities, but even with manual set up and execution of the cross-validation and model testing pipelines, their outputs serve as useful tools with which the data scientist can make decisions regarding the effectiveness of various models. More specifically, the outputs of the cross-validation pipeline assist in selecting a good set of parameters for a specific model, while the model testing pipeline assists in comparing the performance of different models. To provide a concrete example of how the RS supports these activities, the remainder of this section presents a proof-of-concept experiment comparing IMF to BPR. We first describe the characteristics of the dataset that we use for the experiment, then present the parameter sets over which we perform cross-validation, and finally we discuss the results of the model testing.

The dataset consists of a log of transactions from Takealot. For the purposes of the experiment, we have excluded from the dataset products that have been bought by fewer than 10 customers. This is common practice for reducing noise due to infrequent products (Rendle et al., 2009; Shi et al., 2012b). As discussed in section 3.7.1, the RS has explicit support for this practice in the prefiltering and data augmentation stage of the ingestion pipeline. The resulting data set contains the purchase history of 496 820 customers on 55 898 products. In total, 3 426 655 transactions have been recorded.

We then separate the dataset into a training set, validation set and test set according to the time-based data splitting strategy described in section 3.7.2. For the purposes of the experiment, 90% of the data is used for training, 5% for validation, and 5% for testing. Statistics about the three data sets are shown in table 4.3.

Data Set	Transactions	Customers	Products
Training	3 158 734	492 354	55 898
Validation	130 244	51 490	24 455
Test	128 724	55 251	24 606

Table 4.3

In order to get a better understanding of which types of customers account for the greatest proportion of purchases, we place customers into groups according to the number of products they have purchased:

1. 1 to 10,
2. 11 to 30,
3. 31 to 100,
4. 101 to 300,
5. 301 to 1000, and
6. 1000 +.

Then, for each group, we count the number of purchases that appear in the training set. The results are summarized in figure 4.4. The majority of purchases are made by customers who have bought at most 10 products. As a result, many customers have a very limited number of purchases with which we can model their parameters.

A similar analysis can be performed from the persepective of products. Figure 4.5 shows the majority of purchases are of products that have been purchased between 31 and 300 times. This is a more even distribution than for the customer-based analysis. Hopefully, the relatively rich information we have about products translates to meaningful inference about what customers might be interested in. For example, even though a customer has bought only one product, it is likely that the product has also been bought by many other customers. Of these customers, there might be a few have bought many products, which informs the recommendations for the original customer.

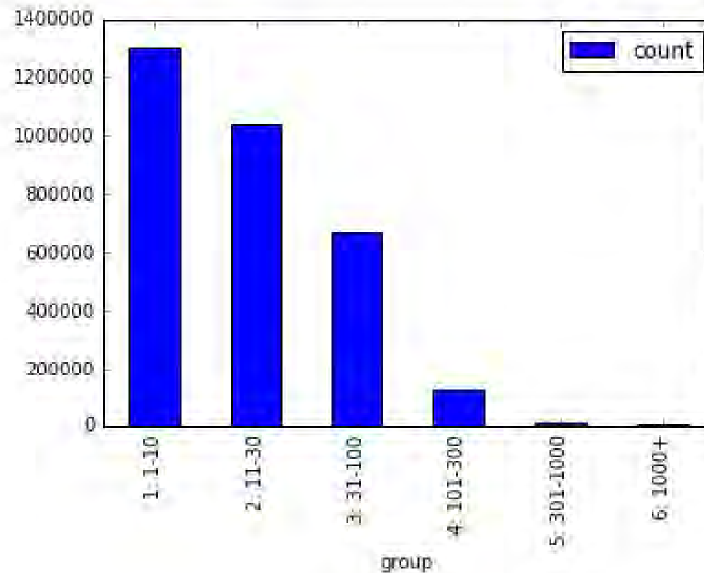


Figure 4.4: Number of purchases accounted for by each customer group in the training set.

Next, we run the cross-validation pipeline for IMF and BPR. As discussed in section 4.1 this is a manual process that is managed through a Jupyter notebook. For IMF, the cross-validation pipeline is configured to train and evaluate models with combinations of the following parameters (see section 3.8.1 for a recap of parameters for IMF):

- `rank`: 10, 20.
- `numUserBlocks`: Default (allow Spark to set this parameter).
- `numItemBlocks`: Default (allow Spark to set this parameter).
- `maxIter`: 10, 20, 30.
- `regParam`: 0.01, 0.1, 1.
- `alpha`: 80.

Similarly for BPR, we train and evaluate models with combinations of the following parameters (section 3.8.2):

- `rank`: 10, 20, 40.
- `lambda`: 0.1.
- `customerStepSize`: 1.
- `posStepSize`: 1.
- `negStepSize`: 1.

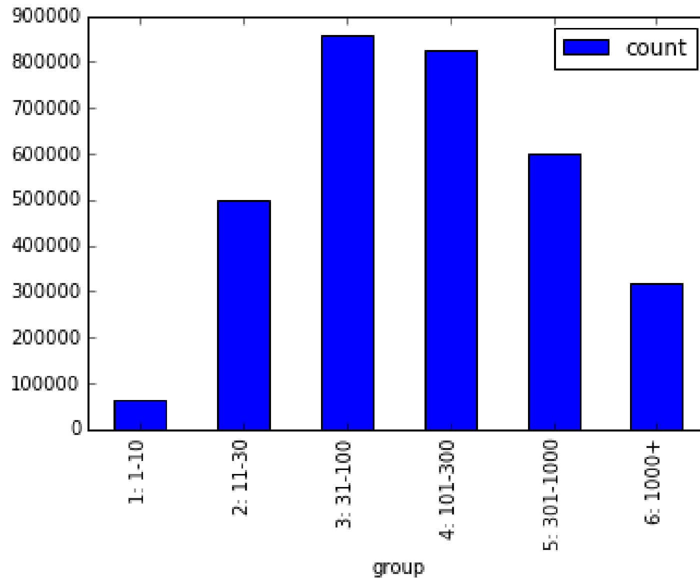


Figure 4.5: Number of purchases accounted for by each product group in the training set.

- `iters`: 5, 10.
- `initPartitions`: 12 – this is the number of cores available in the cluster.

The models are evaluated using the AUC evaluation strategy defined in section 2.8. The result of the cross-validation pipeline is a set of summaries stored on HDFS. The results for BPR and IMF are summarised in table 4.4 and figure 4.6, and table 4.5 and figure 4.7, respectively.

`lambda`	`stepSize`	`m`	`rank`	`iters`	AUC
0.1	1	200	40	10	0.7782
0.1	1	200	40	5	0.7712
0.1	1	150	40	10	0.7687
0.1	1	200	20	10	0.7679
0.1	1	150	40	5	0.7569
0.1	1	200	20	5	0.7556
0.1	1	150	20	10	0.7551
0.1	1	200	10	10	0.7507
0.1	1	150	20	5	0.7438
0.1	1	200	10	5	0.7428
0.1	1	150	10	10	0.7412
0.1	1	150	10	5	0.7283

Table 4.4: Average AUC for BPR models trained during cross-validation. Note that `stepSize` represents the values for `posStepSize`, `negStepSize` and `customerStepSize`.

The best performing parameter setting for BPR can be seen in the first row of table 4.4, according to average AUC. We can provide further evidence for this claim by looking at figure 4.6. The boxplots show the distribution of AUC_u for each parameter setting. They have been sorted from highest average AUC on the left to lowest average AUC on the right. Notice that the boxplot on the far left of the figure has a “tighter” distribution than the others. Its

lower "whisker" is slightly higher than the other whiskers, and its "box" is shorter than the other boxes. This means two things. First, the model on the left makes slightly better predictions for customers whose purchases are difficult to predict. Second, with this model, the middle quartile of customers experiences better predictions. As a result, we move forward to model testing with the parameter setting in the first row of table 4.4.

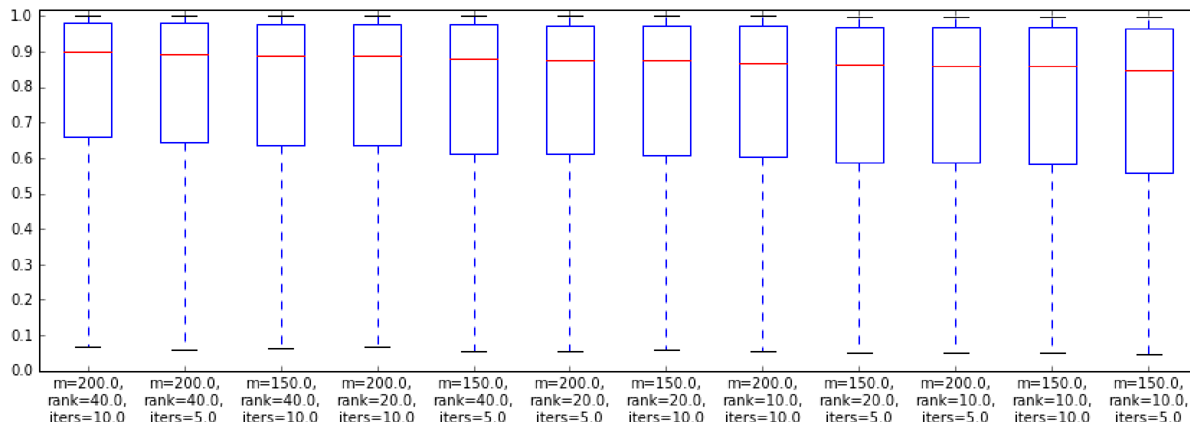


Figure 4.6: Boxplots for each of the parameter settings in table 4.4, ordered by average AUC (highest on the left, lowest on the right). The upper and lower whiskers represent the 98 th and second percentile of AUC_u, respectively.

A similar analysis can be performed for IMF using table 4.5 and figure 4.7. As a result we use the model trained with the parameters in the first row of table 4.5 for our final comparison.

alpha	regParam	maxIter	rank	AUC
80	1.00	20	10	0.7598
80	0.10	20	10	0.7583
80	0.01	20	10	0.7545
80	0.10	10	10	0.7530
80	0.01	10	10	0.7524
80	1.00	10	10	0.7496
80	1.00	20	20	0.7458
80	0.01	20	20	0.7453
80	0.10	20	20	0.7449
80	1.00	10	20	0.7395
80	0.01	10	20	0.7393
80	0.10	10	20	0.7370
80	1.00	20	30	0.7347
80	0.01	20	30	0.7327
80	0.10	20	30	0.7315
80	1.00	10	30	0.7306
80	0.10	10	30	0.7273
80	0.01	10	30	0.7257

Table 4.5: Average AUC for each set of parameters tried during cross-validation for IMF.

Before we move on, however, there is an interesting observation to be made about the behaviour of the two models. Table 4.4 shows that in general for BPR, the average AUC improves for increasing rank, i.e. number of customer and product factors. On the other hand,

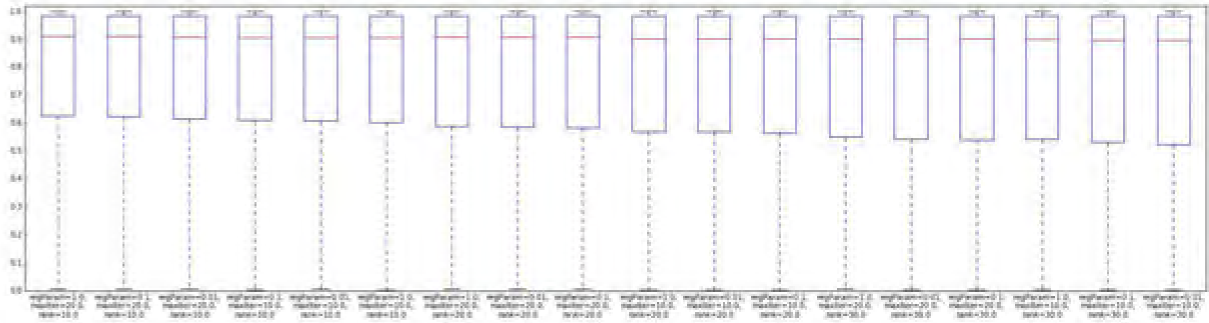


Figure 4.7: Boxplots for each of the parameter settings in table 4.5, ordered by average AUC.

table 4.5 shows the opposite behaviour. As the number of factors increases for IMF, the average AUC decreases. One possible explanation for this behaviour lies in how these two algorithms utilize data. More specifically, BPR augments the dataset through its sampling of “unseen” products for every “seen” product (see section 3.8.2). Thus, as we increase the number of factors for BPR, we can also increase the amount of data that the algorithm sees by sampling more unseen products. On the other hand, we cannot do this for IMF. Therefore, as we increase the number of factors, the ratio of data to parameters decreases, as we have no control over the number of data points which the algorithm sees. As a result, the parameters are not effectively trained and the predictive power of the model decreases.

Additionally, in figures 4.6 to 4.7, the AUC exhibits extreme variability, with many of the boxplots extending almost to zero. This would seem to indicate that, at least in some cases, the predictions are no better than random. In fact, we should expect this behaviour. As mentioned before, most customers have very little information with which we can model their parameters. Since parameters are randomly initialised, we would expect that, for some of these customers, recommendation of random products would have the same predictive power as either of the models. However, for customers who have bought more products, the model significantly out-performs random predictions. This is shown next.

Having found good sets of parameters for IMF and BPR, we set up the model testing pipeline to compare the models. In addition to making use of the model evaluation summaries as before, we use the AUC_u’s stored on HDFS to do some ad-hoc analysis of the results. We also simulate a random recommender by generating a random ranking of products for each customer purchase, then calculating the AUC for that ranking. This allows us to compare the performance of the models to a random recommender. To get a more detailed understanding of the performance of the models for different types of customers, we group results according to the purchase frequency groups described earlier. For each group, we perform a Wilcoxon signed rank test (Demšar, 2006) to determine whether the difference in AUC between the algorithms is significant. The results are summarised in table 4.6. We have also visualised these results in figure 4.8.

	1 to 10	11 to 30	31 to 100	101 to 300	301 to 1000	Overall
BPR	**0.7763***	**0.7839***	0.7585	0.7246	**0.7422**	**0.7758***
IMF	0.7395	0.7749	**0.7686***	**0.7377***	0.7361	0.7526
RANDOM	0.4984	0.5001	0.5014	0.4963	0.5167	0.4992

Table 4.6: Average AUC_u for each customer purchase frequency group. Bold numbers indicate best performance and star indicates statistical significance (p-value < 0.01).

For most customers, a recommendation from either of the models will be much better than a random recommendation. This is shown in figure 4.8, where there is an even spread of the AUC_u's around 0.5 for the random recommender, while IMF and BPR tend to spread around 0.75.

As expected, BPR outperforms IMF overall (Rendle et al., 2009). This is largely due to its superior prediction capability for customers who have seen very few training examples. Since these customers dominate the dataset, BPR does better overall. As customers buy more and more products, IMF performs better than BPR, although both models decline in performance. One possible explanation for this is that there is more structure in the purchases of customers who are new to online shopping. For example, they may tend to buy popular products. As customers explore the catalogue more, it is conceivable that their behaviour becomes more difficult to predict.

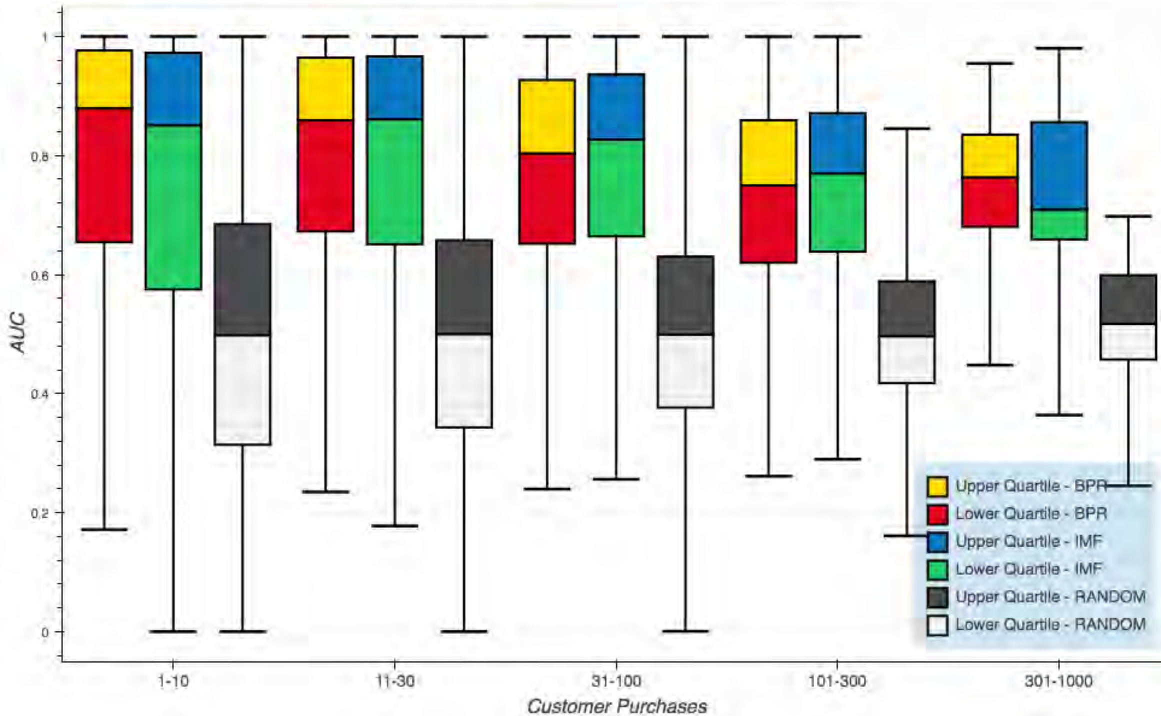


Figure 4.8: Boxplots comparing the distributions of AUC_u for IMF and BPR over the various customer purchase frequency groups.

4.2.6 Extensibility

We have provided extensible software interfaces for almost all the components of the RS. In the data ingestion, data preparation, model building, and model evaluation and testing frameworks, Luigi provides the foundations for exensible tasks. An extensible object oriented (OO) design in the model serving application makes it simple to add new methods for serving recommendations.

To extend the Luigi tasks, the programmer inherits from the task interface, and implements the logic of the task. Depeding on the specific interface, the framework provides varying degrees of abstraction. For example, the model building interface provides abstractions for input and output mechanisms. To implement the IMF and BPR model building tasks that we discussed in section 3.8, we extend the model interface. We add the custom parameters required by the model, and implement the algorithm to train the model. The interface provides access to the input data, and when training is complete, we return a DataFrame, which is then written to HDFS. Thus, the interface hides the complexity of reading and writing input and output.

Other task interfaces provide similar abstractions for input and output data. One counterexample is the data ingestion framework. For this interface, we need only specify an input method that supports the Spark DataFrame API. Reading data and then writing it to HDFS is handled transparently by the interface.

In the model serving application, we provide a base class which programmers can extend to implement new recommendation techniques. The base class provides abstractions for reading information from Redis according to the format specified in section 3.11. As a result, the programmer need only specify the name of their model – procedures for reading data from Redis is handled by the framework.

Certain classes of recommendation techniques share methods for generating recommendations. For example, the algorithm for computing recommendations using models trained with IMF and BPR is exactly the same. Thus, the model server provides abstractions to cater for this characteristic. For IMF and BPR we provide a LF model abstraction. This class provides a method for computing the matrix product and sort required for generating recommendations. Thus, to implement model serving for IMF and BPR, we inherit this class and specify a model name; everything else is handled by the various abstractions.

4.3 Summary

The deployment of the RS in service of the Takealot mobile applications means that it is serving recommendations to real customers. Despite the absence, in this dissertation, of a formal verification of the efficacy of recommendations generated by our RS, we argue that it is, in

fact, adding value to the Takealot offering. From the perspective of the customer, the recommendations are responsive. In section 4.2.2, we showed that the system responds to requests for recommendations within the window of interactivity. From an engineering perspective, the system provides acceptable levels of fault-tolerance and scalability, as described in section 4.2.4 and section 4.2.3. From the perspective of the data scientist, the support for model management activities is useful for examining the performance of various models (section 4.2.5). In addition, for most of the activities supported by the RS, the system is easy to extend (section 4.2.6).

However, there are areas in which the RS could improve. Next, in the concluding chapter of this dissertation, we provide details for possible directions of future research to improve the RS.

Chapter 5

Conclusion

RS's have become a central feature of the e-retail platform. For the e-retailer, RS's provide significant economic benefits. For example, in 2006, Amazon attributed 35 % of their revenue to recommendations. If we consider Amazon as a yardstick for the entire industry, recommendations generate tens of billions of dollars of value every year. This is a compelling argument for any growing e-retailer to invest in a RS. As a result, the purpose of this research has been to develop a prototype RS for the South African e-retailer, Takealot.

Nevertheless, building an effective RS is a daunting endeavour. First, we must identify the characteristics that comprise an effective RS, and second, we must design and implement systems that satisfy these characteristics. Fortunately, in the last decade, there has been an explosion of interest in RS's from an academic perspective, due in part to the Netflix Prize competition of 2006. This allows us to pull together research on different aspects of an RS to generate characteristics of an effective RS. This is the first contribution of our research; a set of six characteristics which all effective RS's should possess:

- Accuracy; recommendations should be useful to the customer (section 2.4).
- Low latency; customer requests should be served within the window of interactivity (section 2.4).
- Durability; the system should be robust to internal failures (section 2.4).
- Scalability; the system should exhibit graceful scaling properties in the face of increasingly large input datasets and model parameter sets (section 2.5 and section 2.7).
- Model management; models should remain up to date and effective (section 2.8).
- Extensibility; software interfaces that allow the data scientist to implement new techniques are an important feature of a RS. In particular, extensible interfaces should be available for; model building, data splitting, and calculating and summarising evaluation metrics.

The second contribution of this research is a description of an RS capable of serving recommendations in an actual e-retail environment (chapter 3). The design of our RS is an attempt to embody the characteristics described above. Scalable and robust technologies support the various features of the system. Data ingestion, data preparation, model building, and model evaluation activities are supported by HDFS and Apache Spark, which provide scalable data storage and computation, respectively. In the model serving layer, Redis provides fast, scalable and fault tolerant storage for model parameters.

Luigi provides the system with appropriate extensibility characteristics. The concept of tasks in Luigi allows the system to abstract trivial details of the implementation away from the programmer, allowing them to focus on the details of the various techniques for generating or evaluating recommendations. This has been particularly useful in the data ingestion, data preparation, model building and model evaluation components of the system. In the model serving layer, we have implemented Python interfaces which abstract away the details of retrieving and modelling model parameters, allowing the programmer to focus on generating recommendations.

An important feature of our RS is the explicit support for decision support in the form of model evaluation summaries. Model management activities are a central feature of an RS, but many systems do not provide the data scientist with the tools necessary to make informed decisions about which models perform well, without the data scientist having to perform laborious manual analysis. In section 4.2.5, we presented a proof-of-concept experiment comparing two popular methods for generating recommendations – which we implemented for this dissertation – namely, IMF and BPR. We showed how the model evaluation summaries are useful for choosing sets of parameters that give the best performance during cross-validation, and for comparing model performance during model testing. In our experiment, BPR performed better, with an overall average AUC of 0.7758.

5.1 Limitations

A significant limitation of this research has been the absence of an online experiment. We discussed in section 2.8 the importance of testing recommendation techniques online to evaluating their true impact on customer behaviour. Therefore, the absence of such a test limits our ability to assess the accuracy of the system.

In addition, the system has some technical weaknesses, most prominently in the responsiveness and scalability of the system. In section 4.2.2, we showed that as the number of products in the system increases, the latency of requests for recommendations will increase. In the current design of the system, we do nothing to mitigate this problem. With respect to scalability, we

presented two bottlenecks in section 4.2.3. The first is our implementation of BPR, where all product and customer factors are kept in the memory of each Spark worker. Second is the model server, where we store all product factors in the memory of the application. As a consequence, as the number of products for which the system needs to recommend increases, there comes a point at which the size of the product factor matrix exceeds the capacity of a single machine.

Another technical limitation of the system is the inability to support continuous ingestion of interaction data. This is an important feature in the modern RS, as shown by Oryx, Velox, and TencentRec (Owen, 2015; Crankshaw et al., 2014; Huang et al., 2015). Continuous ingestion allows the system to maintain model freshness more effectively than batch ingestion. Unfortunately, limitations of the environment at Takealot have thus far prevented us from developing this feature for our RS.

5.2 Future Research

There are a number of possible directions for future work on our RS. In particular, we would like to address the above weaknesses of the system. To evaluate the accuracy of recommendations, an online testing feature should be built. The literature on this subject is fairly broad, and a number of large online companies have published their approaches to performing online experiments. For example, (Xu et al., 2015) describe their system, XLNT, which powers thousands of experiments for the professional social network, LinkedIn.

To improve latency in response to requests for recommendations, we could implement a hashing strategy similar to the one present in Oryx (Owen, 2015). In this scheme, we prune the number of product factors according to a user-specified sample rate. As product factors are created or updated, they are hashed into a number of partitions. At serving time, we only consider a random subsample of the partitions. As a result, even when the space of possible products becomes very large, we can consider a constant number of products for recommendation, so our recommendation time remains fixed. This comes at the cost of decreased recommendation accuracy, as we neglect a portion of the product space.

Resolving the scalability issues of our BPR implementation can be achieved in a number of ways. First we could implement a more careful partitioning scheme for product and customer factors. For example, we could store all product factors on every machine, but partition customer factors across the cluster. Even with this strategy, the size of the product factors eventually becomes a bottleneck. In addition, this defeats the purpose of Splash, which strives to abstract the difficulties parallel stochastic learning from the implementation of the algorithm. Thus, we may be forced to adopt a different computational paradigm altogether for building the model. For example, we could use a parameter server model which provides scalable parameter storage

at the cost of increased communication (Li et al., 2014b).

In section 4.2.3 we provide some direction for improving scalability of the model serving application. Inspired by the model serving strategy of Velox (Crankshaw et al., 2014), the idea is to partition product factors across the memory of several machines. When a request arrives at a particular node, the node begins computing recommendations of products for which it is responsible, and forwards the request to the other nodes. It then collects results from other nodes, and performs the final sorting of products.

We would like also to explore the use of continuous data integration strategies for keeping models fresh. For the latent factor models described in this dissertation, Owen (2015) and Crankshaw et al. (2014) give proposals for online update strategies that approximate a full update of the model. In these strategies, the model is continually updated as new data arrives, and occasionally the entire model is retrained on the full dataset to maintain prediction accuracy.

Besides online update of current models, we would also like to explore hybrid recommendation models that take advantage of diverse datasets. For example, we might implement the TF method described in section 2.7.3 by Kanagal et al. (2012), or one of its extensions described by Zhang et al. (2014) and Ahmed et al. (2013). In addition to different models, the RS should also be extended to support different evaluation metrics. Apart from AUC, there are many other useful metrics for measuring the offline performance of RS's, including MRR (Shi et al., 2012b), MAP (Shi et al., 2012a) and NDCG (Shani and Gunawardana, 2011). These extensions would make the RS a much more complete system.

5.3 Acknowledgements

The RS presented in this dissertation has not been built in isolation. The design process as a whole has enjoyed the continued support of the Machine Learning team at Takealot, which at the time of writing consists of Maien Hamed, Carl Scheffler, Philip Sterne, and myself. In terms of implementation, the parameter server described in section 3.10.2 was built by Philip Sterne.

References

Deepak Agarwal and Bee-Chung Chen. Regression-based latent factor models. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 19–28. ACM, 2009.

Neeraj Agrawal. The saas adventure, 2015. URL `http://techcrunch.com/2015/02/01/the-saas-travel-adventure/`.

Amr Ahmed, Bhargav Kanagal, Sandeep Pandey, Vanja Josifovski, Lluis Garcia Pueyo, and Jeff Yuan. Latent factor models with additive and hierarchically-smoothed user preferences. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 385–394, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1869-3. doi: 10.1145/2433396.2433445. URL `http://doi.acm.org/10.1145/2433396.2433445`.

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742797. URL `http://doi.acm.org/10.1145/2723372.2742797`.

Nora Aufreiter, Josh Liebowitz, and Kelly Ungerman. The secrets of amazon: Lessons for multichannel retailers, 2012. URL `http://www.slideshare.net/McK_CMSOForum/amazons-secret-sauce`.

Peter Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2015. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-206.html`.

Nicola Barbieri, Giuseppe Manco, and Ettore Ritacco. Probabilistic approaches to recommendations. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 5(2):1–197, 2014.

R. M. Bell and Y. Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 43–52, Oct 2007. doi: 10.1109/ICDM.2007.90.

J. Bennet and S. Lanning. The netflix prize. In *Proceedings of KDD Cup and Workshop 2007*, 2007.

Greg Bensinger. Amazon's new secret weapon: Delivery lockers, 2012. URL `http://www.wsj.com/articles/SB10000872396390443545504577567763829784538`.

Erik Bernhardsson and Elias Freider. *Luigi*. Spotify, New York, NY, USA, 2015. URL `http://luigi.readthedocs.org/en/stable/`.

David M. Blei, Thomas L. Griffiths, and Michael I. Jordan. The nested chinese restaurant process and bayesian nonparametric inference of topic hierarchies. *J. ACM*, 57(2):7:1–7:30, February 2010. ISSN 0004-5411. doi: 10.1145/1667053.1667056. URL `http://doi.acm.org/10.1145/1667053.1667056`.

Broadband Commision. The state of broadband 2014: broadband for all. Technical report, Broadband Commision, Geneva, Swizterland, September 2014.

Centre for Retail Research. Online retailing: Britain, europe, us and canada 2015. RetailMeNot, Inc., May 2015.

Colby Ronald Chiles and Mauaritte Thi Dau. An analysis of supply chain best practices in the retail industry with case studies of wal-mart and amazon.com. Master's thesis, Massachusetts Institute of Technology, 2005.

Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. 2014. URL `http://arxiv.org/abs/1409.3809`.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`.

Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, 7:1–30, December 2006. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1248547.1248548`.

Frank Dernoncourt. Machine learning: What is an intuitive explanation of auc, April 2015. URL `https://www.quora.com/Machine-Learning-What-is-an-intuitive-explanation-of-AUC`.

Scott Devitt, Andrew Ruud, David Gober, Joseph Parkhill, Kimberly Greenberger, Mark Wiltamuth, Richard Ji, Philip Wan, Timothy Chan, Robert Lin, Angela Moh, Geoff Ruddell,

Edouard Aubin, Anisha Singhal, Louise Singlehurst, Edward Hill-Wood, Nicholas Ashworth, Maryia Berasneva, Loredana Serra, Tom Kierath, Crystal Wang, Tetsuro Tsusaka, Zachary Arrick, and Nishant Verma. ecommerce disruption: A global theme. Technical report, Morgan Stanley, January 2013.

Richard Dobbs, Yougang Chen, Gordon Orr, James Manyika, Michael Cui, and Elsie Chang. China's e-tail revolution: Online shopping as a catalyst for growth. Technical report, McKinsey Global Institute, March 2013.

DripStat. The highly variable network performance of amazon ec2, 2015. URL http://blog.dripstat.com/the-highly-variable-network-performance-of-amazon/.

Ayman Farahat. 26 trillion app recommendation using 100 lines of spark code. Spark Summit Europe, October 2015. URL http://www.slideshare.net/SparkSummit/26-trillion-app-recomendations-using-100-lines-of-spark-code-ayman-farahat.

R. Fletcher and H. Crawford. *International Marketing: An Asia-Pacific Perspective*. Pearson Higher Education AU, 2013. ISBN 9781442561250. URL https://books.google.co.za/books?id=8iziBAAAQBAJ.

Hadoop Development Team. *HDFS User Guide*. Apache Software Foundation, Los Angeles, USA, 2015. URL https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.

Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin K. Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C.j.c. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. 2013. URL http://media.nips.cc/nipsbooks/nipspapers/paper_files/nips26/631.pdf.

Hortonworks. What hdfs does, 2016. URL http://hortonworks.com/hadoop/hdfs/. [Online; accessed 13-January-2016].

Diane J. Hu, Rob Hall, and Josh Attenberg. Style in the long tail: Discovering unique interests with latent variable models in large scale social e-commerce. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1640–1649, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623338. URL http://doi.acm.org/10.1145/2623330.2623338.

Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 263–272. IEEE, 2008.

Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. Tencentrec: Real-time stream recommendation in practice. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 227–238, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742785. URL `http://doi.acm.org/10.1145/2723372.2742785`.

Christopher C Johnson. Logistic matrix factorization for implicit feedback data. In *NIPS 2014 Workshop on Distributed Machine Learning and Matrix Computations*, 2014.

Bhargav Kanagal, Amr Ahmed, Sandeep Pandey, Vanja Josifovski, Jeff Yuan, and Lluis Garcia-Pueyo. Supercharging recommender systems using taxonomies for learning user purchase behavior. *Proceedings of the VLDB Endowment*, 5(10):956–967, 2012.

Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online controlled experiments at large scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1168–1176, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2174-7. doi: 10.1145/2487575.2488217. URL `http://doi.acm.org/10.1145/2487575.2488217`.

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, pages 426–434, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-193-4. doi: 10.1145/1401890.1401944. URL `http://doi.acm.org/10.1145/1401890.1401944`.

Yehuda Koren. The bellkor solution to the netflix grand prize. *Netflix prize documentation*, 81: 1–10, 2009.

Yehuda Koren and Robert Bell. Advances in collaborative filtering. In *Recommender systems handbook*, pages 145–186. Springer, 2011.

Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014a. ACM. ISBN 978-1-4503-3252-1. doi: 10.1145/2670979.2670985. URL `http://doi.acm.org/10.1145/2670979.2670985`.

Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, Berkeley, CA, USA, 2014b. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685095.

Ji Liu, Stephen J. Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. 11 2013. URL http://arxiv.org/abs/1311.1873.

Pasquale Lops, Marco Gemmis, and Giovanni Semeraro. *Recommender Systems Handbook*, chapter Content-based Recommender Systems: State of the Art and Trends, pages 73–105. Springer US, Boston, MA, 2011. ISBN 978-0-387-85820-3. doi: 10.1007/978-0-387-85820-3_3. URL http://dx.doi.org/10.1007/978-0-387-85820-3_3.

James Manyika, Armando Cabral, Lohini Moodley, Safroadu Yeboah-Amankwah, Suraj Moraje, Michael Chui, Jerry Anthonyrajah, and Acha Leke. Lions go digital: The internet's transformative potential in africa. Technical report, McKinsey Global Institute, November 2013.

Massmart. Reviewed consolidated results. Technical report, Massmart, December 2015.

Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

Cataldo Musto, Giovanni Semeraro, Marco De Gemmis, and Pasquale Lops. Word embedding techniques for content-based recommender systems: an empirical evaluation. In *Poster Proceedings of ACM RecSys 2015.*, Vienna, Austria, September 2015. URL http://ceur-ws.org/Vol-1441/recsys2015_poster23.pdf.

Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. 06 2011. URL http://arxiv.org/abs/1106.5730.

Sean Owen. Oryx overview, 2015. URL http://oryx.io/index.html.

Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.53. URL http://ipython.org.

Jérome Picault, Myriam Ribière, David Bonnefoy, and Kevin Mercer. *Recommender Systems Handbook*, chapter How to Get the Recommender Out of the Lab?, pages 333–365. Springer

US, Boston, MA, 2011. ISBN 978-0-387-85820-3. doi: 10.1007/978-0-387-85820-3_10. URL http://dx.doi.org/10.1007/978-0-387-85820-3_10.

Pick n Pay. Integrated annual report 2015. Technical report, Pick n Pay, June 2015.

Provincial Treasury - Gauteng Province. The retail industry on the rise in south africa. Technical report, Provincial Treasury - Gauteng Province, 2012.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL http://www.R-project.org/.

Steffen Rendle and Christoph Freudenthaler. Improving pairwise learning for item recommendation from implicit feedback. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 273–282. ACM, 2014.

Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 452–461. AUAI Press, 2009.

Francesco Ricci, Lior Rokach, and Bracha Shapira. *Recommender Systems Handbook*, chapter Recommender Systems: Introduction and Challenges, pages 1–34. Springer US, Boston, MA, 2015. ISBN 978-1-4899-7637-6. doi: 10.1007/978-1-4899-7637-6_1. URL http://dx.doi.org/10.1007/978-1-4899-7637-6_1.

J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. The adaptive web. chapter Collaborative Filtering Recommender Systems, pages 291–324. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 978-3-540-72078-2. URL http://dl.acm.org/citation.cfm?id=1768197.1768208.

Guy Shani and Asela Gunawardana. *Recommender Systems Handbook*, chapter Evaluating Recommendation Systems, pages 257–297. Springer US, Boston, MA, 2011. ISBN 978-0-387-85820-3. doi: 10.1007/978-0-387-85820-3_8. URL http://dx.doi.org/10.1007/978-0-387-85820-3_8.

Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Alan Hanjalic, and Nuria Oliver. Tfmap: Optimizing map for top-n context-aware recommendation. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 155–164, New York, NY, USA, 2012a. ACM. ISBN 978-1-4503-1472-5. doi: 10.1145/2348283.2348308. URL http://doi.acm.org/10.1145/2348283.2348308.

Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Nuria Oliver, and Alan Hanjalic. Climf: Learning to maximize reciprocal rank with collaborative less-is-more filtering. In *Proceedings of the Sixth ACM Conference on Recommender Systems*, RecSys '12, pages

139–146, New York, NY, USA, 2012b. ACM. ISBN 978-1-4503-1270-7. doi: 10.1145/2365952.2365981. URL http://doi.acm.org/10.1145/2365952.2365981.

Shoprite Holdings, Ltd. Annual financial statements 2015. Technical report, Shoprite Holdings, Ltd., June 2015.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972. URL http://dx.doi.org/10.1109/MSST.2010.5496972.

Spark Development Team. *Apache Spark*. Apache Software Foundation, Los Angeles, USA, 2016. URL http://spark.apache.org/.

Statistics South Africa. General household survey. Technical report, Statistics South Africa, January 2013.

Statistics South Africa. Quarterly employment statistics. Technical report, Statistics South Africa, December 2014.

Statistics South Africa. Retail trade sales. Technical report, Statistics South Africa, December 2015a.

Statistics South Africa. Gross domestic product. Technical report, Statistics South Africa, December 2015b.

Diane Tang, Ashish Agarwal, Deirdre O'Brien, and Mike Meyer. Overlapping experiment infrastructure: More, better, faster experimentation. In *Proceedings 16th Conference on Knowledge Discovery and Data Mining*, pages 17–26, Washington, DC, 2010.

James Turner. Hadoop: What it is, how it works, and what it can do, 2011. URL https://www.oreilly.com/ideas/what-is-hadoop.

U.S. Census Bureau. Quarterly retail e-commerce sales - 1st quarter 2015. U.S. Department of Commerce, May 2015.

Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 193–204, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807161. URL http://doi.acm.org/10.1145/1807128.1807161.

World Wide Worx. 2014 mastercard online shopping behaviour study - results from south africa. Technical report, MarterCard, March 2014.

Ya Xu, Nanyu Chen, Addrian Fernandez, Omar Sinno, and Anmol Bhasin. From infrastructure to culture: A/b testing challenges in large scale social networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 2227–2236, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3664-2. doi: 10.1145/2783258.2788602. URL http://doi.acm.org/10.1145/2783258.2788602.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228301.

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522737. URL http://doi.acm.org/10.1145/2517349.2522737.

Yuchen Zhang and Michael I. Jordan. Splash: User-friendly programming interface for parallelizing stochastic algorithms. *CoRR*, abs/1506.07552, 2015. URL http://arxiv.org/abs/1506.07552.

Yuchen Zhang, Amr Ahmed, Vanja Josifovski, and Alexander Smola. Taxonomy discovery for personalized recommendation. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 243–252. ACM, 2014.

Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 249–256, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2409-0. doi: 10.1145/2507157.2507164. URL http://doi.acm.org/10.1145/2507157.2507164.