Western Graduate&PostdoctoralStudies

Western University

Scholarship@Western

Electronic Thesis and Dissertation Repository

1-23-2017 12:00 AM

Contextual Model-Based Collaborative Filtering for Recommender Systems

Dennis E. Bachmann, *The University of Western Ontario*

Supervisor: Dr. Miriam A. M. Capretz, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Engineering Science degree in Electrical and Computer Engineering

© Dennis E. Bachmann 2017

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Computer and Systems Architecture Commons

Recommended Citation

Bachmann, Dennis E., "Contextual Model-Based Collaborative Filtering for Recommender Systems" (2017). *Electronic Thesis and Dissertation Repository*. 4466.
https://ir.lib.uwo.ca/etd/4466

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

Recommender systems have dramatically changed the way we consume content. Internet applications rely on these systems to help users navigate among the ever-increasing number of choices available. However, most current systems ignore that user preferences can change according to context, resulting in recommendations that do not fit user interests. Context-aware models have been proposed to address this issue, but these models have problems of their own. The ever-increasing speed at which data are generated presents a scalability challenge for single-model approaches. Moreover, the complexity of these models prevents small players from adapting and implementing contextual models that meet their needs.

This thesis addresses these issues by proposing the $(CF)^2$ architecture, which uses local learning techniques to embed contextual awareness into collaborative filtering (*CF*) models. *CF* has been available for decades, and its methods and benefits have been extensively discussed and implemented. Moreover, the use of context as filtering criteria for local learning addresses the scalability issues caused by the use of large datasets. Therefore, the proposed architecture enables the creation of contextual recommendations using several models instead of one, with each model representing a context. In addition, the architecture is implemented and evaluated in two case studies. Results show that contextual models trained with a small fraction of the data resulted in similar or better accuracy compared to *CF* models trained with the total dataset. Moreover, experiments indicate that local learning using contextual information outperforms random selection in accuracy and in training time.

Keywords: Recommender System, Collaborative Filtering, Context Aware, Local Learning, Instance Selection

Acknowlegements

First and foremost, I would like to thank my supervisor Dr. Miriam Capretz for her continuous supervision, mentoring, and for believing in my potential as a researcher. I will always be grateful for the opportunities she gave me and for entrusting me with the freedom to pursue my own research interests.

I would also like to thank my father, Dorian Luiz Bachmann, and my mother, Nilce Harth Bachmann, for their support and encouragement throughout my decisions in life. Despite the huge geographical distance between us, they have always been there for me. I am blessed to be a part of their family and I know that I can always count on them.

Special thanks to my brother Christian, who has offered me countless hours of comfort and guidance. His professionalism and determination have always been a source of inspiration to me.

I would also like to extend my heartfelt appreciation to my brother Adrian and my sister-in-law Megumi, who did everything in their power to assist me in coming to Canada and whose patience and appreciation was a fundamental part of this thesis.

I would also like to thank my outstanding research team for helping me throughout the course of this research: Daniel Berhane Araya, Wander Queiroz, Sara Abdelkader, Márcio Lopes, and Alex L'Heureux. Special thanks to Dr. Katarina Grolinger, Dr. Wilson Higashino, and Dr. Hany ElYamany for believing in my potential, for all their patience reviewing my work, and for taking innumerable hours out of their schedule to broaden my horizons.

This thesis would also not have been made possible without the support of all my friends. Thanks to my old-time friends Mauro Ribeiro and Roberto Barboza Jr. for beeing nearby and always willing to meet me for a pint of beer. Thanks to my new Canadian friends who have embraced me as one of their own: Fred, Al, Alix, Matt, Jason, Doug, Justin, Amberley, Jerusha, Melissa, Kira, Lynsey, Billy, and both Johns. Needless to say that I am also grateful for the support received from my distant friends: Gustavo Karazawa, Sérgio Meyenberg Jr., Thiago Ferreira, Antônio Takahara, Arthur Valadares, Nikolas Aschermann, Fellipe Choi, Larissa Mayer Pontes, Maylla Hiromoto, Fernanda Túlio, Katia Portes, Vanessa Hamann, and all of those that will never forgive me for not mentioning them in this symbolic list.

Last but not least, I would like to thank my recently born nephew Kai, who has already provided me several moments of laughter and joy.

Contents

Abstract ii

Acknowlegements iii

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Motivation . . . 2
1.2 Contribution . . . 3
1.3 Organization of the Thesis . . . 4

2 Background and Literature Review 5

2.1 Background . . . 5
2.1.1 Recommender Systems . . . 5
2.1.2 Local Learning . . . 10
2.1.3 Instance Selection . . . 11
2.1.4 Contextual Inference . . . 12
2.2 Literature Review . . . 13
2.2.1 Local Learning and Instance Selection . . . 13
2.2.2 Contextual Information in Recommender Systems . . . 15
2.3 Summary . . . 17

3 Context Filtering for Model-Based Collaborative Filtering Recommender System Architecture 19

3.1 External Entities . . . 21
3.1.1 Recommendations . . . 21
3.1.2 Client . . . 21
3.1.3 Recommender Service . . . 21

3.1.4 Application 21
3.1.5 Trainer 22
3.1.6 External Knowledge 22
3.2 Storage Layer 22
3.2.1 Rating Storage 22
3.2.2 Contextual Storage 23
3.2.3 Contextual Model Storage 24
3.3 Training Layer 24
3.3.1 Contextual Filtering 26
3.3.2 Recommender Trainer 28
3.4 Production Layer 28
3.4.1 Request Inspector 29
3.4.2 Recommender Engine 31
3.5 Summary 32

4 $(CF)^2$ Framework 33
4.1 Storage Layer 33
4.1.1 Rating Storage 33
4.1.2 Contextual Storage 35
4.1.3 Contextual Model Storage 36
4.2 Training Layer 37
4.2.1 Contextual Filtering 37
4.2.2 Recommender Trainer 41
4.3 Production Layer 43
4.3.1 Request Inspector 43
4.3.2 Recommender Engine 47
4.4 Summary 48

5 Evaluation 49
5.1 Methodology 50
5.2 Case Studies 52
5.2.1 Case Study 1: Embedded Context 53
5.2.2 Case Study 2: Contextual Inference 60
5.3 Summary 68

6 Conclusions and Future Work 69
6.1 Conclusions 69

6.2 Future Work 70

Bibliography **72**

Curriculum Vitae **77**

List of Figures

2.1 Ratings given to movies by users. 6
2.2 Example of fine-point ratings scale. 9
2.3 Example of contextual inference for location and weather conditions. 13

3.1 $(CF)^2$ architecture. 20
3.2 Dataset divided into smaller subsets based on contextual attributes. 20
3.3 Rating data are requested and formatted to meet the requirements of $(CF)^2$. . . 23
3.4 External service providing external contextual attributes. 24
3.5 Training phase using the weather condition context. 25
3.6 Production phase using the weather condition context. 30

4.1 Class diagram for the $(CF)^2$ architecture. 34

5.1 Training set T segmented by contextual attributes and random dataset reduction. 50
5.2 Training set T segmented into subsets T_c to train model m_c. 51
5.3 Validation set V segmented into subsets V_c to predict ratings using model m_c. . 52
5.4 MSE of m_t, m_c, and m_r^c segmented by dataset. 56
5.5 Average MSE for the traditional method and each dataset reduction technique. . 57
5.6 MSE value by dataset size for each dataset reduction technique. 58
5.7 Training time in milliseconds taken to train each model. 59
5.8 MSE of m_t, m_c, and m_r^c segmented by dataset. 63
5.9 Average MSE for the traditional method and each dataset reduction technique. . 64
5.10 MSE value by dataset size for each dataset reduction technique. 66
5.11 Training time in milliseconds taken to train each model. 67

List of Tables

4.1 Column structure of the matrix returned by the *rating storage* component. . . . 35
4.2 Generic data source structure used by the *contextual storage* component. 35
4.3 Column structure of the matrix supplied to the *contextualizeDataset* method. . . 38
4.4 Column structure of the matrix returned by the *inferContextForDataset* method. 39
4.5 Structure supplied to the *matchContextualAttributesWithExternalDataset* method. 41
4.6 Structure of the requests passed as parameters to the *train* method. 42
4.7 Structure of the requests returned by the *contextualizeRequest* method. 44

5.1 Notations used throughout the chapter. 51
5.2 Parameters used to train the *CF* models. 53
5.3 Size of each dataset classified by contextual attribute. 55
5.4 *MSE* values obtained using dataset reduction. 56
5.5 Training time (in milliseconds) taken to train each contextual model. 57
5.6 Size of each dataset classified by contextual attribute. 61
5.7 *MSE* values obtained using dataset reduction. 62
5.8 Training time (in milliseconds) taken to train each contextual model. 65

Chapter 1

Introduction

In everyday life, it is not uncommon to rely on recommendations by friends or family to decide on a restaurant for dinner. During the years, many publications around the globe have specialized in providing people with lists of recommendations for all kinds of areas. Although *Zagat*[1] and *Michelin Red Guide*[2] are two of the best-known publications in this category, many others are available and range from recommending the best food in town to suggesting which dog breed best matches someones personality. Nowadays, Internet applications have turned to recommender systems to help users navigate among the ever-increasing number of available choices. Perhaps *Netflix*[3] is the most common example of such a system and helps video enthusiasts to decide on which movie or TV show marathon to watch next.

Recommendation systems can be broadly categorized as collaborative, content-based, or hybrid [1]. Collaborative recommendation resembles word-of-mouth communication, in which the opinions of others are used to determine the relevance of a recommendation. In this case, a collaborative recommender system uses the ratings provided by its users either to recommend an interesting item or to identify like-minded users. For instance, *Netflix* uses this technique to recommend videos that were highly rated by people who, in the past, have rated videos in a similar manner to the user. Content-based recommendation focusses on using the content of an item to assert its relevancy. To put it differently, a music recommender application that uses the content-based approach may use the genre of a piece of music to recommend other musical selections that share the same genre. The hybrid category is reserved for those systems that use both techniques when deliberating on a recommendation. As an illustration, the same music recommender that uses the content-based approach can also be extended to incorporate the collaborative method. In this scenario, the recommendation would contain pieces of music

[1] http://zagat.com
[2] http://travelguide.michelin.com
[3] http://netflix.com

of the same genre that were also appreciated by like-minded users.

1.1 Motivation

Much work has been done to develop new recommender methods that focus both on the item and on the user [2, 3, 4]. Nevertheless, in many applications, this does not suffice. In fact, if only items and users are used to provide recommendations, it would be safe to assume that a travel agency is doing a proper recommendation if it suggests a ski package to someone who has skiing as an interest. However, this probably would not be a wise idea if the recommendation were given in the peak of summer. Therefore, it is also important to incorporate contextual information into the recommendation process.

In recent years, with the proliferation of smart phones, smart watches, and other smart devices, applications have access to more and more contextual information from their users, and yet recommender systems fail to use this contextual information explicitly when giving recommendations. One way of addressing this issue is to create new models that can incorporate user context and thereby improve the quality of their recommendations [5, 6].

Although these efforts are promising, several issues arise when incorporating context into the recommendation process. The first is the volume and speed at which contextual data are generated, making it a challenge to train and use a single contextual model. More specifically, the processing power required to train a model using such datasets is enormous [7]. Therefore, this work leverages local learning techniques to distribute the load among several models instead of one, reducing the number of entries used to train each model.

Moreover, the difficulty involved in incorporating user context into new models presents another challenge because it adds new dimensions to the model. This work proposes an architectural design that relies on existing collaborative techniques to provide contextual awareness to recommendations. Assuming that existing collaborative models are unaware of user context, the architecture uses the contextual information as filtering criteria for a local learning technique, which attempts to locally adjust the capacity of the training system to the properties of the training set in each area of the input space [8]. By rearranging the data in this way, each generated model represents a context, after which these models are used to generate contextual recommendations.

Another problem is the complexity of understanding and implementing new recommender models, as well as the costs involved. Large companies have enough resources to invest in their own models and peculiarities. For instance, not long ago, *Netflix* paid 1 million dollars for statisticians to improve their *Cinematch* algorithm by 10.06% [9]. On the other side, small companies, developers, and researchers in general struggle to find a flexible solution that re-

moves from them the need to understand the inner workings of each model. Therefore, another motivation of this work is to present a framework that enables small players to implement their own recommender systems using existing models that best suit their needs.

1.2 Contribution

This research provides several contributions aimed at overcoming the challenges mentioned and ultimately at enabling the development of Context Filtering for Model-Based Collaborative Filtering recommender system $(CF)^2$ architecture.

The main contribution of this thesis is $(CF)^2$, a modular architecture that uses local learning techniques to embed contextual awareness into collaborative filtering models. $(CF)^2$ enables algorithms that focus solely on items and users to leverage contextual information in addition when making recommendations. Moreover, this added capability is achieved without requiring changes to the algorithms involved. The components of $(CF)^2$ are described in detail, and their roles, functions, and relationships are explained.

Another major contribution of this thesis is to introduce a framework that can be used as a foundation to implement $(CF)^2$. Details are presented at an algorithmic level, which enables small players to build context-aware recommender systems using widely available collaborative filtering libraries. Moreover, the concepts laid out by this thesis do not restrict their implementation by enforcing a single machine language or paradigm, enabling use of the technology that best fits the needs of each application. The proposed framework was implemented using the functional paradigm of the *Scala*[4] language. This implementation used the large-scale data-processing properties of the *Spark*[5] engine and was made available in a public repository.[6]

The proposed approach was evaluated in the *Find Good Items* task. A methodology to evaluate $(CF)^2$ is provided, and two case studies using implicit ratings are presented. The first case study uses embedded context and analyzes how $(CF)^2$ performs when the contextual attribute operating system and platform is used to provide recommendations for a Web site specialized in weather- and traveller-related content and technology. Moreover, contextual inference is used in a second case study where weather conditions are used as contextual attributes. Both case studies were evaluated using the same methodology and had their accuracy results compared against traditional methods using *CF* models. The results indicated that contextual models trained with a small fraction of the data resulted in similar or better accuracy compared to models trained using the total dataset. In addition, local models created using random sam-

[4]http://www.scala-lang.org
[5]http://spark.apache.org
[6]https://github.com/dennisbachmann/cf2-scala

pling instead of contextual dataset reduction were compared, and the results demonstrate that the use of contextual information outperforms random selection in accuracy.

The results obtained in this thesis shows that contextual information can be used as filtering criteria for local learning algorithms and that existing collaborative filtering algorithms can be used by $(CF)^2$ to leverage contextual information.

1.3 Organization of the Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 provides background information that is useful in understanding this work as well as a literature review of work done in the areas of local learning, instance selection, and context-aware recommender systems. This chapter first provides an introduction to the technical terms and concepts that are used throughout this thesis. Moreover, this chapter presents a review of current studies on local learning and instance selection as well as the pre-filtering approach towards context-aware recommender systems.

- Chapter 3 describes the components of the $(CF)^2$ architecture. Besides providing an overview of the function of each component, this chapter also describes how each component interacts with others. The description is broken down into four sections: the first is external entities, which covers the produced artifact as well as the external services that are part of $(CF)^2$. The second is the storage layer, which describes the components that serve as abstractions to prevent direct access to the datasets. The third is the training layer, which covers the components involved during the training phase of $(CF)^2$. Finally, the production layer explains the components involved in the production phase.

- Chapter 4 provides a framework that can be used as a foundation to implement the $(CF)^2$ architecture. It starts with a discussion of the design and structure of the framework. Afterwards, a generic implementation of each component is provided at the method level.

- Chapter 5 presents an evaluation of $(CF)^2$. It starts with a description of the scenario being evaluated and how rating data were captured. Next, the methodology used during the evaluation process is presented and discussed. Finally, this chapter presents two case studies that validate $(CF)^2$ in different scenarios, assess the systems execution time, and analyze the effects of using contextual dataset reduction instead of random sampling.

- Chapter 6 presents the conclusions of this work, as well as a discussion of areas of future work involving the $(CF)^2$ architecture.

Chapter 2

Background and Literature Review

The objective of this chapter is two-fold: first, it introduces the background terminology and concepts related to the topics discussed in this thesis; second, it gives an overview of existing research done in the area of local learning and instance selection and also studies that use contextual information in recommender systems.

2.1 Background

This section introduces and discusses the concepts of recommender systems, local learning, instance selection, and contextual inference, which are the foundation for understanding $(CF)^2$.

2.1.1 Recommender Systems

As Resnick and Varian [10] mentioned in their 1997 article, the term *recommender systems* is applied to systems that, by using various information sources, can suggest relevant items to a user. In addition to filtering out irrelevant content from the list of recommended items, they also try to balance factors like accuracy, novelty, diversity, and stability in these recommendations.

The data sources may contain rating information acquired explicitly (typically by collecting users' ratings) or implicitly (typically by monitoring users' behaviour, such as songs heard, applications downloaded, Web sites visited, and books read). Other information sources include demographic databases, social networks, and data from *Internet of Things* (*IoT*) devices, with the last being a new trend in the development of Recommender Systems [11]. Use of *IoT* devices enables the acquisition and use of contextual information in a manner unobtrusive to the users, which is a desirable feature in next-generation Recommender Systems [4].

As mentioned by Bobadilla et al. [11], Lu et al. [12], Ricci et al. [13], and Adamavicius and Tuzhilin [4], recommender systems can use several filtering algorithms or techniques. The

most relevant are further described below.

Collaborative Filtering

Collaborative filtering (*CF*) was the first technique used by a recommender system and is also considered to be the most popular and widely implemented [13]. Examples of popular Web sites that make use of this technique are *Amazon*[1], *TiVo*[2], and *Netflix* [14, 15].

CF provides recommendations based on the opinions of others who share the same interests as the user [12]. These opinions are often represented as the ratings matrix R [16]. This matrix is an $m \times n$ matrix containing m users and n items. Hence, the rating of user i for item j is given by r_{ij}. Because in any recommender system, the number of ratings obtained is usually very small compared to the number of recommendations that must be made, the matrix R is often a sparse one [4]. This property is usually exemplified by considering an example of a video streaming service. Most users have watched only a small subset of the available videos. Therefore, even if users rate all watched videos, many items r_{ij} in the matrix R will still be missing.

The approach adopted by *CF* methods is that these missing ratings can be guessed because the observed ratings are often highly correlated across various users and items [17]. For example, if two users share similar ratings among items, the *CF* algorithm will identify these users as having similar taste. Therefore, this similarity can be used to make inferences about those ratings that are missing a value. Most *CF* models focus on leveraging inter-item correlations or inter-user correlations [17]. This is the reason why *CF* is often referred to as "people-to-people correlation" [18]. Figure 2.1 illustrates this rating inference based on similarities among users. In this illustration, it is clear that Bob and Alice share the same taste, whereas Chris does not share these interests.

Users	Gone with the Wind	The Godfather	Rambo	Citizen Kane
Alice	★★★★★	★★☆☆☆	★☆☆☆☆	★★★★★
Chris	★☆☆☆☆	★★★★★		★☆☆☆☆
Bob	★★★★★	★★☆☆☆	★☆☆☆☆	

Figure 2.1: Ratings given to movies by users.

This method has problems of sparsity and limited coverage and is still an open research field. Data reduction techniques appear to be a promising research directions to solve this

[1]http://www.amazon.com
[2]http://www.tivo.com

problem [4]. The use of *CF* also has the advantage of using an approach that enables it to obtain meaningful relations between users or items that are not directly connected [13].

According to Adamavicius and Tuzhilin [4], *CF* algorithms can be grouped into two categories: *memory-based* and *model-based*.

1. *Memory-based methods*: Memory-based algorithms, also referred to as *neighbourhood-based algorithms*, were among the earliest *CF* algorithms [16]. These algorithms predict the ratings of user-item combinations based on their neighbourhood [17]. These neighbourhoods can be defined in one of two ways:

    - *User-based collaborative filtering*: these algorithms provide recommendations of items that were liked by similar users [4]. Therefore, the objective is to recommend ratings for items not yet rated by a user u. This can be achieved by computing weighted averages of the ratings provided by users sharing the same interests as the user u [17]. To exemplify, whenever Alice and Bob have similar ratings among rated movies, a user-based *CF* algorithm can use the rating given by Alice to the movie *Rambo* to predict the rating that Bob would have given if he had to rate it himself.
    - *Item-based collaborative filtering*: these algorithms provide recommendations of items similar to those that the user liked in the past [4]. Consequently, to predict the rating for an item i given by a user u, the first step involves determining the set S of items that are most similar to item i. The ratings in set S are then used to predict whether the user u will like item i. Hence, if Bob gave positive ratings to classic movies like *Gone with the Wind*, these ratings can be used to predict his ratings of other classic movies, like *Citizen Kane*.

    The decision on which approach to use usually relies on the ratio of the number of users to the number of items. In those cases where the number of users is greater than the number of items, item-based approaches are more appropriate because they provide more accurate recommendations while being more computationally efficient [13]. On the other hand, user-based approaches usually provide more original recommendations [19].

    These neighbourhood-based methods can be viewed as generalizations of k-nearest neighbours classifiers. Therefore, these methods are considered *instance-based learning methods*, which are specific to the instance being predicted [20]. Among the advantages of these algorithms are their simplicity of implementation and the ease of explanation of their recommendations. Even so, these algorithms do not work very well with sparse rating matrices [17].

2. *Model-based methods*: Model-based algorithms make use of machine learning and data mining methods to generate predictive models that will be used to predict missing entries in the ratings matrix R [20]. As with others supervised or unsupervised machine learning methods, these predictive models are created before the prediction phase. Examples of traditional machine-learning methods that can be generalized to the model-based *CF* scenario include decision trees, rule-based methods, Bayes classifiers, regression models, support vector machines, and neural networks [21].

   According to Aggarwal [20], model-based collaborative filtering recommender systems (*MB-CFRS*) often have a number of benefits over memory-based methods:

   - *Space efficiency*: typically, the generated model is much smaller than the original rating matrix.
   - *Training and prediction speed*: whereas memory-based methods have a quadratic preprocessing stage, model-based systems are usually much faster in the pre-processing phase, while in most cases also being able to make predictions efficiently.
   - *Avoiding overfitting*: although overfitting is a serious problem in machine-learning algorithms, the summarization approach used by model-based methods can often help in avoiding overfitting.

   Although memory-based systems offer the advantage of simple implementation [17], they often lack accuracy. In general, model-based techniques are the most accurate methods, especially when using latent factor models [20].

Regardless of which method is used, *CF* provides recommendations based on the opinions of others who share the same interests as the user [12]. These opinions are captured in the form of ratings and are often specified on a scale that indicates how satisfied a user is with an item. Although it is possible for the rating scale to be represented as a set of continuous values, this is relatively rare. Usually the rating scale is represented as a discrete interval representing how satisfied the user is with a certain item [17]. In addition, instead of using numerical values, the rating scale can be represented as icons. A common example of such a representation is illustrated in Figure 2.2, where a five-point scale representing the set $\{-2, -1, 0, 1, 2\}$ is portrayed as a star drawing.

Star Representation	Numerical Representation	Textual Representation
★★★★★	2	Excellent
★★★★☆	1	Good
★★★☆☆	0	Fair
★★☆☆☆	-1	Poor
★☆☆☆☆	-2	Terrible

Figure 2.2: Example of fine-point ratings scale.

Moreover, the interpretation of the rating scale may vary from one vendor to another. Even though both *Amazon* and *Netflix* use the same five-point rating scale, the interpretation varies. Whereas *Amazon* uses a linear scale, *Netflix* chose to use a five-star rating system in which the four-star point level corresponds to “really liked it” and the central three-start point level corresponds to “liked it”. Therefore, there are three favourable ratings and two unfavourable ratings [17].

A special case of a rating scale is the *unary* scale [17]. In these cases, users have the opportunity of providing their “like” opinion towards an item, but there is no alternative to specify a dislike. A *unary* rating scale is normally associated with systems that use of implicit feedback [22], in which users’ preferences are obtained based on their activities instead of being explicitly given by users. The impact of using such a scale must be taken into account by the recommender algorithm because there is no information on user dislikes.

Content-Filtering

The Content-Filtering recommends items similar to those the user liked in the past [12], but unlike the *CF* method, content-filtering uses the content or features of the compared item to calculate the similarity among all the other items in the system [12]. This technique can also make use of ontologies and semantic analysis to retrieve user preferences, storing them in a user profile [13].

To provide recommendations, the content-filtering recommender systems can use heuristics with common information-retrieval techniques, or can use machine-learning methods to build models based on the historical interest of the user [4].

Demographic Method

This method is justified by assuming that users belonging to the same demographic group share the same interests [11]. Although these approaches have been popular in the market-

ing literature, there has been relatively little recommender-systems research on demographic systems [12].

Context-Aware Systems

Context-aware recommender systems (*CARS*) are a widely researched topic that has empowered recommender applications in several areas, including movies, restaurants, travel, music, news, shopping assistants, mobile advertising, mobile apps, and many others [23].

Most current-generation recommender systems operates in the two-dimensional $User \times Item$ space, meaning that they take into consideration only user and item information [24]. However, *CARS* can provide multidimensionality to a model [23].

This multidimensionality can be an advantage because in many situations, the relevance of a certain item to a user may depend on the time of purchase, or the circumstances under which the item will be consumed [23]. For example, a travel agency Web site may want to use the season of the year to determine whether they should sell beach resort packages or ski vacations to their users.

Although the ability to add other dimensions to the model is desirable, many standard two-dimensional recommendation algorithms cannot be directly extended to the multidimensional case. On the other hand, reduction-based algorithms, which use only ratings belonging to the user context, can be used with any standard two-dimensional recommendation method, which suits the purpose of this research [4].

To put this into context, when reduction-based algorithms are used to provide recommendations, the *CARS* will use only the available ratings that match the context information desired by the user, thus behaving as a filter [23].

Hybrid Filtering

This technique uses two or more recommender methods to provide recommendations to the user [13]. Combining two or more methods is usually an attempt to overcome the drawbacks of one solution by incorporating another method [12].

The most common practice is to use *CF* recommendation techniques with content-filtering to avoid cold-start, sparseness, and scalability problems [4].

2.1.2 Local Learning

Local learning algorithms attempt to divide the training set into several local clusters to capture more effectively the properties of each neighbourhood of the input space [25]. This results in creating separate local models for each cluster [26].

Local learning is based on the assumption that large training datasets are very rarely evenly distributed in the input space [8]. Moreover, current machine-learning systems are not inherently efficient or scalable enough to deal with large data volumes, and therefore a growing fraction of data remain unexplored or underexploited [26]. Hence, local learning is considered a suitable approach for machine-learning algorithms that use large data volumes [26].

As defined by Bottou and Vapnik [8], local learning can be accomplished by performing a simple local algorithm for each testing pattern:

1. Select the training samples located in the vicinity of the test pattern.

2. Train the model using only these samples.

3. Apply the resulting model to the test pattern.

Recent studies have shown that local learning yields results far superior to a global learning strategy, especially on datasets that are not evenly distributed [8, 27, 28, 29].

In addition, for computationally intensive algorithms, it is faster to find solutions for k problems of size m/k than for one problem of size m [30].

2.1.3 Instance Selection

The exponential growth of available information has enabled machine-learning methods to be used in a wide variety of fields. However, training on large datasets is often a very slow process and has become a bottleneck [7]. For example, training of support vector machines (*SVM*) implies a high training-time complexity $O(n^3)$ and a spatial complexity $O(n^2)$ [31, 32, 33]. Moreover, in classification problems, a large training set often results in slower response times [34].

To overcome this issue, many methods have been developed to reduce the high computational complexity of machine-learning algorithms that use large datasets [35]. Among the proposed techniques, scaling down the training set has proved to be one of the most direct and effective ways to solve large-scale classification problems [7].

According to Liu [36], instance selection has the following functions:

- *Enabling*: Instance selection renders the impossible possible. When a data set is too large, it may not be possible to run a data mining algorithm. Instance selection reduces data and enables a data mining algorithm to function and work effectively with huge data.

- *Focussing*: The data includes almost everything in a domain, but a particular application is normally about only one aspect of the domain. It is natural and sensible to focus on the part of the data that is relevant to the application so that search is more focussed and mining more efficient.

- *Cleaning*: The "garbage in, garbage out" principle applies to all or almost all, data mining algorithms. It is therefore of paramount importance to clean data before mining. By selecting relevant instances, usually irrelevant ones as well as noise and/or redundant data are removed. High-quality data lead to high-quality results and reduced costs for data mining.

To obtain a scaled-down training set, similar instances are removed from the dataset, resulting in a subset that can be used to make inferences about the original dataset [7]. However, when noisy instances are present, the accuracy of classification models can suffer. To alleviate these problems, the instance selection technique also proposes to delete these noisy instances [34].

Instance selection approaches are diverse and include random selection, genetic algorithm-based selection, progressive sampling, using domain knowledge, and cluster sampling [7].

2.1.4 Contextual Inference

Because useful contextual attributes may not be available in the historical data, the architecture uses the *contextual inference* concept, which uses inference rules or external knowledge to provide extended contextual attributes to the dataset.

An example of this *contextual inference* concept is illustrated in Figure 2.3, which shows the process of extending a request containing an IP address and a time to contain in addition the location and the weather condition at the time of the request. This is achieved by first querying a geolocation service to obtain the approximate geographic location of the client and then using this location and the time of the request to query a weather service to obtain the appropriate weather conditions.

Although this example uses external data or services to extend the contextual attributes, this may not be necessary. On some occasions, these transformations can be done using internal logic, like the classification of a given day into a weekday or a weekend day.

These examples demonstrate the potential that *contextual inference* can provide to implementations of $(CF)^2$, because contextual attributes that may look uninteresting at first can be extended to provide powerful contextual meaning to data.

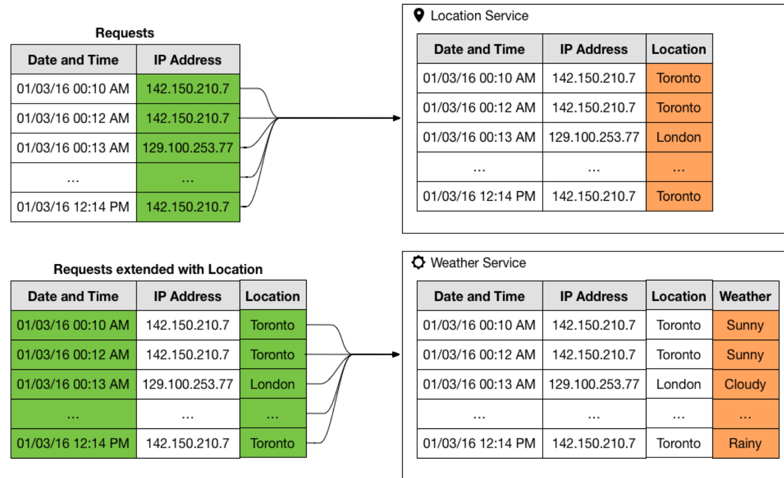

Figure 2.3: Example of contextual inference for location and weather conditions.

2.2 Literature Review

This section provides a literature review and is broken down into two sub-sections. The first covers instance selection and local learning and provides a review of how these techniques have been used in the academia, and the second section describes the use of a pre-filtering approach for *CARS*.

2.2.1 Local Learning and Instance Selection

Several previous studies have used instance selection with the intent of reducing the number of instances in the training set without affecting classification accuracy [37, 38, 39, 40, 41]. Leyva *et al.* [37] proposed three selection strategies with several accuracy-reduction trade-offs. In their work, the focus was on memory-based algorithms to prioritize instance selection for nearest-neighbour classification problems. Their results involved 26 databases and were compared with 11 state-of-the-art methods in standard and noisy environments. The comparison was performed using the Technique for Order Preference by Similarity to Ideal Solution (*TOPSIS*), and the proposed Local Set-Based Centroids Selector method (LSCo) achieved the best rankings for all levels and types of noise but one, for which it was the second-best.

García *et al.* [38] outlined the importance of instance selection in the data reduction phase of knowledge discovery and data mining. Their work demonstrated that instance selection performs a complementary function to feature selection. In addition, they defined instance selection as an intelligent operation of instance categorization according to the degree of irrelevance or noise and depending on the data mining task. Furthermore, they separated the

instance selection task into two processes: training set selection and prototype selection. Their work also presented a list of almost 100 prototype selection methods that had been proposed in the literature.

Carbonneau *et al.* [39] proposed the Random Subspace Instance Selection (RSIS) method to improve pattern recognition problems that can be modelled using multiple-instance learning (*MIL*). According to their work, state-of-the-art *MIL* methods provide high performance when strong assumptions are made about the underlying data distributions and the ratios of positive to negative instances in positive bags (instances containing all patterns). Their method focused on cases where prior assumptions about data structure could not be made and the ratios of instances in bags were unknown. To achieve this result, instance selection probabilities were computed based on training data clustered in random sub-spaces and then used to generate a pool of classifiers.

Chen *et al.* [40] performed instance selection by leveraging a genetic algorithm approach to improve the performance of data mining algorithms. In their work, they introduced a novel instance selection algorithm called a genetic-based biological algorithm (GBA). GBA fits a “biological evolution” into the evolutionary process, where the most streamlined process also complies with the reasonable rules. This means that, after long-term evolution, organisms find the most efficient way to allocate resources and evolve. Consequently, this technique closely simulates the natural evolution of an algorithm to become both efficient and effective.

Silva *et al.* [41] proposed the e-MGD method for instance selection. This method is as an extension of the Markov Geometric Diffusion (MGD) method, which is a linear complexity method used in computer graphics to simplify triangular meshes. The original method was extended to reduce datasets commonly found in data mining.

In contrast to these studies [37, 38, 39, 40, 41], this research introduces context to instance selection because a point might be considered irrelevant or noise in one context, but not in another. The studies described earlier considered the training set as a whole, rather than treating it as a collection of sub-spaces. Moreover, these studies dealt with neighbourhood-based classifiers, whereas this research is focussed on model-based classifiers.

Other studies have used local learning to achieve simplified local models and increased precision. Piegat and Pietrzykowski [25] presented a new version of the mini-model method. Generally, these models do not identify the full global model of a system, but only a local model of the neighbourhood of the query point of special interest. In their work, the authors extended the mini-model method to include local dimensionality reduction.

The work conducted by Domeniconi *et al.* [42] also aimed to perform dimensionality reduction. In their study, they tackled the curse of dimensionality suffered by clustering algorithms by discovering clusters in sub-spaces spanned by several combinations of dimensions using

local feature weightings. Their method associated with each cluster a weight vector, whose values were then used to capture the relevance of features within the corresponding cluster. This approach avoids the risk of information loss encountered in global dimensionality reduction techniques and does not assume any data distribution model.

Wu and Schölkopf [43] presented a local learning approach for clustering. Their idea was that an adequate clustering result should have the property that the cluster label of each data point can be well predicted based on its neighbouring data and their cluster labels. Relaxation and eigen-decomposition techniques were used to solve this problem. In addition, the authors provided a parameter selection method for the proposed clustering algorithm.

Chitta *et al.* [44] proposed a sparse kernel *k*-means clustering algorithm that incrementally sampled the most informative points from the dataset using importance sampling and constructed a sparse kernel matrix using these sampled points. This sparse kernel matrix was then used to perform clustering and obtain cluster labels. This combination of sampling and sparsity reduces both the running time and the memory complexity of kernel clustering. The authors also showed analytically that only a small number of points from the dataset need to be sampled, yielding a well-bounded error for the resulting approximation. Zhou *et al.* [45] proposed a global and local structure-preserving sparse sub-space learning model for unsupervised feature selection. Their model can perform feature selection and sub-space learning simultaneously. In addition, they developed a greedy algorithm to implement a generic combinatorial model.

These studies [25, 42, 43, 44, 45] focussed on ways of adapting existing models to simplify them and improve their accuracy while also performing dimensionality reduction. Nevertheless the models presented were static and could not be extended to other existing models used in *MB-CFRS*. In contrast, the present work focusses on contextual information explicitly available in the dataset or inferred to perform clustering. Hence, by using data embedded in the dataset, the properties of local learning can be applied to existing algorithms.

2.2.2 Contextual Information in Recommender Systems

Several studies have focussed on enhancing recommendation accuracy by using contextual information gathered from user interactions. Some have focussed on using contextual information to remove the variability and noise that are an intrinsic part of human behaviour [46, 47], whereas others used this additional information to retrieve personalized items for users according to their particular context [48, 49, 50, 51, 52]. In addition, other studies used contextual information to present a multidimensional approach to recommender systems that can provide contextual recommendations using existing *CF* algorithms [53, 54, 55, 56].

Jawaheer *et al.* [46] discussed the user modelling preferences incorporated into context-

aware recommender systems and questioned how user feedback is applied to recommender systems. In their work, they proposed a classification framework to use explicit and implicit user feedback in recommender systems based on a set of distinct properties that included Cognitive Effort, User Model, Scale of Measurement, and Domain Relevance. Akuma *et al.* [47] investigated the relationship between implicit parameters and explicit user ratings during search and reading tasks. In their work, they identified implicit parameters that were statistically correlated with the explicit user ratings through user study and used these parameters to develop a predictive model that could be used to represent the perceived users' relevance. Their findings suggest that there is no significant difference between predictive models based on implicit indicators and on eye gaze within the context examined.

Colombo-Mendoza *et al.* [48] proposed a movie recommender system that combined location, time, and crowd information with a semantic Web to recommend movies to a users. To obtain location and crowd information, the proposed approach used an explicit check-in performed by the user through the Foursquare application. Wang *et al.* [49] proposed a context-aware recommender for a Web news mobile application that relied on explicit input of user preferences modelled as a tensor to perform a cold-start of the recommender system.

Aghdam *et al.* [50] proposed a context-aware recommender algorithm based on hierarchical hidden Markov modelling. This approach represents the contextual changes in the user's preferences as hidden variables in the model and uses them to produce personalized recommendations to the user. Hussein *et al.* [51] proposed a software framework called *Hybreed* for developing context-aware hybrid recommender systems. This software framework enables developers to create new hybrid recommender systems by combining existing algorithms while facilitating the incorporation of context and third-party applications into the recommendation process. Liu and Wu [52] also proposed a generic framework to learn context-aware latent representations for context-aware collaborative filtering. In their work, the proposed framework combined contextual contents by means of a function that produces a context influence factor, which was then combined with each latent factor to derive latent representations. Moreover, a stochastic gradient descent-based optimization procedure was applied to fit the model by jointly learning the weight of each context and of the latent factors.

These studies [46, 47, 48, 49, 50, 51, 52] used explicitly obtained contextual information to model user preferences and then applied hard-wired algorithms to obtain recommendations. In contrast, the present research uses local learning properties to extend existing *CF* algorithms. Hence, algorithms previously developed and fine-tuned to address a specific scenario can now be extended to play a context-aware role without major modifications. Moreover, using contextual inference, new contextual attributes obtained from external sources can be used to provide contextual meaning to data.

Adomavicius *et al.* [53] presented a multidimensional approach to recommender systems that can provide recommendations based on additional contextual information beyond typical information on users and items used in most current recommender systems. They proposed to obtain a multidimensional rating estimate by selecting two-dimensional segments of ratings pertinent to the recommendation context and applying standard collaborative filtering or other traditional two-dimensional rating estimation techniques to these segments. Although they proposed this multidimensional approach, their work did not provide an extensible architecture that could make use of this technique. Building on this multidimensional approach, this study proposes an extensible architecture that can leverage contextual information to create contextualized local models.

Campos *et al.* [54] took advantage of time contexts in *CF* by grouping and exploiting ratings according to the contexts in which they were generated. This approach showed positive effects on recommendation in the presence of significant differences among user preferences within distinct contexts. The authors focussed only on time contexts, whereas this study has adopted a generic approach that can use any kind of contextual information.

Yao *et al.* [55] proposed a graph-based generic recommendation framework that constructs a multi-layer context graph from implicit feedback data and then executes ranking algorithms in this graph to produce context-aware recommendations. The proposed graph models the interactions between users and items and incorporates a variety of contextual information into the recommendation process. Pessemier *et al.* [56] described a framework to detect the current context and activity of a user by analyzing data retrieved from various sensors available on mobile devices. On top of this framework, a recommender system was built to provide users a personalized content offer, consisting of relevant information such as points of interest, train schedules, and tourist information, based on the user's current context.

In contrast to these studies [53, 54, 55, 56], the present research focusses on creating an architecture that enables recommendations to be generated using any kind of contextual information. Hence, by using local learning with contextual information to generate the models, only the ratings made in the same context as the target prediction are used. Moreover, by using a pre-filtering approach towards the training set, any *CF* model can be used to generate the recommendations. These models can range from simple classifiers to more complex regression-based approaches.

2.3 Summary

In this chapter, an overview of the concepts involved in local learning for *MB-CFRS* has been presented. More specifically, an introduction to the terminology of recommender systems has

been presented. In addition, an introduction to local learning and instance selection, which play a role in this research, was provided. Finally, current studies on local learning and instance selection domain as well as contextual information in recommender systems were discussed.

Chapter 3

Context Filtering for Model-Based Collaborative Filtering Recommender System Architecture

This chapter introduces the Context Filtering for the Model-Based Collaborative Filtering Recommender System $(CF)^2$ architecture. The $(CF)^2$ uses user rating data with embedded contextual attributes to generate smaller datasets for *model-based collaborative filtering recommender systems* (*MB-CFRS*). In addition, if a context cannot be directly obtained, $(CF)^2$ introduces a technique that enables the use of contextual knowledge to provide new attributes to the dataset.

Given that using large datasets to train *MB-CFRS* requires substantial computing resources, $(CF)^2$ proposes that instead of relying on a single model to generate user recommendations, several models should be used, each trained with just a small subset of the original dataset. $(CF)^2$ uses contextual attributes to perform dataset reduction. This is exemplified in Figure 3.2, where a full dataset is divided into smaller subsets based on contextual criteria.

By means of this approach, each model is trained using only the portion of the rating data that matches the contextual criteria. As a result, each training procedure requires fewer computational resources than if it were to use the totality of the rating data. The exact computational gain will depend on the complexity of the recommender algorithm.

In addition, not only does $(CF)^2$ provides improved performance during training, but it also embeds contextual awareness into the recommender system. All this can be achieved because model training is performed using only the subset of data that matches the contextual attributes.

$(CF)^2$ operates in two phases: a *training phase* and a *production phase*. Each of these phases is represented as a layer in the architecture. Moreover, an additional third layer, called the *storage layer*, is used to handle rating data and contextual information retrieval. This layer is also responsible for storing the trained models and their auxiliary files. The three layers

Figure 3.1: $(CF)^2$ architecture.

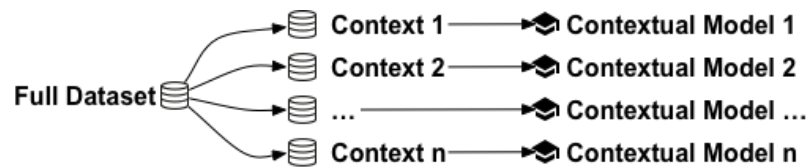

Figure 3.2: Dataset divided into smaller subsets based on contextual attributes.

ensure better separation of concerns and provide higher decoupling between the phases.

The following sections describe the architectural details proposed by $(CF)^2$ and illustrated in Figure 3.1.

3.1 External Entities

This section describes the artifact produced as well as the external entities of $(CF)^2$.

3.1.1 Recommendations

The only artifact generated by $(CF)^2$ is the list of recommendations that are returned when requested by the *client*. *MB-CFRS* allows recommendations to be a list of items for a user or a list of users for a certain item. Hence, the recommendations can belong to either of these two types.

3.1.2 Client

The *client* is a software application, acting on behalf of a user, which is mostly involved in consuming content served by an *application*. Eventually, the *client* may also be involved in obtaining recommendations from the *recommender service* of this *application*.

3.1.3 Recommender Service

The *recommender service* is the service responsible for processing requests issued by *clients* when they are in need of recommendations. Requests issued to this service must be accompanied by a client identifier, an identifier for the item or client for which the requesting party wants to obtain recommendations, and a list of contextual attributes.

3.1.4 Application

The *application* is the service that handles regular requests from *clients* interested in consuming its content, usually an item. Because the use of *MB-CFRS* requires data containing ratings given by users to items to generate the recommender model, this service is also responsible for gathering and storing ratings provided by *clients*.

Sometimes these data are also copied into a data warehouse. Because $(CF)^2$ is involved only with the rating data (item identifier, client identifier, rating, and contextual attributes), the data-warehousing solution can assume the role of *application* if desired.

3.1.5 Trainer

The *trainer* is the service responsible for initiating the training phase. This service can be manually invoked by a system administrator or it can be periodically invoked by a time-based job scheduler.

3.1.6 External Knowledge

The *external knowledge* is the service that holds the additional contextual information used by the *contextual storage* component. This service is often provided by an external vendor and is accessible through the use of an application programming interface (API). Examples of this service are weather forecast systems, demographic databases, and geo-location services.

3.2 Storage Layer

This layer contains all the components required by $(CF)^2$ to access the data in a standard manner. Because the actual data may be stored by third-party applications, this layer provides a set of components that serves as abstractions to these datasets.

3.2.1 Rating Storage

The *rating storage* ensures a standardized interface to access the historical dataset containing the ratings stored in the *application*. This interface will be used during the training phase to provide access to these ratings.

Implementation of this component will fetch a copy of the ratings available through the API of the *application* and format it in a way that can be easily used by other architecture components.

Although the actual format of the rating is not enforced by $(CF)^2$ because it can vary according to the application domain, it must contain at least the following elements:

- Client identification
- Item identification
- Rating
- Contextual attributes

Figure 3.3: Rating data are requested and formatted to meet the requirements of $(CF)^2$.

Figure 3.3 gives an example of this interface. A copy of the ratings is requested from the *application* and then formatted to maintain only the elements that are meaningful for the training.

3.2.2 Contextual Storage

The *contextual storage* component ensures a standardized interface to access the datasets that can provide additional contextual information to the data.

$(CF)^2$ may use more than one service to provide external contextual data. Hence, this component implements a method for each of these services to abstract the complexity of handling the peculiarities of each external service.

The use of external contextual attributes is achieved by matching embedded contextual attributes available in the *rating storage* component with those provided by the *contextual storage* component. Therefore, the returned value should follow a structure containing the following elements:

- Contextual attribute 1
- Contextual attribute ...
- Contextual attribute n
- External contextual attribute.

To exemplify such a data structure, Figure 3.4 illustrates a scenario in which two contextual attributes embedded in the dataset (location and date) are used to match an external contextual value (weather condition).

Location	Date	Weather Condition
Toronto	15/04/2016	Sunny
London	16/04/2016	Cloudy
...	...	...
Toronto	10/11/2016	Rainy

Embedded Context — External Context

Figure 3.4: External service providing external contextual attributes.

3.2.3 Contextual Model Storage

The *contextual model storage* component functions as the interface between the *training phase* and the *production phase*, handling the storage of and access to all the contextual collaborative filtering models trained during the *training phase*. These models will later be used by the *recommender engine* to provide customized *recommendations*.

To accomplish this task, the *contextual model storage* component must be implemented in such a way that only the most recent model is used for each context.

3.3 Training Layer

The *training layer* contains all the components required to extract contextual attributes from the historical data provided by the *rating storage*. These components are used to generate the contextual models that the *recommender engine* will use to provide personalized *recommendations* during the *production phase*.

Due to the complexity of this task, this layer is separated into two components. The *contextual filtering* component is responsible for identifying the contextual attributes of past requests and performing contextual inference when required. The *recommender trainer* component is responsible for carrying out training for each of the identified contextual attributes.

Moreover, to accommodate new ratings captured by the *application* and to provide more up-to-date recommendations, this layer must be periodically invoked by an external system. The exact periodicity is domain-specific and must be addressed on a case-by-case basis.

During the *training phase*, a traditional *MB-CFRS* uses historical data to generate a single

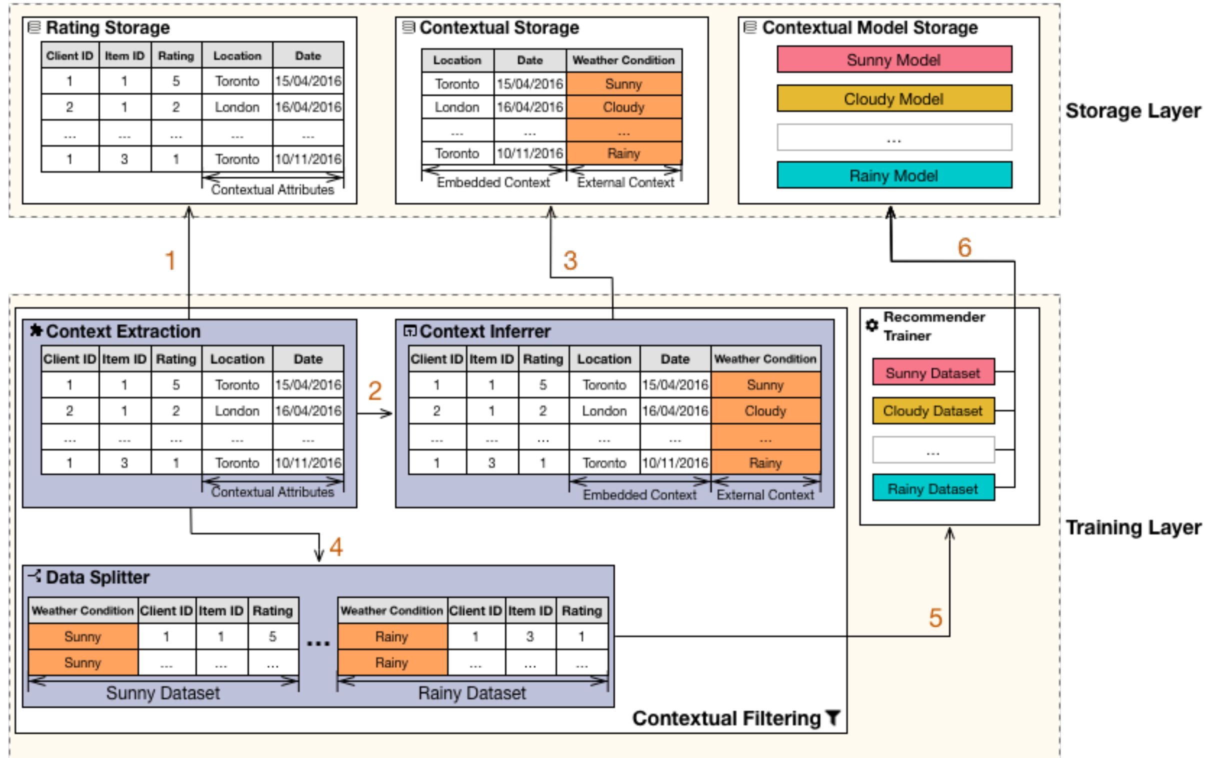

Figure 3.5: Training phase using the weather condition context.

model that can be used to create a list of recommendations. Instead of following this pattern, $(CF)^2$ uses contextual subsets of historical data to generate several contextual models.

Figure 3.5 illustrates the steps followed by $(CF)^2$ to process a historical dataset with its contextual attributes. Moreover, the *recommender trainer* uses these attributes to train the contextual models. This illustration of the *training phase* is expanded as follows:

1. The starting point of the *training phase* occurs within the *context extraction* sub-component. This sub-component requests the *rating storage* component to provide a list of the ratings stored in the *application*. Using the example illustrated in Figure 3.5, the list contains the *client identifier*, *item identifier*, *rating*, and contextual attributes *location* and *date*. Because the weather condition cannot be obtained directly from these attributes, the *contextual extraction* sub-component delegates the contextual inference task to the *context inferrer* sub-component;

2. The *context inferrer* sub-component either infers the contextual attributes using internal rules or reaches out to the *contextual storage* component to obtain the external knowledge. In this case, historical weather conditions provided by an external data source are required;

3. To obtain these weather conditions, the *context inferrer* sub-component reaches out to the *contextual storage* component. In this case, the *contextual storage* component provides a list containing the location and date followed by the weather condition;

4. With the list of contexts in place, the *context extraction* component invokes the *data splitter* sub-component to divide the rating dataset into smaller datasets, each of them representing a different context, in this case a weather condition;

5. For each contextual dataset generated, the *data splitter* sub-component invokes the *recommender trainer* component to train the collaborative filtering models effectively;

6. This results in the creation of a set of new models and auxiliary files that will be stored in the *contextual model storage* component.

An extended description of these components is presented below.

3.3.1 Contextual Filtering

Contextual filtering is responsible for identifying the contextual attributes of past requests and creating their corresponding training datasets.

This component initiates the *training phase* by requesting that all ratings captured during a period of time be sent to the *rating storage* component, regardless of their context. The proper contextual identification is then performed for each request in the dataset. For cases where a useful context cannot be directly obtained, this component also performs *contextual inference*. Once the contextual attributes have been identified, this component splits the historical data into contextual subsets. Finally, it invokes the *recommender trainer* component to train each contextual collaborative filtering models. An example of this process is illustrated by arrows 1, 2, 3, and 4 in Figure 3.5.

Because these tasks are well defined, $(CF)^2$ proposes the creation of three sub-components to create a better separation of concerns. These sub-components, which are already present in Figure 3.5, are described below.

Context Extraction

To achieve the contextual filtering goal of this component, the first step is to extract a list of contextual attributes available in the *rating storage* component and remove those that are not useful for training purposes.

The exact contextual attributes used are domain-specific, but (although this is not mandatory) the decision on which ones to use should always be preceded by an analysis of past user

behaviour. This step determines whether the chosen context attributes affect user behaviour, thereby ensuring a better use of $(CF)^2$.

It is important to realize that these contextual attributes may not be sufficient to conduct a proper contextual separation of the historical dataset. It may be necessary to perform a *contextual inference* to obtain the proper contextual attributes. In these cases, this component delegates the *contextual inference* process to the *context inferrer* sub-component.

With the contextual attributes identified, the *context extraction* sub-component can then invoke the *data splitter* sub-component to conclude the splitting process.

Context Inferrer

This optional sub-component is responsible for inferring contextual attributes that are not explicitly available in the historical dataset.

These inferred contexts can be as simple as using the date of the request to determine whether it was made during a weekday or on the weekend, or they can be more complex, such as using location and time to infer whether a request was made during the day or at night, taking sunrise and sunset times for different seasons into consideration.

Although this is not mandatory, these inferences may require access to an external knowledge base. In these cases, the *contextual storage* is contacted in to provide the desired information. This interaction is exemplified by arrow 3 in Figure 3.5.

By using this sub-component, the contextual information previously available is extended in numerous ways, enabling $(CF)^2$ to make use of several new contexts to perform data splitting process.

Data Splitter

With the contextual attributes properly identified, the only task left for the *contextual filtering* component to fulfill its goal is to perform the splitting itself. This is the responsibility of the data splitter sub-component.

Invoked by the *context extraction* sub-component, each identified context attribute is used to perform splitting on the historical dataset, resulting in several subsets matching the desired context.

It is worth mentioning that the data present in the *application* or in *rating storage* are not altered by $(CF)^2$ at any point because all the filtering is done in memory or using auxiliary storage within the component. The choice of location depends on the implementation.

To conclude, each of the split datasets is then forwarded to the *recommender trainer* component so that the latter can perform training using only the ratings performed within the desired

context.

3.3.2 Recommender Trainer

The last component of the *training layer*, the *recommender trainer* component, is responsible for training the recommender model using the subsets provided. This process is invoked for each contextual attribute and can be carried out using one of the many collaborative filtering techniques available. After training takes place, the model, along with its contextual attributes, is forwarded to the *contextual model storage* to be stored and used during the *production phase*. This flow is exemplified by arrows 5 and 6 in Figure 3.5, which illustrates the generated contextual models and their storage.

By using a technology-agnostic approach, $(CF)^2$ ensures that the best technique can be used for each application domain, enabling its adoption by a wide gamut of applications.

3.4 Production Layer

The *production layer* is responsible for inspecting all incoming requests made to the *recommender service* and generating a list of personalized *recommendations* that match the requested content.

To accomplish this task, this layer is divided into two components. The *request inspector* component is responsible for inspecting all incoming requests in order to identify their contextual attributes. The *recommender engine* component is responsible for choosing the proper model to produce the *recommendations* and for generating them.

During the *production phase*, a traditional *MB-CFRS* is involved in using the model obtained during the *training phase* to create a list of recommended items for a user or a list of users for a certain item. $(CF)^2$ can do this because it uses several contextual models instead of one.

Figure 3.6 illustrates the steps followed by $(CF)^2$ for an incoming request during the *production phase*. In this case, the incoming request has its contextual attribute inferred by the *request inspector* component, and the *recommender engine* uses this attribute to select the proper model to generate the recommendations list. This illustration of the *production phase* is further described below.

1. The *production phase* starts every time a *client* issues a request to the *recommender service*. Using the example illustrated in Figure 3.6, the request contains the *client identifier*, *item identifier*, and contextual attributes *location* and *date*;

2. Because the *recommender service* processes only those requests that need a list of recommendations to provide content, it delegates the request to the *context extraction* sub-component of the *request inspector* component;

3. The *context extraction* sub-component extracts the contextual attributes present in the request. In this example, because the weather condition cannot be obtained directly from *location* and *date*, the *contextual extraction* sub-component delegates the contextual inference task to the *context inferrer* sub-component;

4. The *context inferrer* sub-component either infers the contextual attributes using an internal rule or reaches out to the *contextual storage* component to obtain external knowledge. In this case, weather condition requires the use of an external datasource. To obtain this weather condition, the *context inferrer* sub-component reaches out to the *contextual storage* component for the current weather condition given a location and a date;

5. With the context properly identified, the *context extraction* sub-component invokes the *context chooser* sub-component of the *recommender engine* component. The purpose of this invocation is to obtain the corresponding contextual model along with its auxiliary files;

6. Because the actual contextual models are stored in the *contextual model storage* component, the *context chooser* sub-component fetches the model from the *contextual model storage* component;

7. The retrieved model with its auxiliary files is then passed to the *predictor* sub-component, which queries the model for the list of recommendations;

8. This list of recommendations is then returned to the client.

An extended description of these two components is presented below.

3.4.1 Request Inspector

The first component of the *production phase*, the *request inspector* component, performs a proper contextual identification of the incoming requests and, for those cases where a useful context cannot be directly obtained, also performs contextual inference. Once the contextual attributes have been identified, the *request inspector* invokes the *recommender engine* to generate the list of *recommendations* requested by the *client*.

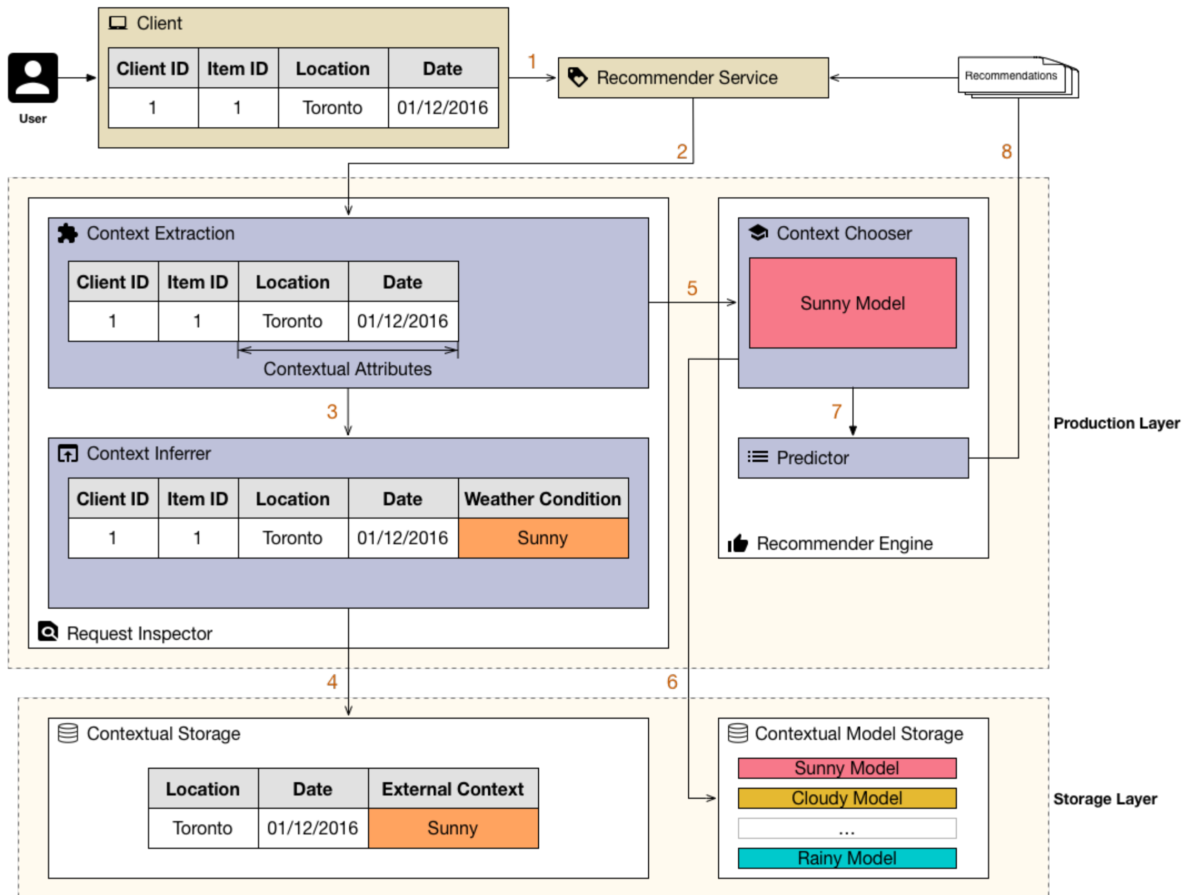

Figure 3.6: Production phase using the weather condition context.

Because these tasks are well defined and very different, $(CF)^2$ separates them into two sub-components, much as was done in the *contextual filtering* component of the *training layer*. These sub-components are described below.

Context Extraction

Similarly to the sub-component with the same name present in the *contextual filtering* component of the *training layer*, the *context extraction* component is responsible for extracting all contextual attributes for a request, but instead of working with past usage data, this component deals with requests made during the *production phase*.

The exact contextual attributes are domain-specific, but they should be the same ones defined in the *training layer*.

Because the context may require inference of contextual attributes, it may be necessary to carry out *contextual inference* to obtain the proper contextual attributes. In these cases, this component delegates the *contextual inference* process to the *context inferrer* sub-component.

With the contextual attributes identified, the *context extraction* sub-component then delegates the remaining work to the *recommender engine* component.

Context Inferrer

Implementation of this component often relies on the same logic as the *context inferrer* sub-component of the *contextual filtering* component. This is the case because often the inference rules are the same, independently of whether the request was performed in the past or the present. In those rare cases where they differ, the *context inferrer* sub-component of the *production phase* should reflect these changes.

3.4.2 Recommender Engine

The last component of the *production layer*, the *recommender engine* component is responsible for generating the list of *recommendations* that is returned to the *client*. To achieve this goal, the component must choose the proper recommender model and use it to predict which *recommendations* are most suitable for the current request.

These tasks were divided into two sub-components, each of them being responsible for a part of this procedure.

Context Chooser

The *context chooser* sub-component uses the contextual attributes identified by the *request inspector* component to query the *contextual model storage* component for the matching recommender model. This query returns the corresponding model to be used by the next sub-component to generate the list of *recommendations* requested by the *client*.

Predictor

With the appropriate model in place, the only thing left to do to obtain the list of *recommendations* is to query the model for recommendations. Because a recommendation can be a list of items for a user or a list of users for a certain item, this sub-component needs to determine which recommendation type was requested and provide the list to the *recommender service*.

3.5 Summary

This chapter introduced the $(CF)^2$ architecture and discussed each of its components. The discussion provided an explanation of the responsibilities of each layer and components of the architecture and also explained how they interact with each other. Because the architecture operates in two phases, each phase was presented individually, along with a description of the steps to be performed. Finally, this chapter also introduced the contextual inference concept, which enables the architecture to use inference rules or external knowledge to provide extended contextual attributes to the dataset.

Chapter 4

$(CF)^2$ Framework

This chapter proposes a framework for the *(CF)*^{2} architecture that can be used as a foundation for implementing the proposed work in different programming languages and paradigms.

The framework uses an object-oriented approach design to provide a class diagram (Figure 4.1) for each architectural component and sub-component and to describe the methods implemented by each component or sub-component. This chapter also explains the inputs and outputs structure and presents the algorithms at a high level. An exceptions is made for the methods responsible for identifying contextual attributes. This logic often relies on the application domain and changes for each deployed system. This framework uses packages to represent layers and components containing sub-components. It also uses constants to manage workflow and prevent declaration redundancy.

The following subsections details the *(CF)*^{2} layers, components, and sub-components.

4.1 Storage Layer

This section provides the implementation details for each architectural component present in the *storage layer*.

4.1.1 Rating Storage

The *rating storage* component is responsible for providing an interface between the $(CF)^2$ architecture and the historical dataset, which is stored in the *application* and accessed through an Application Programming Interface (API). Because each *application* provides a different API, the logic to access this dataset varies.

This logic must be implemented using a method called *getDataBetweenPeriods*, which receives two parameters: a starting date and time, and a final date and time. This method

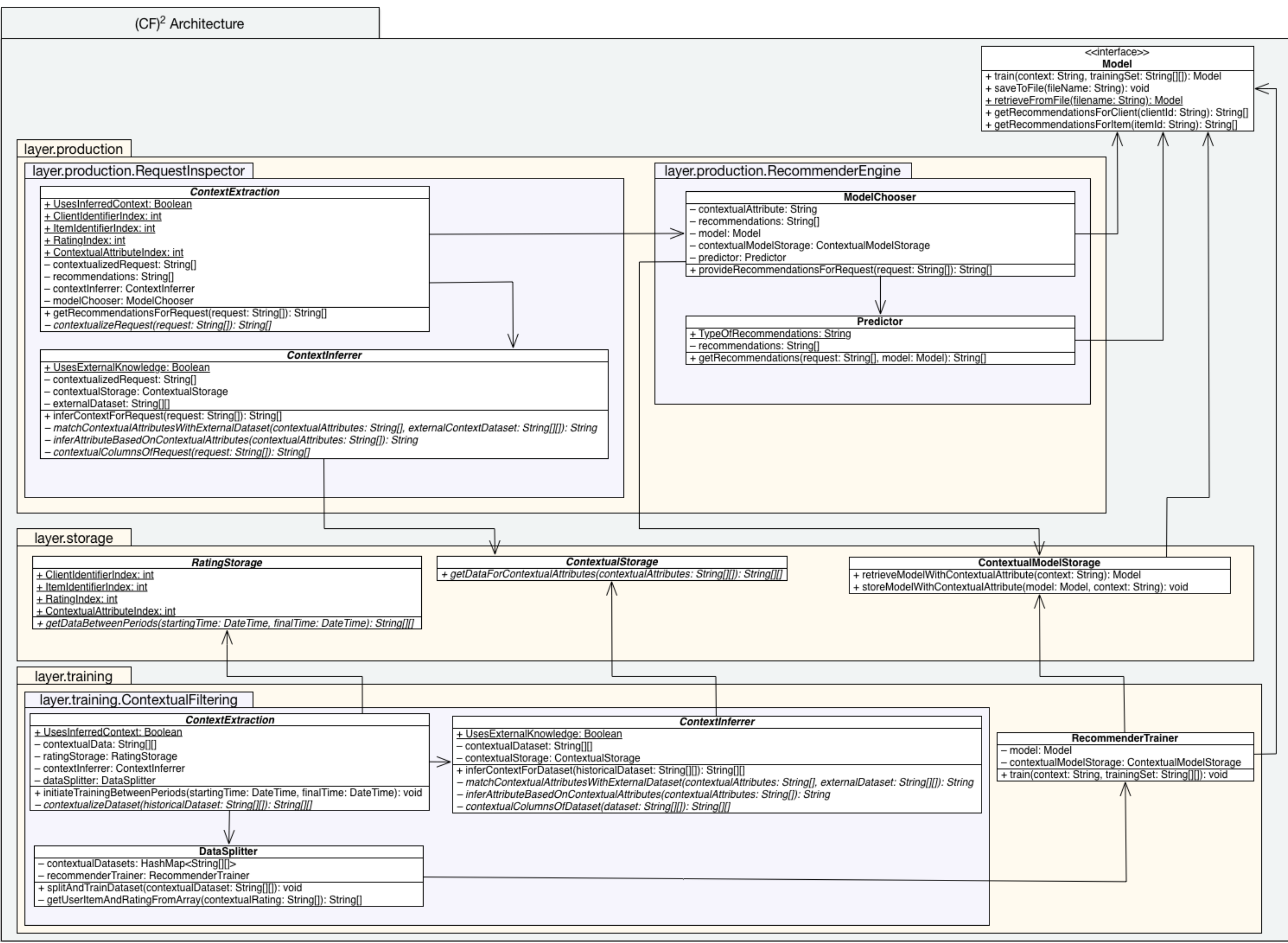

Figure 4.1: Class diagram for the *(CF)*^{2} architecture.

Table 4.1: Column structure of the matrix returned by the *rating storage* component.

Column	Description
1	Client Identification
2	Item Identification
3	Rating
4	Contextual Attribute 1
...	...
n+3	Contextual Attribute *n*

Table 4.2: Generic data source structure used by the *contextual storage* component.

Column	Description
1	Contextual attribute 1
...	...
n	Contextual attribute *n*
n+1	External Contextual Attribute

submits a request to the *application* for a copy of the historical data between the two specified points in time, which returns this data as a matrix structured as in Table 4.1. Each request to this historical dataset is represented as a line in the returned matrix, and the columns of this matrix should follow the structure presented on Table 4.1.

Because the implementation of this method varies according to the dataset, an example of the implementation of this method will be presented for each case study.

4.1.2 Contextual Storage

The *contextual storage* component is responsible for providing an interface between the architecture and the external contextual dataset. Because each external contextual dataset is accessed differently, the logic to access them varies.

This logic must be implemented in a method called *getDataForContextualAttributes*, which receives a matrix containing contextual attributes as parameters. This method submits a request to the external service for extended contextual data that match those passed as parameters and returns the data as a matrix containing these contextual attributes and their corresponding external values. As with the *rating storage*, each line of the matrix corresponds to one data entry, and the columns follow the structure presented in Table 4.2.

Because the implementation of this method varies according to the dataset, an implementation will be presented for each case study.

4.1.3 Contextual Model Storage

The *contextual model storage* component is responsible for storing the most recent models generated by the *recommender trainer* and for providing an instance of these models when requested by the *recommender engine*. To do so, it implements two methods: *storeModelWithContextualAttribute* and *retrieveModelWithContextualAttribute*.

The *storeModelWithContextualAttribute* method receives two parameters, the model to be saved and a string containing the contextual attribute. This implementation opted to store the model as a file named after the contextual attribute. This ensures that only the most recent model is used. This procedure is outlined in Algorithm 4.1. The auxiliary methods *FileExists* and *DeleteFile* are usually provided by the programming language.

Algorithm 4.1: *storeModelWithContextualAttribute* method of the *contextual model storage* component

```
Input: Model model to be persisted
       String context containing the contextual attribute of the model
1 if FileExists(context) is true then
2 |   DeleteFile(context);
3 end
4 model.saveToFile(context);
```

The *retrieveModelWithContextualAttribute* method receives a single string parameter containing the contextual attribute. This implementation loads the model from a file named after the contextual attribute, therefore ensuring that the most recent model is always used for each contextual attribute. This procedure is outlined in Algorithm 4.2.

Algorithm 4.2: *retrieveModelWithContextualAttribute* method of the *model storage* component

```
Input  : String context containing the contextual attribute of the model
Output: Model model
1 model ←null;
2 if FileExists(context) is true then
3 |   model ←Model.retrieveFromFile(context);
4 end
5 return model;
```

4.2 Training Layer

This section provides the implementation details for each architectural component present in the *training layer*.

4.2.1 Contextual Filtering

This component is separated into three sub-components, which are described below.

Context Extraction

The *context extraction* sub-component is responsible for initiating the training process, as well as for identifying the contextual attribute to be used as a filtering criterion. Because each application is different, identification of the contextual attribute varies among applications, with each requiring its identification logic to be implemented in its own method.

Granted that obtaining the contextual attribute may require the use of *contextual inference*, the implementation defines the constant *UsesInferredContext* to identify which logic to execute. This constant is of *Boolean* type and has the value *true* when an inference is necessary or *false* when the contextual attribute can be directly obtained.

This sub-component implements two methods: *initiateTrainingBetweenPeriods* and *contextualizeDataset*. The first method is responsible for initiating the training process, whereas the second contains the contextual attribute identification logic. When a *contextual inference* is necessary, the *contextualizeDataset* method is not implemented.

The *initiateTrainingBetweenPeriods* method receives a starting date and time and a final date and time as parameters and does not return any value. The procedure performed by this method is outlined by Algorithm 4.3.

The *contextualizeDataset* method receives the historical dataset in the format supplied by the *getDataBetweenPeriods* method of the *rating storage* component and returns a matrix striped of the columns containing contextual attributes that are different from the one used as a filtering criterion. The column structure is presented in Table 4.3.

Because each application performs this logic differently, this methods must be addressed on a case-by-case basis. An implementation of this method will be presented in the case study section.

Context Inferrer

This optional sub-component is required only when the application uses *contextual inference* to infer the contextual attribute to be used as a filtering criterion. Because each application is

Algorithm 4.3: *initiateTrainingBetweenPeriods* method of the *contextual extraction* sub-component

Input : DateTime startingTime containing the starting date and time
DateTime finalTime containing the final date and time
Constant: Boolean UsesInferredContext determining whether the algorithm should make use of contextual inference

```
1  ratingStorage ←RatingStorage.new();
2  historicalDataset
     ←ratingStorage.getDataBetweenPeriods(startingTime,finalTime);
3  if UsesInferredContext is true then
4  |   contextInferrer ←ContextInferrer.new();
5  |   contextualData
   |     ←contextInferrer.inferContextForDataset(historicalDataset);
6  else
7  |   contextualData ←contextualizeDataset(historicalDataset);
8  end
9  dataSplitter ←DataSplitter.new();
10 dataSplitter.splitAndTrainDataset(contextualData);
```

Table 4.3: Column structure of the matrix supplied to the *contextualizeDataset* method.

Column	Description
1	Client Identification
2	Item Identification
3	Rating
4	Contextual attribute to be used as filtering criterion

Table 4.4: Column structure of the matrix returned by the *inferContextForDataset* method.

Column	**Description**
1	Client Identification
2	Item Identification
3	Rating
4	Inferred contextual attribute to be used as filtering criteria

different, the inference logic for the contextual attribute varies among applications, with each requiring its inference logic to be implemented in its own method.

Granted that the contextual inference may require the use of external database, the implementation defines the constant *UsesExternalKnowledge* to identify which logic to execute. This constant is of *Boolean* type and has the value *true* when an external knowledge is necessary or *false* when the *contextual inference* can be obtained locally.

This sub-component implements three methods: *inferContextForDataset*, *matchContextualAttributesWithExternalDataset*, and *inferAttributeBasedOnContextualAttributes*. The first method is responsible for initiating the contextual inference process, the second contains the logic that determines how to match existing contextual attributes with the obtained external knowledge, and the last contains the contextual inference logic.

Only the *inferContextForDataset* method and one of the other two methods need to be implemented. When external knowledge is needed, the method to be implemented is *matchContextualAttributesWithExternalDataset*, whereas in other cases, the method to be implemented is *inferAttributeBasedOnContextualAttributes*.

The *inferContextForDataset* method receives the historical dataset in the format supplied by the *getDataBetweenPeriods* method of the *rating storage* component and returns a matrix containing the client identification, the item identification, the rating, and the inferred contextual attribute. The column structure is presented in Table 4.4, and the procedure performed by this method is outlined as Algorithm 4.4.

The *matchContextualAttributesWithExternalDataset* method receives two parameters: an array of contextual attributes in the format presented in Table 4.5, and the external contextual dataset in the same format supplied by the *getDataForContextualAttributes* method of the *context storage* component. This method returns the external contextual attribute used as a filtering criterion.

The *inferAttributeBasedOnContextualAttributes* method receives as parameters an array of contextual attributes in the format presented in Table 4.5 and returns the inferred contextual

Algorithm 4.4: *inferContextForDataset* method of the *context inferrer* sub-component

```
    Input    : Matrix historicalDataset containing the historical data with contextual
               attributes
    Output   : Matrix containing the client identification, the item identification, the
               rating, and the inferred contextual attribute
    Constant: Boolean UsesExternalKnowledge determining whether the algorithm
               uses external knowledge
 1 /* InitializeMatrix() auxiliary method to create empty matrix */
 2 contextualDataset ←InitializeMatrix();
 3 /* contextualColumnsOfDataset() is a method that returns only
      the contextual columns of a given dataset */
 4 contextualAttributes ← contextualColumnsOfDataset( historicalDataset);
 5 if UsesExternalKnowledge is true then
 6 |   contextualStorage ←ContextualStorage.new();
 7 |   externalDataset
 |      ←contextualStorage.getDataForContextualAttributes(contextualAttributes);
 |
 8 |   for line in historicalDataset do
 9 |   |   contextualDataset [line][RatingStorage.ClientIdentifierIndex] ←
   |   |     historicalDataset [line][RatingStorage.ClientIdentifierIndex];
10 |   |   contextualDataset [line][RatingStorage.ItemIdentifierIndex] ←
   |   |     historicalDataset [line][RatingStorage.ItemIdentifierIndex];
11 |   |   contextualDataset [line][RatingStorage.RatingIndex] ← historicalDataset
   |   |     [line][RatingStorage.RatingIndex];
12 |   |   contextualDataset [line][RatingStorage.ContextualAttributeIndex] ←
   |   |     matchContextualAttributesWithExternalDataset(
   |   |     contextualAttributes [line], externalDataset);
13 |   end
14 else
15 |   for line in historicalDataset do
16 |   |   contextualDataset [line][RatingStorage.ClientIdentifierIndex] ←
   |   |     historicalDataset [line][RatingStorage.ClientIdentifierIndex];
17 |   |   contextualDataset [line][RatingStorage.ItemIdentifierIndex] ←
   |   |     historicalDataset [line][RatingStorage.ItemIdentifierIndex];
18 |   |   contextualDataset [line][RatingStorage.RatingIndex] ← historicalDataset
   |   |     [line][RatingStorage.RatingIndex];
19 |   |   contextualDataset [line][RatingStorage.ContextualAttributeIndex] ←
   |   |     inferAttributeBasedOnContextualAttributes(
   |   |     contextualAttributes [line]);
20 |   end
21 end
22 return contextualDataset;
```

Table 4.5: Structure supplied to the *matchContextualAttributesWithExternalDataset* method.

Column	Description
1	Contextual attribute 1
...	...
n	Contextual attribute *n*

attribute to be used as a filtering criterion.

Because each implementation performs this logic differently, this methods must be addressed on a case-by-case basis. The implementation of this method will be presented in the case study section.

Data Splitter

The *data splitter* sub-component is responsible for splitting the historical dataset into contextual sub sets, and invoking the *recommender trainer* to train the models. To achieve this goal, this components implements the *splitAndTrainDataset* method.

This method receives as parameter a matrix containing requests with client identification, item identification, rating, and the contextual attribute that serves as a filtering criterion. This is the same format supplied by the *contextualizeDataset* method of the *context extraction* sub-component or the *inferContextForDataset* method of the *context inferrer* sub-component.

This method does not return any value, and the procedure is performs is outlined as Algorithm 4.5.

4.2.2 Recommender Trainer

The *recommender trainer* component is responsible for training and sustaining the contextual collaborative filtering models. This is achieved by implementing the *train* method, which is outlined as Algorithm 4.6.

This method receives two parameters, a string containing the contextual attribute, and a matrix containing the requests, each accompanied by its client identifier, item identifier, and rating. The column structure of these parameters is presented in Table 4.6.

This method does not return any value and concludes the training process.

Algorithm 4.5: *splitAndTrainDataset* method of the *data splitter* sub-component

Input : Matrix contextualDataset containing the client identification, item identification, rating, and the contextual attribute that will serve as a filtering criterion

Constant: Integer RatingStorage.ContextualAttributeIndex containing the array index that contains the contextual attribute

```
1  /* InitializeHashWithDefaultValueOfMatrix() auxiliary method to
      create hashmap objects with default value of matrix         */
2  contextualDatasets ←InitializeHashWithDefaultValueOfMatrix();
3  for line in contextualDataset do
4  |   /* getUserItemAndRatingFromArray() is a method that returns
   |      the userId, itemId, and rating from an array            */
5  |   contextualDatasets
6  |      .get(line [RatingStorage.ContextualAttributeIndex])
7  |      .add(getUserItemAndRatingFromArray(line));
8  end
9  for (key, value) of contextualDatasets do
10 |   recommenderTrainer ← RecommenderTrainer.new();
11 |   recommenderTrainer.train(key,value);
12 end
```

Algorithm 4.6: *train* method of the *recommender trainer* component

Input : String context containing the contextual attribute of the model
Matrix trainingSet containing the client identification, item identification, and rating

```
1  model ←Model.new();
2  model.train(context, trainingSet);
3  contextualModelStorage ←ContextualModelStorage.new();
4  contextualModelStorage.storeModelWithContextualAttribute(model,context);
```

Table 4.6: Structure of the requests passed as parameters to the *train* method.

Column	Description
1	Client Identification
2	Item Identification
3	Rating

4.3 Production Layer

This section provides the implementation details for each architectural component present in the *production layer*.

4.3.1 Request Inspector

This component is separated into two sub-components, which are described below.

Context Extraction

The *context extraction* sub-component is called whenever the *recommender service* is in need of *recommendations*. This sub-component is responsible for identifying the contextual attribute that determines which contextual collaborative filtering model to use. Because each application is different, the identification of the contextual attribute varies among applications, with each application requiring identification logic to be implemented in its own method.

Granted that the contextual attribute may require *contextual inference*, the implementation defines the constant *UsesInferredContext* to identify which logic to execute. This constant is of *Boolean* type and has the value *true* when an inference is necessary or *false* when the contextual attribute can be obtained directly.

This sub-component implements two methods: *getRecommendationsForRequest* and *contextualizeRequest*. The first method is responsible for initiating the recommendation process, whereas the second contains the contextual identification logic. In cases where a *contextual inference* is necessary, the contextualization is delegates to the *context inferrer* sub-component, hence the *contextualizeRequest* method is not implemented.

The *getRecommendationsForRequest* method receives as parameter a request in the form of an array containing the client identifier, item identifier, and contextual attributes and returns a list of recommendations. The parameters supplied follow the structure of Table 4.1, and the procedure performed by this method is outlined as Algorithm 4.7.

The *contextualizeRequest* method receives as its only parameter the same request as the *getRecommendationsForRequest* and returns a contextualized request, in the form of an array containing the client identifier, item identifier, and contextual attribute, as presented in Table 4.7.

Because each implementation performs this logic differently, this methods must be addressed on a case-by-case basis.

Algorithm 4.7: *getRecommendationsForRequest* method of the *contextual extraction* sub-component

Input : Array request containing the client identification, item identification, and contextual attributes
Output : List of recommendations
Constant: Boolean UsesInferredContext determining whether the algorithm should make use of contextual inference

```
1  contextualizedRequest ←null;
2  if UsesInferredContext is true then
3  |   contextInferrer ←ContextInferrer.new();
4  |   contextualizedRequest
   |     ←contextInferrer.inferContextForRequest(request);
5  else
6  |   contextualizedRequest ←contextualizeRequest(request);
7  end
8  modelChooser ←ModelChooser.new();
9  recommendations ← modelChooser.provideRecommendationsForRequest(
     contextualizedRequest);
10 return recommendations;
```

Table 4.7: Structure of the requests returned by the *contextualizeRequest* method.

Column	Description
1	Client Identification
2	Item Identification
3	Contextual attribute that determines which model to use

Context Inferrer

This optional sub-component is required only when the application uses the *contextual inference* concept and is responsible for inferring the contextual attribute to be used as a filtering criterion. Because each application is different, the inference logic for the contextual attribute varies among applications, requiring each application's inference logic to be implemented in its own method.

Granted that the contextual inference may require the use of external knowledge, the implementation defines the constant *UsesExternalKnowledge* to identify which logic to execute. This constant is of *Boolean* type and has the value *true* when an external knowledge is necessary or *false* when the *contextual inference* can be obtained locally.

This sub-component implements three methods: *inferContextForRequest*, *matchContextualAttributesWithExternalDataset*, and *inferAttributeBasedOnContextualAttributes*. The first method is responsible for initiating the contextual inference process, the second contains the logic that matches existing contextual attributes with obtained external knowledge, and the last contains the contextual inference logic.

Only the *inferContextForRequest* method and one of the other two methods need to be implemented. When external knowledge is needed, the method to be implemented is *matchContextualAttributesWithExternalDataset*, whereas in those other cases, the method to be implemented is *inferAttributeBasedOnContextualAttributes*.

The *inferContextForRequest* method receives as parameter a request in the form of an array containing the client identifier, item identifier, and contextual attributes, and returns an array containing the client identification, the item identification, and the inferred contextual attribute. The column structure of this parameter is presented in Table 4.7, and the procedure performed by this method is outlined as Algorithm 4.8.

The *matchContextualAttributesWithExternalDataset* method receives two parameters: an array of contextual attributes in the format presented in Table 4.5, and the external contextual dataset in the same format as supplied by the *getDataForContextualAttributes* method of the *context storage* component. This method returns the external contextual attribute to be used as a filtering criterion.

The *inferAttributeBasedOnContextualAttributes* method receives as parameter an array of contextual attributes in the format presented in Table 4.5 and returns the inferred contextual attribute to be used as a filtering criterion.

Because each implementation performs these logics differently, these methods must be addressed on a case-by-case basis.

Algorithm 4.8: *inferContextForRequest* method of the *context inferrer* sub-component

Input : Array request containing the client identification, item identification, and contextual attributes
Output : Array containing the client identification, the item identification, the rating, and the inferred contextual attribute
Constant: Boolean UsesExternalKnowledge determining whether the algorithm uses external knowledge
Integer ContextExtraction.ClientIdentifierIndex containing the array index that contains the client identifier
Integer ContextExtraction.ItemIdentifierIndex containing the array index that contains the item identifier
Integer ContextExtraction.RatingIndex containing the array index that contains the rating
Integer ContextExtraction.ContextualAttributeIndex containing the array index that contains the contextual attribute

```
1  /* InitializeArray() auxiliary method to create empty array */
2  contextualizedRequest ←InitializeArray();
3  contextualizedRequest [ContextExtraction.ClientIdentifierIndex] ← request
     [ContextExtraction.ClientIdentifierIndex];
4  contextualizedRequest [ContextExtraction.ItemIdentifierIndex] ← request
     [ContextExtraction.ItemIdentifierIndex];
5  contextualizedRequest [ContextExtraction.RatingIndex] ← request
     [ContextExtraction.RatingIndex];
6  if UsesExternalKnowledge is true then
7     /* contextualColumnsOfRequest() is a method that returns only
         the contextual columns of a given request */
8     contextualAttributes ←contextualColumnsOfRequest(request);
9     contextualStorage ←ContextualStorage.new();
10    externalDataset ← contextualStorage.getDataForContextualAttributes(
        [contextualAttributes ]);
11    contextualizedRequest [ContextExtraction.ContextualAttributeIndex] ←
        matchContextualAttributesWithExternalDataset(
        contextualAttributes, externalDataset);
12 else
13    contextualizedRequest [ContextExtraction.ContextualAttributeIndex] ←
        inferAttributeBasedOnContextualAttributes(contextualAttributes);
14 end
15 return contextualizedRequest;
```

4.3.2 Recommender Engine

This component is separated into two sub-components as described below.

Model Chooser

The *model chooser* sub-component is responsible for selecting the model and requesting the *predictor* sub-component to predict the best recommendations for the incoming request.

This logic is implemented using a method called *provideRecommendationsForRequest*, which receives as parameter a contextualized request in the form of an array containing the client identifier, item identifier, and contextual attribute. This is the same format supplied by the *contextualizeRequest* method of the *context extraction* sub-component or the *inferContextForRequest* method of the *context inferrer* sub-component. As return value, this method returns the list of recommendations provided by the *getRecommendations* method of the *predictor* sub-component.

The procedure performed by this method is outlined as Algorithm 4.9.

Algorithm 4.9: *provideRecommendationsForRequest* method of the *model chooser* sub-component

Input : Array request containing the client identification, item identification, rating, and contextual attribute
Output : List of recommendations
Constant: Integer ContextExtraction.ContextualAttributeIndex containing the array index that contains the contextual attribute

```
1 contextualAttribute ←request [ContextExtraction.ContextualAttributeIndex];
2 contextualModelStorage ← ContextualModelStorage.new();
3 model ← contextualModelStorage.retrieveModelWithContextualAttribute(
    contextualAttribute);
4 predictor ← Predictor.new();
5 /* getUserItemAndRatingFromArray() is a method that returns the
      userId, itemId, and rating from an array                     */
6 recommendations ← predictor.getRecommendations(
    getUserItemAndRatingFromArray( request), model);
7 return recommendations;
```

Predictor

The *predictor* sub-component is responsible for querying the model for a prediction on the best recommendations for the incoming request.

This logic is implemented in a method called *getRecommendations*, which receives as parameters a request in the form of the model to be used and an array containing the client identifier and the item identifier. This method returns a list of recommendations that are judged best by the model.

Because recommendations can be a list of items for a client or a list of clients for a certain item, the recommendations returned to the client can be of any of either type. This implementation defines the constant *TypeOfRecommendations* to identify which logic to execute. This constant is of *string* type and has the value *item* when recommendations of items are desired or *client* when recommendations of clients are desired.

The procedure performed by this method is outlined as Algorithm 4.10.

Algorithm 4.10: *getRecommendations* method of the *predictor* sub-component

Input : Array request containing the client identification, item identification, and rating
Model model to fetch recommendations
Output : List of recommendations
Constant: Boolean TypeOfRecommendations determining what type of recommendation should be provided by the model
Integer ContextExtraction.ClientIdentifierIndex containing the array index that contains the client identifier
Integer ContextExtraction.ItemIdentifierIndex containing the array index that contains the item identifier

```
1 recommendations ←null;
2 if TypeOfRecommendations is "Client" then
3   recommendations ←model.getRecommendationsForClient(request
      [ContextExtraction.ClientIdentifierIndex]);
4 else
5   recommendations ←model.getRecommendationsForItem(request
      [ContextExtraction.ItemIdentifierIndex]);
6 end
7 return recommendations;
```

4.4 Summary

This chapter proposed a framework for the $(CF)^2$ architecture and discussed each of the components that are part of this framework. The discussion provided an algorithmic description of the methods of each layer and component of the architecture, and also explained how they interact with each other.

Chapter 5

Evaluation

This chapter presents an evaluation of the *(CF)*^{2} architecture using two case studies. Because both experiments were conducted using data sources belonging to the same application domain, a contextualization of the domain is provided. Moreover, the methodology used to conduct the evaluation process is presented. To conclude, each case study is introduced and evaluated, providing an analysis of how the proposed architecture performs compared to the traditional method.

In this research, the case studies were evaluated in the context of a Web site dedicated to provide personalized recommendations of pages to its clients. For this purpose, past interactions between clients and the service were used to train the collaborative filtering models.

These past interactions were provided by a multi-media company specialized in weather- and traveller-related content and technology. The $(CF)^2$ architecture was evaluated by analysing the traffic on their Web site. Traffic was captured in clickstream form by two on-line marketing tools and Web analytics applications: *Google Analytics*[1] and *Omniture*[2], an *Adobe*[3] company.

The task being evaluated is defined in the literature as *Find Good Items*. In this task, the recommender system is interested in suggesting items to a user, but displaying only those that are a "best bet". Because on-line evaluation was not a viable option for this research, evaluation was conducted through an off-line evaluation containing only real data (not synthesized).

One characteristic of recommender systems based on clickstream data is the implicit nature of the ratings. Because an explicit rating is not provided by users, an implicit rating r_{ui} of 1 is used to indicate that a user u likes the requested page i. Moreover, because *MB-CFRS* models need enough data to generate good recommendations, r_{ui} with contextual attributes representing less than 0.1% of the total dataset were purged. This procedure was necessary to

[1] http://google.com/analytics
[2] http://www.omniture.com
[3] http://www.adobe.com

prevent results provided by sparse models from contaminating the evaluation process.

5.1 Methodology

The evaluation scenario consists of a dataset D divided into two subsets, a training set T and a validation set V. The training set represents 80% of the dataset and is obtained by random selection from the original dataset without repetition. The remaining 20% represents the validation set. The sets are then saved as different *comma-separated value* (*CSV*) files.

Because the *(CF)²* architecture proposes the use of contextual datasets to train *CF* models, each contextual attribute c is represented by a training subset T_c representing ratings of T containing the contextual attribute c. Moreover, because this research aims to prove that the use of contextual attributes as filtering criteria is better than random dataset reduction, the training subset T_r^c represents a randomly selected subset of T with the same size as T_c. This process is illustrated in Figure 5.1.

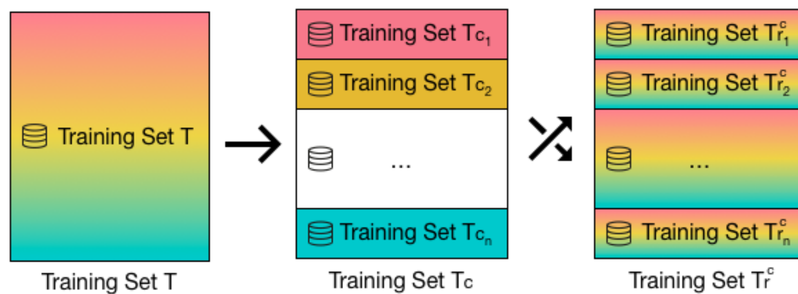

Figure 5.1: Training set T segmented by contextual attributes and random dataset reduction.

Similarly, each contextual attribute c is represented by a validation subset V_c representing ratings of V containing the contextual attribute c, and the subset V_r^c represents a randomly selected subset of V with the same size as V_c.

Moreover, because the evaluation is accomplished by comparing the proposed architecture with the traditional method, the subsets T_t and V_t (subsets of T and V respectively) represent the average rating r_{ui} given by a user u to a page i, regardless of context. Both sets are also saved as different *CSV* files and will be used to train and validate the traditional *CF* model m_t.

To assess the predictive quality of the models, this research used *predictive accuracy metrics*. These metrics measure how close the predictions made by the recommender system are to the true user rating. The metric used is the *mean squared error* (*MSE*), which is given by the equation

$$\text{MSE(V)} = \frac{1}{|V|} \cdot \sum_{(u,i)\in V} (r_{ui} - \hat{r}_{ui})^2 \tag{5.1}$$

Table 5.1: Notations used throughout the chapter.

Symbol	Description
r_{ui}	Rating given by a user u to page i
$\hat{r}_{ui}$	Predicted rating given for a user u of page i
D	Dataset containing ratings given by a user u to page i in context c
T	Subset of D containing random elements without repetition and where $\lvert T\rvert = 0.8 \cdot \lvert D\rvert$
T_u	Subset of T containing the average rating r_{ui} given by a user u to page i
T_c	Subset of T containing ratings given in a context c
T_r^c	Subset of T containing random ratings without repetition and where $\lvert T_r^c\rvert = \lvert T_c\rvert$
V	Set of $D - T$
V_u	Subset of V containing the average rating r_{ui} given by a user u to page i
V_c	Subset of V containing ratings given in a context c
V_r^c	Subset of V containing random ratings without repetition and where $\lvert V_r^c\rvert = \lvert V_c\rvert$
m_u	*CF* model trained using set T_u
m_c	*CF* model trained using set T_c
m_r^c	*CF* model trained using set T_r^c

where V is the validation set, $|V|$ is the size of the set V, r_{ui} is the true user rating, and $\hat{r}_{ui}$ is the predicted rating. These and other notations used throughout the chapter are presented in Table 5.1. The adoption of *MSE* over other accuracy metrics was based on the efficiency of its calculation.

Using the implementation provided in Chapter 4, the training dataset T (available as a *CSV* file) is then used to feed the training process by means of the *rating storage* component. This step ensures the creation of a different model m_c for each contextual attribute c present in T. This process is illustrated in Figure 5.2.

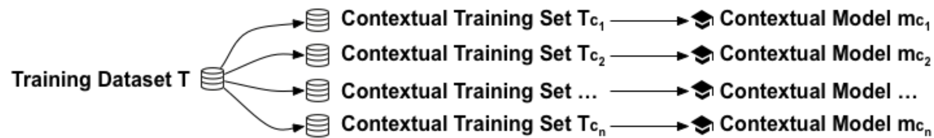

Figure 5.2: Training set T segmented into subsets T_c to train model m_c.

Validation is then performed by splitting the validation set V into different contextual subsets V_c and using these subsets to compare the predicted rating $\hat{r}_{ui}$ with the rating r_{ui} available in the validation set V_c. The average of the sums of the squared differences of $\hat{r}_{ui}$ and r_{ui} provides the *MSE* for the contextual model m_c. This process is illustrated in Figure 5.3.

Validation of the models m_t, representing the traditional method, and m_r^c, representing ran-

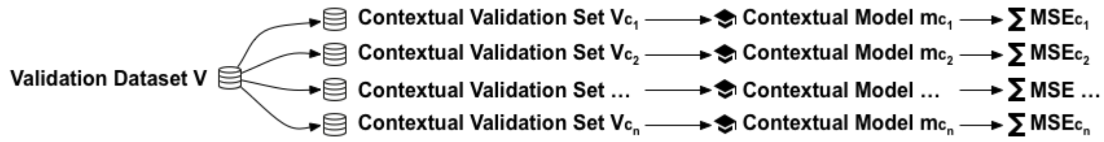

Figure 5.3: Validation set V segmented into subsets V_c to predict ratings using model m_c.

dom dataset reduction, is performed differently. Because the architecture has the option of determining the context using the method *inferAttributeBasedOnContextualAttributes* of the *context inferrer* sub-component, an implementation that always returns the same context regardless of input is used, ensuring that all ratings in the training set are classified as being part of the same context.

By using this adaptation, each set of T_t and T_r^c is used as a data source for the *rating storage* component, resulting in training of the models m_t and m_r^c respectively. Validation of these models is done in the same way as for the models m_c, but using the validation set V_t for model m_t and V_r^c for model m_r^c.

Furthermore, because the training time of each model is different, the training time values (in milliseconds) are then used to compare the two approaches.

Because this architecture can use two types of contextual attributes, embedded and inferred, the evaluation process was divided in two case studies, each covering one of these types.

5.2 Case Studies

The $(CF)^2$ evaluation was performed by means of two case studies. Each case study used its own dataset and different types of contextual attributes. The first case study used contextual attributes embedded in the data source, whereas the second case study used *contextual inference*.

To conduct the evaluation, the implementation described in Chapter 4 was adapted to use the functional paradigm of the *Scala* language and the large-scale data processing properties of the *Spark* engine. The resulting implementation is available in a public repository[4].

For the *CF* recommendation engine, both case studies used the *matrix factorization technique* [57] using the *alternating least squares* [57] (*ALS*) algorithm. This has already been implemented by *Spark's spark.mllib* machine learning library, providing an "out of the box" solution that can process large volumes of data. Moreover, this implementation includes a special training technique based on the work done by Hu *et al.* [22], which specializes in training *CF* models using implicit ratings.

[4]https://github.com/dennisbachmann/cf2-scala

Table 5.2: Parameters used to train the *CF* models.

Parameter	Value
λ (Regularization Parameter)	100
Number of Features	20
Number of Iterations	10
α (Confidence Level)	1500

Because the application domains for both case studies were the same, this decision did not have a significant impact on the evaluation. These values were obtained after performing a cross-validation with partitions of the T dataset. Various configurations of the regularization parameter (λ), number of features, number of iterations, and confidence level (α) were considered. The parameter values that resulted in a minimum stable *MSE* were chosen and are given in Table 5.2.

An attempt to execute the experiments on Amazon's EC2 cloud computing services was made. The cluster was configured with four nodes of type "m4.large", meaning that each node used 2 virtual processors and 8 gigabytes of memory. After 2,152 hours of utilization, the cluster did not yield any result. As a consequence, the decision to run the experiments on a more powerful machine was made. Therefore, each case study was executed on a private server with 24 Cores Intel Xeon E5-2630 2.3 GHz and 96 GB RAM DDR3 1600 MHz running Ubuntu 14.04.2 LTS.

5.2.1 Case Study 1: Embedded Context

The first case study used clickstream data captured by the *Google Analytics* tool during the summer of 2016 (June 20 to September 22) to create recommender models based on the contextual attribute "operating system with platform". To achieve this goal, the data were exported as a *CSV* file and pre-processed to remove entries captured by the clickstream that did not represent a page view. These entries usually represent interactions with objects inside a Web page that do not trigger a page change, like interaction with map objects or social media snippets. After this step, the resulting dataset was filtered to contain only unique values with the following properties:

- Visitor identifier
- Uniform Resource Locator (URL)
- Operating System

- Platform is mobile (true or false)

This process resulted in a dataset containing 130,994,883 unique samples. Because the dataset does not contain the desired contextual attribute as a single attribute, the *operating system* property must be combined with the *platform* property. This is achieved by the logic outlined in Algorithm 5.1.

Algorithm 5.1: Logic to obtain the contextual attributes for case study 1

Input : String operatingSystem containing the operating system of the request
Boolean isMobile containing whether the platform is mobile or not
Output: The computed contextual attribute

```
1 contextualAttribute ←"";
2 if isMobile is true then
3 |  contextualAttribute ←contextualAttribute.append("MOBILE - ");
4 else
5 |  contextualAttribute ←contextualAttribute.append("NOT MOBILE - ");
6 end
7 contextualAttribute.append(operatingSystem);
8 return contextualAttribute;
```

This logic was then implemented in the *contextualizeDataset* and *contextualizeRequest* methods of the *context extraction* sub-components in the *training layer* and *production layer*. The constant *UsesInferredContext* was set to *false*, forcing the algorithm to execute these methods.

Because the evaluation methodology specifies that contextual attributes representing less than 0.1% of the total dataset should be purged, the 309,948 entries classified under this category had to be removed. Moreover, because of the sensitive nature of this information, all contextual names were anonymized to the format "OS/Platform". The last step before exporting the dataset was to inject the rating of 1 into each tuple. The final dataset containing 130,684,845 unique samples was then split into a training set T and a validation set V. Each set was then exported as a *CSV* file. The final size of each dataset classified by contextual attribute and already anonymized is displayed in Table 5.3.

The training set was then used to create the contextual models m_c, m_r^c, and m_t, and to calculate the time spent to train them. Moreover, the validation set was used to calculate the *MSE* of each model. This process was repeated 30 times, and the final analysis used the average values of these properties.

Table 5.3: Size of each dataset classified by contextual attribute.

Contextual Attribute	Full Dataset Size	Training Set Size	Validation Set Size
OS/Platform 1	17,099,871	13,681,058	3,418,813
OS/Platform 2	151,710	120,902	30,808
OS/Platform 3	2,464,423	1,971,104	493,319
OS/Platform 4	1,623,950	1,300,121	323,829
OS/Platform 5	65,221,859	52,179,154	13,042,705
OS/Platform 6	385,096	307,823	77,273
OS/Platform 7	923,827	738,438	185,389
OS/Platform 8	626,817	501,195	125,622
OS/Platform 9	11,812,359	9,452,167	2,360,192
OS/Platform 10	30,374,933	24,300,878	6,074,055
Total	130,684,845	104,552,840	26,132,005

Results and discussion

The performance of the $(CF)^2$ architecture in the context of this case study is shown in Table 5.4 and graphically represented in Figure 5.4. The *MSE* obtained for the traditional method, represented by model m_t, is compared with each model m_c, and its equivalent model m_r^c , obtained by random reduction. To facilitate interpretation, the *MSE* for the model m_t is displayed as a dotted reference line.

The analysis conducted on these values indicates that using the proposed architecture often provides better results than the traditional method, and significantly better results than using random dataset reduction. This is also corroborated by the average *MSE* value for each dataset reduction technique when compared with the traditional method, as displayed in Figure 5.5.

Taking dataset size into consideration, the analysis shows that when using contextual dataset reduction, the *MSE* values tend to fluctuate around the value obtained when using the traditional method, regardless of the dataset size. The same is not true for the random dataset reduction. When using this technique, the error increases as the dataset becomes smaller and converges to the accuracy of the traditional method as the dataset increases in size. This is to be expected because there is almost no dataset reduction when the size approaches the original one. This situation is shown in Figure 5.6, which also has the *MSE* value for the model m_t as a dotted reference line.

Furthermore, the times (in milliseconds) taken to train each model m_t, m_t, and m_r^c were compared. The values are shown in Table 5.5, and illustrated in Figure 5.7.

These values indicate that the time to train each model increases almost linearly as the

Table 5.4: *MSE* values obtained using dataset reduction.

Contextual Attribute	Contextual Reduction	Random Reduction
OS/Platform 1	0.2527/0.2671	0.5854/0.2671
OS/Platform 2	0.2676/0.2671	1.0764/0.2671
OS/Platform 3	0.2179/0.2671	0.9121/0.2671
OS/Platform 4	0.2017/0.2671	0.9705/0.2671
OS/Platform 5	0.2499/0.2671	0.3542/0.2671
OS/Platform 6	0.2192/0.2671	1.0295/0.2671
OS/Platform 7	0.2032/0.2671	1.0156/0.2671
OS/Platform 8	0.2476/0.2671	1.0308/0.2671
OS/Platform 9	0.2253/0.2671	0.6624/0.2671
OS/Platform 10	0.2823/0.2671	0.4763/0.2671

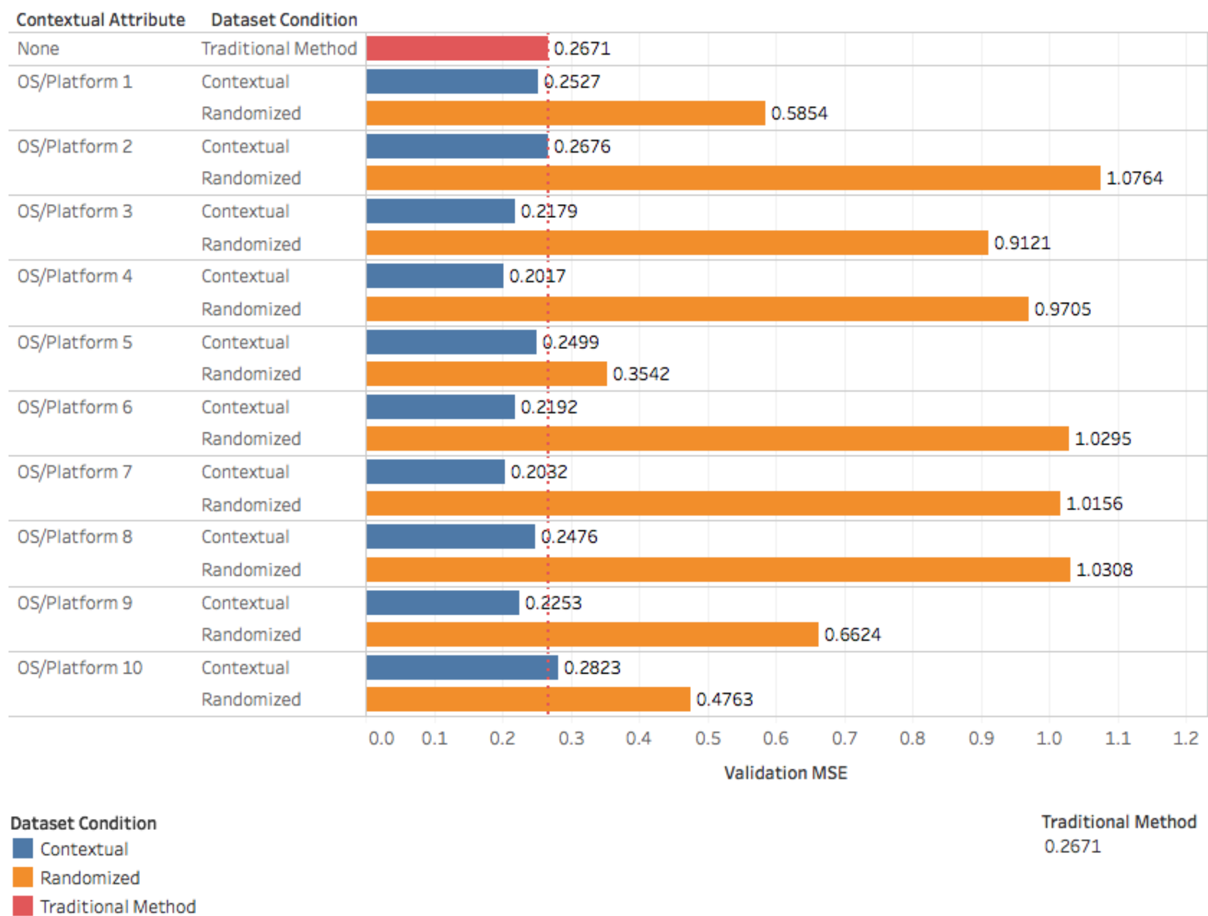

Figure 5.4: *MSE* of m_t, m_c, and m_r^c segmented by dataset.

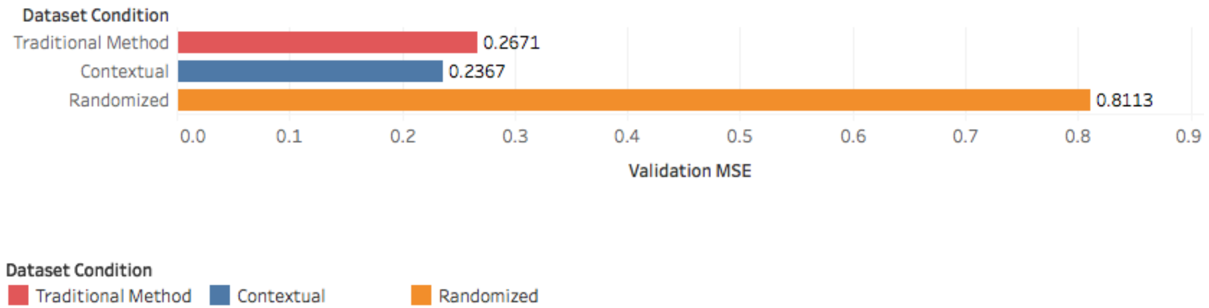

Figure 5.5: Average *MSE* for the traditional method and each dataset reduction technique.

Table 5.5: Training time (in milliseconds) taken to train each contextual model.

Contextual Attribute	Contextual Reduction	Random Reduction
OS/Platform 1	394,317/1,448,614	516,530/1,448,614
OS/Platform 2	135,793/1,448,614	139,874/1,448,614
OS/Platform 3	171,586/1,448,614	189,306/1,448,614
OS/Platform 4	167,061/1,448,614	180,242/1,448,614
OS/Platform 5	836,524/1,448,614	1,129,829/1,448,614
OS/Platform 6	166,619/1,448,614	171,285/1,448,614
OS/Platform 7	174,793/1,448,614	185,463/1,448,614
OS/Platform 8	171,915/1,448,614	175,023/1,448,614
OS/Platform 9	264,369/1,448,614	365,504/1,448,614
OS/Platform 10	606,478/1,448,614	691,034/1,448,614

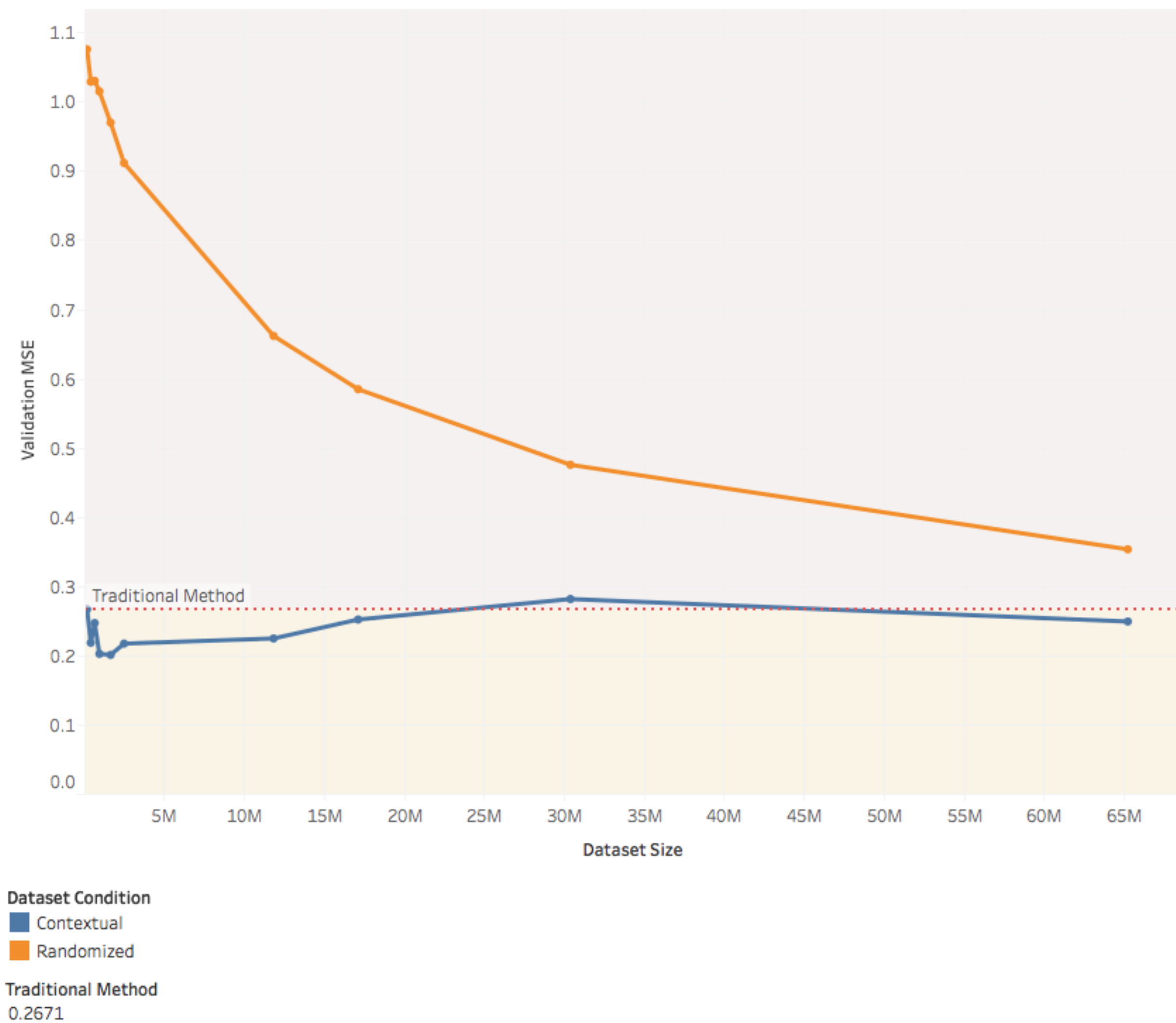

Figure 5.6: *MSE* value by dataset size for each dataset reduction technique.

dataset size increases. This is especially true in the case of *Spark's* implementation of *ALS*, but other *matrix factorization* implementations should also experience a significant reduction in training time when dataset reduction is used. Moreover, the contextual dataset reduction technique requires less training time than the random dataset reduction technique.

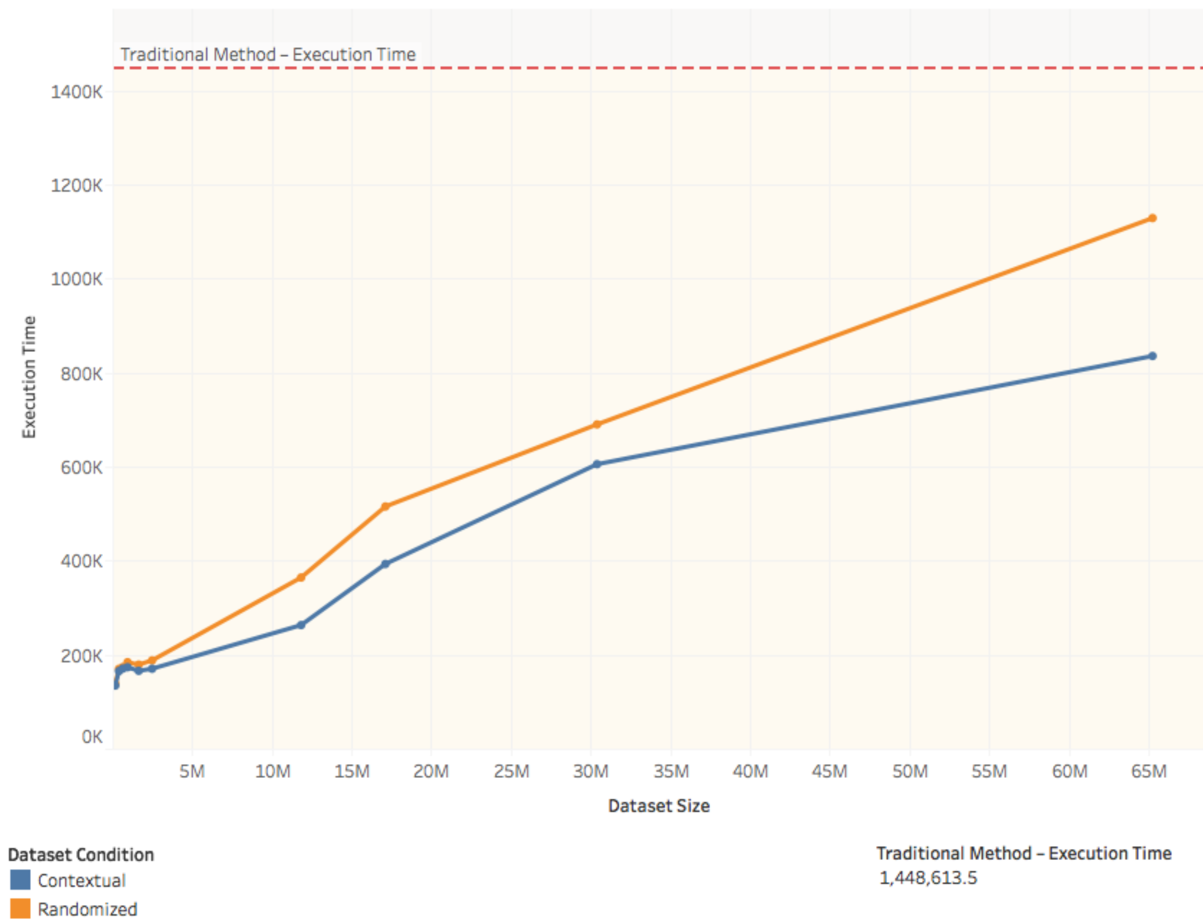

Figure 5.7: Training time in milliseconds taken to train each model.

5.2.2 Case Study 2: Contextual Inference

The second case study used clickstream data captured by the *Omniture* tool. The data was collected during the period of April 1st 2015 to June 30th 2015 and includes only visits generated by users in London, ON, Canada. Recommender models were created based on the contextual attribute "weather condition". Similarly to the previous case study, the data were exported as a *CSV* file and pre-processed to remove entries captured by the clickstream that did not represent a page view. After this step, the resulting dataset was filtered to contain only unique values with the following properties:

- Visitor identifier
- Uniform Resource Locator (URL)
- Time of Access
- Location

This process resulted in a dataset containing 7,729,696 samples. Because the dataset does not contain the desired contextual attribute, the *time of access* property along with the *location* property were used to obtain the weather condition of each visit. After removing the duplicate entries, the dataset containing the tuples *visitor identifier*, *URL*, and *weather condition* was reduced to 3,185,660 entries. Because the evaluation methodology specifies that contextual attributes representing less than 0.1% of the total dataset should be purged, the 3,852 entries classified under this category had to be removed. The last step before exporting the dataset containing 3,181,808 unique samples was to inject the rating of 1 into each tuple. The final dataset was then split into a training set T and a validation set V. Each set was then exported as a *CSV* file. The final size of each dataset classified by contextual attribute is displayed in Table 5.6.

The training set was then used to create the contextual models m_c, m_r^c, and m_t, and to calculate the time spent to train them. Moreover, the validation set was used to calculate the *MSE* of each model. This process was repeated 30 times, and the final analysis used the average values of these properties.

Results and discussion

The performance of the $(CF)^2$ architecture in the context of this case study is shown in Table 5.7 and graphically represented in Figure 5.8. The *MSE* obtained for the traditional method, represented by model m_t, is compared with each model m_c, and its equivalent model m_r^c, obtained

Table 5.6: Size of each dataset classified by contextual attribute.

Contextual Attribute	**Full Dataset Size**	**Training Set Size**	**Validation Set Size**
Clear	120,430	96,511	23,919
Fog	32,945	26,437	6,508
Haze	7,241	5,832	1,409
Heavy Thunderstorms and Rain	9,478	7,564	1,914
Light Drizzle	49,873	40,000	9,873
Light Rain	105,339	84,186	21,153
Light Rain Showers	241,165	192,911	48,254
Light Snow Grains	10,159	8,163	1,996
Light Snow Showers	61,258	48,943	12,315
Light Thunderstorms and Rain	30,470	24,344	6,126
Missing	9,164	7,315	1,849
Mostly Cloudy	1,112,850	890,189	222,661
Overcast	758,930	607,310	151,620
Partly Cloudy	314,711	251,800	62,911
Rain	5,719	4,574	1,145
Rain Showers	14,002	11,216	2,786
Scattered Clouds	262,744	210,320	52,424
Shallow Fog	11,711	9,369	2,342
Thunderstorm	10,979	8,678	2,301
Thunderstorms and Rain	12,640	10,126	2,514
Total	3,181,808	2,545,788	636,020

Table 5.7: *MSE* values obtained using dataset reduction.

Contextual Attribute	Contextual Reduction	Random Reduction
Clear	0.2356/0.1485	0.5601/0.1485
Fog	0.1894/0.1485	0.7857/0.1485
Haze	0.2094/0.1485	0.9302/0.1485
Heavy Thunderstorms and Rain	0.1061/0.1485	0.9491/0.1485
Light Drizzle	0.1767/0.1485	0.7672/0.1485
Light Rain	0.1710/0.1485	0.5688/0.1485
Light Rain Showers	0.1911/0.1485	0.4072/0.1485
Light Snow Grains	0.2183/0.1485	0.9472/0.1485
Light Snow Showers	0.1837/0.1485	0.6613/0.1485
Light Thunderstorms and Rain	0.1384/0.1485	0.7599/0.1485
Missing	0.2314/0.1485	0.8465/0.1485
Mostly Cloudy	0.1641/0.1485	0.2059/0.1485
Overcast	0.1738/0.1485	0.2446/0.1485
Partly Cloudy	0.2258/0.1485	0.3647/0.1485
Rain	0.2005/0.1485	0.9974/0.1485
Rain Showers	0.1970/0.1485	0.9577/0.1485
Scattered Clouds	0.2353/0.1485	0.3930/0.1485
Shallow Fog	0.2591/0.1485	0.9044/0.1485
Thunderstorm	0.1496/0.1485	0.8584/0.1485
Thunderstorms and Rain	0.1999/0.1485	0.9239/0.1485

by random reduction. To facilitate interpretation, the *MSE* for the model m_t is displayed as a dotted reference line.

The analysis conducted on these values indicate that $(CF)^2$ yields similar results to the traditional method, while outperforming the accuracy obtained when using random dataset reduction. Moreover, results indicate that accuracy improves when weather conditions deteriorate. Most compelling evidence is the *MSE* obtained for the "heavy thunderstorms and rain" weather condition. This happens because under these circumstances the rating pattern among users are shared. In this case study, users have the tendency to check the same Web pages during bad weather.

Despite the fact that there were improvements, the obtained results indicate that weather condition is not a suitable contextual attribute for this application. This is also corroborated by the average *MSE* values displayed in Figure 5.9. Nevertheless, the use of contextual attribute as filtering criterion for local learning is more appropriate than random selection.

Further analysis indicates that navigation behaviour remains similar among contexts, resulting in contextual models built with scattered points. To put in another way, the contextual

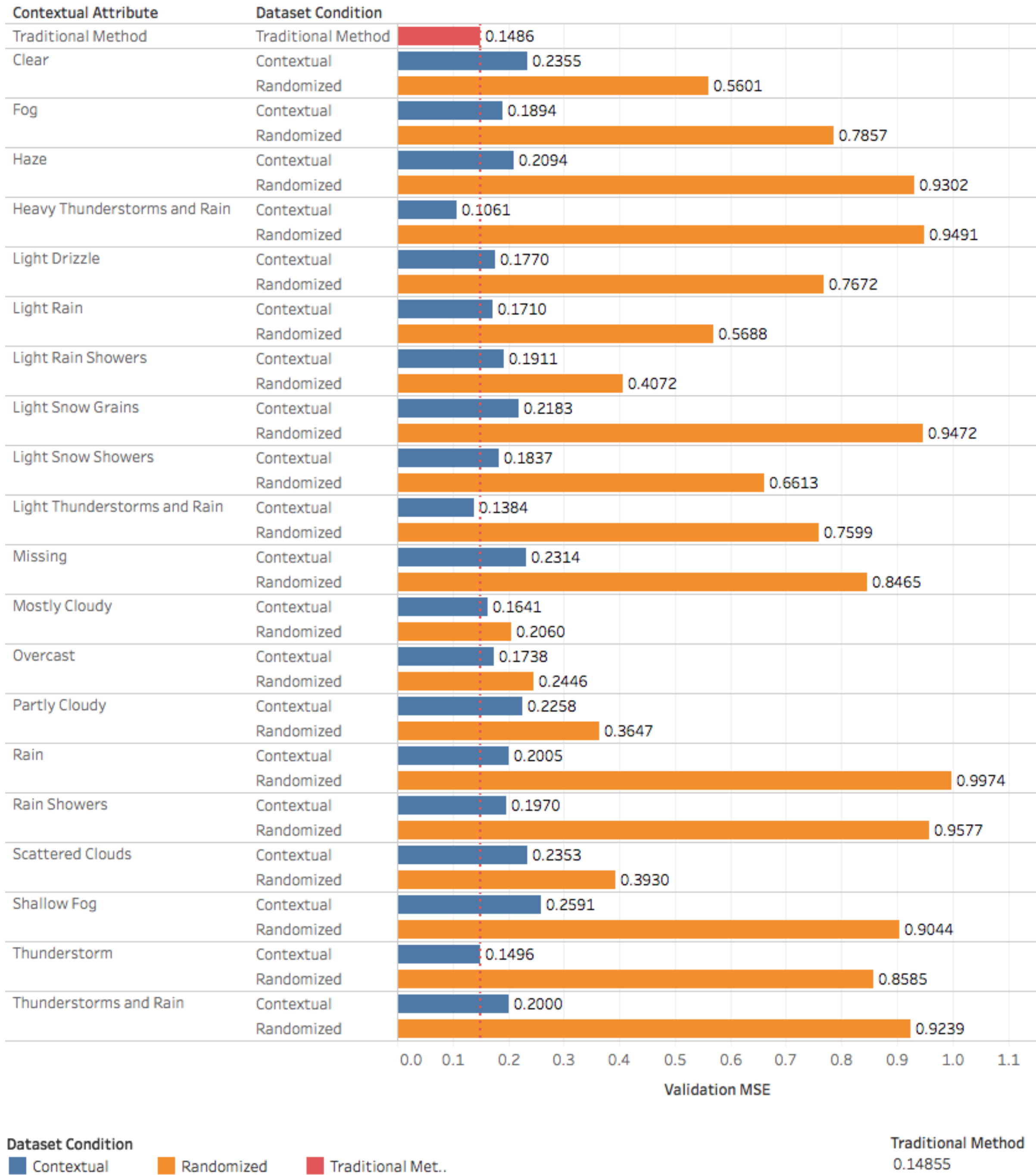

Figure 5.8: *MSE* of m_t, m_c, and m_r^c segmented by dataset.

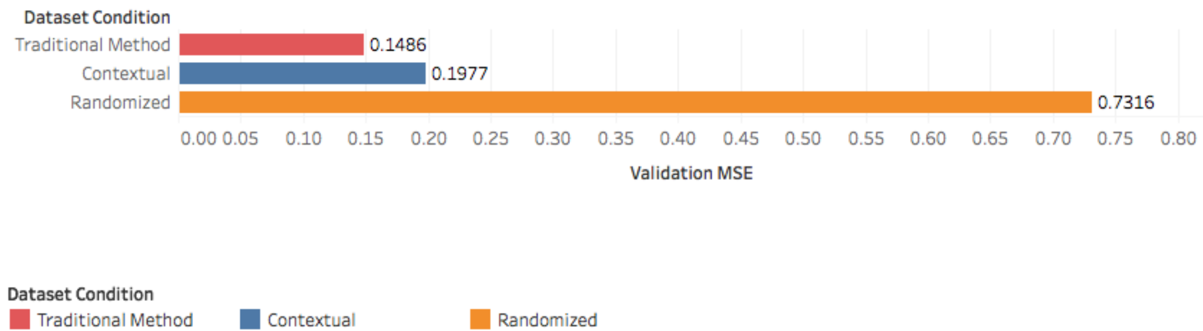

Figure 5.9: Average *MSE* for the traditional method and each dataset reduction technique.

attribute often served as an instance selection mechanism, rather than serving as a clustering factor towards local models.

Taking dataset size into consideration, the analysis shows that when using contextual dataset reduction, the *MSE* values tend to fluctuate around the value obtained when using the traditional method, regardless of the dataset size. The same is not true for the random dataset reduction. When using this technique, the error increases as the dataset becomes smaller and converges to the accuracy of the traditional method as the dataset increases in size. This is to be expected because there is almost no dataset reduction when the size approaches the original one. This situation is shown in Figure 5.10, which also has the *MSE* value for the model m_t as a dotted reference line.

Furthermore, the times (in milliseconds) taken to train each model m_t, m_t, and m_r^c were compared. The values are shown in Table 5.8, and illustrated in Figure 5.11.

These values indicate that the time to train each model increases almost linearly as the dataset size increases. This is especially true in the case of *Spark's* implementation of *ALS*, but other *matrix factorization* implementations should also experience a significant reduction in training time when dataset reduction is used. Moreover, the contextual dataset reduction technique requires less training time than the random dataset reduction technique.

Table 5.8: Training time (in milliseconds) taken to train each contextual model.

Contextual Attribute	Contextual Reduction	Random Reduction
Clear	6,907/83,869	10,331/83,869
Fog	4,208/83,869	4,479/83,869
Haze	1,741/83,869	2,405/83,869
Heavy Thunderstorms and Rain	2,332/83,869	2,658/83,869
Light Drizzle	3,044/83,869	9,355/83,869
Light Rain	6,582/83,869	8,745/83,869
Light Rain Showers	10,464/83,869	17,299/83,869
Light Snow Grains	2,991/83,869	2,852/83,869
Light Snow Showers	4,613/83,869	8,369/83,869
Light Thunderstorms and Rain	2,894/83,869	4,361/83,869
Missing	3,465/83,869	3,778/83,869
Mostly Cloudy	35,505/83,869	42,515/83,869
Overcast	29,036/83,869	33,740/83,869
Partly Cloudy	19,247/83,869	37,703/83,869
Rain	1,927/83,869	2,172/83,869
Rain Showers	2,753/83,869	3,118/83,869
Scattered Clouds	11,402/83,869	17,638/83,869
Shallow Fog	2,703/83,869	2,940/83,869
Thunderstorm	2,907/83,869	3,857/83,869
Thunderstorms and Rain	2,998/83,869	2,946/83,869

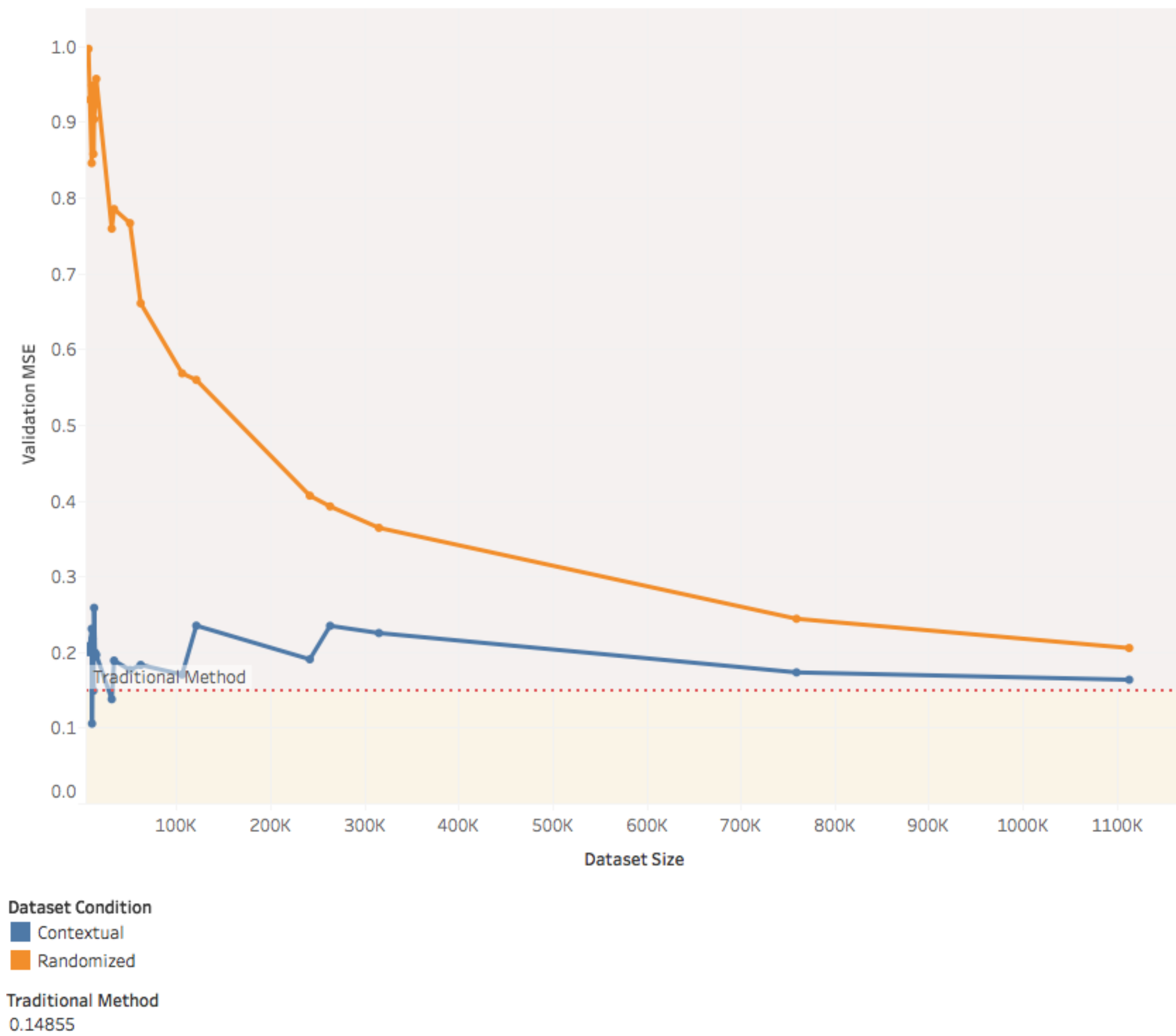

Figure 5.10: *MSE* value by dataset size for each dataset reduction technique.

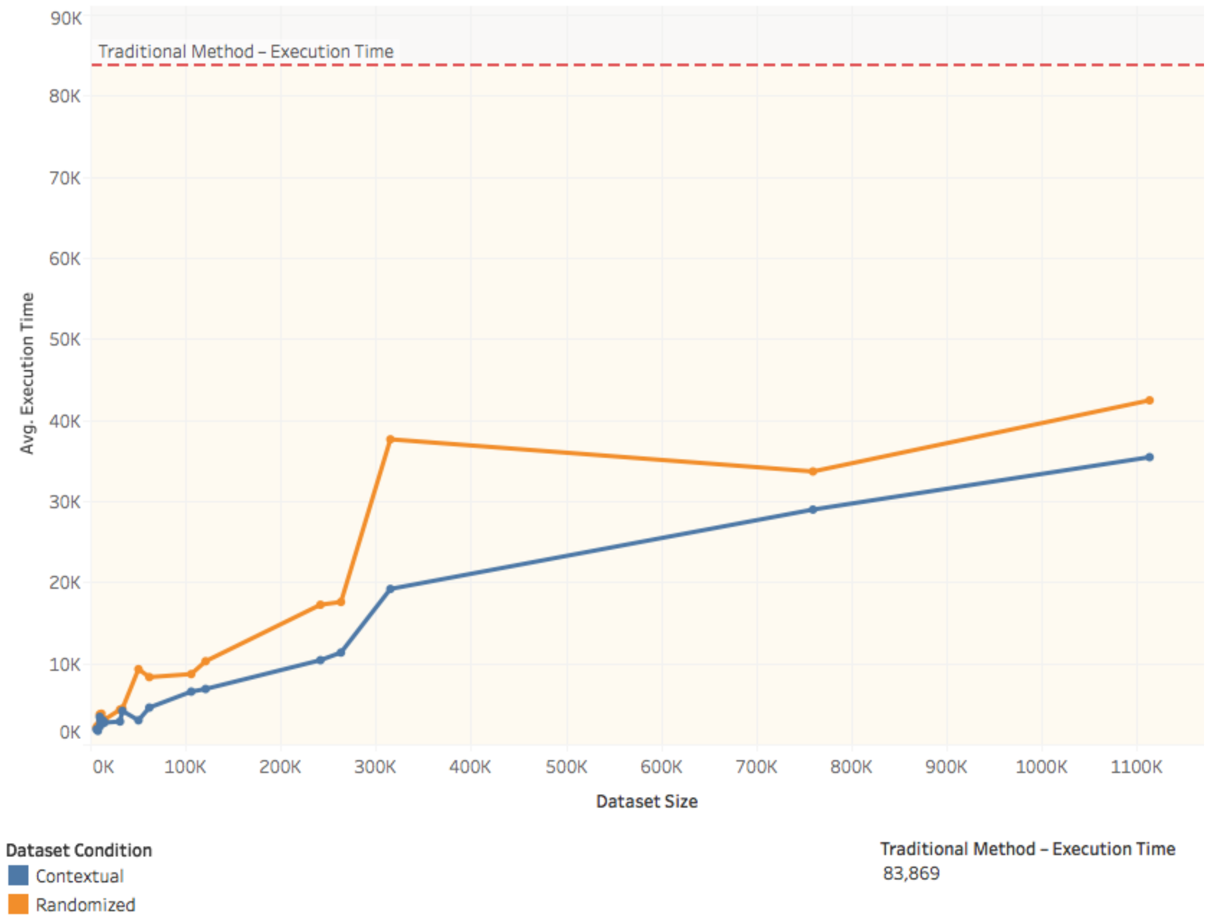

Figure 5.11: Training time in milliseconds taken to train each model.

5.3 Summary

In this chapter, the evaluation of the architecture described in Chapter 3 and the framework presented in Chapter 4 was presented. Moreover, the methodology used in the experiments as well as the results of the two case studies were discussed. In the first case study, the details of the implementation of *(CF)*^{2} using embedded contextual information was presented. Furthermore, a comparison between models created using contextual subsets, randomized subsets and the training set in its totality was presented. In the second case study, contextual inference was used to provide weather condition information to the dataset. Randomized training subsets were also created and a comparison between contextual, randomized and traditional models was presented. The results show that contextual models trained with a small fraction of the data resulted in better or similar accuracy when compared to the traditional method. Moreover, local learning using contextual information outperforms random selection in accuracy and in training time.

Chapter 6

Conclusions and Future Work

This chapter presents a concluding summary based on the contributions of the proposed context filtering for model-based collaborative filtering recommender system $(CF)^2$ architecture. In addition, a description of possible future research on the proposed frameworks will be presented.

6.1 Conclusions

The work described in this thesis presents the $(CF)^2$ architecture and uses local learning techniques to embed contextual awareness into collaborative filtering models. Typically, collaborative models are implemented with a focus on user-item interactions. By incorporating context into standard collaborative user-item models, recommender applications used in context-sensitive domains can provide recommendations with a better chance of being relevant to users. Moreover, local learning using context as a filtering criterion enables the scalability of recommender systems that use large datasets. An overview of the architecture has been presented, including its components, their roles, and their relationships.

The $(CF)^2$ architecture is also introduced in the form of a framework. This framework is presented at an algorithmic level and can serve as a foundation to implement the $(CF)^2$ architecture.

Because of the layered and orthogonal properties of $(CF)^2$, context-aware recommender systems can be implemented using one of the several widely available collaborative filtering libraries. Moreover, the concepts presented in this thesis do not restrict $(CF)^2$ implementation by enforcing a single machine language or paradigm, enabling the system builder to use the technology that best fits the needs of each application.

To demonstrate the applicability of the architecture and framework, this thesis also provided an implementation of the framework using the functional paradigm of the *Scala* language and

the large-scale data processing properties of the *Spark* engine. Using this implementation, two case studies were evaluated on the "*Find Good Items*" task. The first case study used embedded context, and the second used external knowledge by means of the contextual inference. Both case studies were evaluated using the same methodology, and their accuracy was compared against the traditional method. The results indicate that contextual models trained with a small fraction of the data gave better or similar accuracy than models trained with the full data set. In addition, local models created using random sampling instead of contextual dataset reduction were compared, and the results demonstrate that using contextual information outperforms random selection in accuracy and in training time.

6.2 Future Work

This section presents several areas of future work that can be explored:

- The $(CF)^2$ architecture presented in this study considers all samples in the training set to perform model training. Future work will consider using instance selection techniques to reduce dataset sizes even further. Although the architecture uses contextual information to perform local learning, integrating instance selection techniques would enhance scalability on training and prediction time even further.

- Another future project is to adapt $(CF)^2$ to incorporate the micro-services architectural style. Micro-services is an approach to software development that divides a single application into a combination of small services, each running in its own process and communicating linked data services with lightweight mechanisms. This communication often occurs by means of an HTTP resource API. Incorporation of micro-services will lead to componentization through services, increasing the decoupling of the architecture and facilitating deployment of the recommender application into the cloud.

- In this study, the proposed architecture was designed with the assumption that the implemented system runs in a secure environment and that users agree to have their ratings used by the system. Future work will explore privacy and security issues associated with recommender systems.

- Future work will explore the use of a fine-tuning component to ensure the best parameters for each contextual model. For instance, a model trained for one context may achieve better accuracy than another if the regularization parameter is modified. By using a cross-validation procedure for each contextual model generated, the best parameters for each model will be used to generate future recommendations.

- Given the challenges of using implicit feedback to generate recommendations, this work focused on this feedback to demonstrate that the proposed approach works even for the most challenging cases. To expand the architectural evaluation, case studies using explicit rating can be used.

- The framework implemented to run the evaluation was developed as a proof of concept. A future study will focus on performance improvements, taking advantage of *JVM* parameters and *Spark* functionalities to achieve a better distribution of workload among cluster nodes.

- In this study, contextual information is available as a single and flat attribute. Although composite contexts can always be expressed as fine-grained contexts, e.g., by concatenating all composites of a context, the use of hierarchical contexts is not supported by the architecture. Future work will implement the hierarchical nature of contexts to enhance recommendation quality. A starting point towards this direction would be the use of *Linked Data*, which uses published structured data to interlink semantic queries.

- To reduce the impact of sparse matrices generated by dataset reduction, future work can experiment with incorporating a rating estimation component. This component will attribute rating values to unrated items, hence reducing matrix sparsity.

Bibliography

[1] M. Balabanović and Y. Shoham, "Fab: Content-based, collaborative recommendation," *Communications of the ACM*, vol. 40, no. 3, pp. 66–72, 1997.

[2] X. Su, "Collaborative filtering: A survey," in *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*, p. 1, IEEE, September 2015.

[3] Y. Shi, M. Larson, and A. Hanjalic, "Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges," *ACM Computing Surveys*, vol. 47, pp. 3:1–3:45, May 2014.

[4] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 6, pp. 734–749, 2005.

[5] G. Adomavicius and A. Tuzhilin, *Context-Aware Recommender Systems*, pp. 217–253. Boston, MA: Springer US, 2011.

[6] U. Panniello, A. Tuzhilin, M. Gorgoglione, C. Palmisano, and A. Pedone, "Experimental comparison of pre- vs. post-filtering approaches in context-aware recommender systems," in *Proceedings of the Third ACM Conference on Recommender Systems*, RecSys '09, pp. 265–268, ACM, 2009.

[7] H. Liu and H. Motoda, *Instance selection and construction for data mining*, vol. 608. Springer Science & Business Media, 2013.

[8] L. Bottou and V. Vapnik, "Local learning algorithms," *Neural computation*, vol. 4, no. 6, pp. 888–900, 1992.

[9] J. Bennett and S. Lanning, "The netflix prize," in *KDD Cup and Workshop*, 2007. www.netflixprize.com.

[10] P. Resnick and H. R. Varian, “Recommender systems,” *Communications of the ACM*, vol. 40, pp. 56–58, March 1997.

[11] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, “Recommender systems survey,” *Knowledge-Based Systems*, vol. 46, pp. 109–132, July 2013.

[12] J. Lu, D. Wu, M. Mao, W. Wang, and G. Zhang, “Recommender system application developments: A survey,” *Decision Support Systems*, vol. 74, pp. 12 – 32, 2015.

[13] F. Ricci, L. Rokach, and B. Shapira, eds., *Recommender Systems Handbook*. Boston, MA: Springer US, 2015.

[14] Y. 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, pp. 426–434, ACM, 2008.

[15] G. Linden, B. Smith, and J. York, “Amazon. com recommendations: Item-to-item collaborative filtering,” *IEEE Internet computing*, vol. 7, no. 1, pp. 76–80, 2003.

[16] C. C. Aggarwal, *Neighborhood-Based Collaborative Filtering*, pp. 29–70. Cham: Springer International Publishing, 2016.

[17] C. C. Aggarwal, *An Introduction to Recommender Systems*, pp. 1–28. Cham: Springer International Publishing, 2016.

[18] J. B. Schafer, J. A. Konstan, and J. Riedl, *E-Commerce Recommendation Applications*, pp. 115–153. Boston, MA: Springer US, 2001.

[19] M. D. Ekstrand, F. M. Harper, M. C. Willemsen, and J. A. Konstan, “User perception of differences in recommender algorithms,” in *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys ’14, (New York, NY, USA), pp. 161–168, ACM, 2014.

[20] C. C. Aggarwal, *Model-Based Collaborative Filtering*, pp. 71–138. Cham: Springer International Publishing, 2016.

[21] C. C. Aggarwal, *Data mining: the textbook*. Springer, 2015.

[22] Y. Hu, Y. Koren, and C. Volinsky, “Collaborative filtering for implicit feedback datasets,” in *2008 Eighth IEEE International Conference on Data Mining*, pp. 263–272, December 2008.

[23] G. Adomavicius and A. Tuzhilin, “Context-Aware Recommender Systems,” in *Recommender Systems Handbook* (F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, eds.), vol. 54, pp. 191–226, Boston, MA: Springer US, 2015.

[24] C. C. Aggarwal, *Time- and Location-Sensitive Recommender Systems*, pp. 283–308. Cham: Springer International Publishing, 2016.

[25] A. Piegat and M. Pietrzykowski, *Local Modeling with Local Dimensionality Reduction: Learning Method of Mini-Models*, pp. 375–383. Cham: Springer International Publishing, 2016.

[26] O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, and K. Taha, “Efficient machine learning for big data: A review,” *Big Data Research*, vol. 2, no. 3, pp. 87 – 93, 2015.

[27] N. Gilardi and S. Bengio, “Local machine learning models for spatial data analysis,” *Journal of Geographic Information and Decision Analysis*, vol. 4, no. 1, pp. 11–28, 2000.

[28] H. Zhang, A. C. Berg, M. Maire, and J. Malik, “Svm-knn: Discriminative nearest neighbor classification for visual category recognition,” in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’06)*, vol. 2, pp. 2126–2136, IEEE, 2006.

[29] K. Lau and Q. Wu, “Local prediction of non-linear time series using support vector regression,” *Pattern Recognition*, vol. 41, no. 5, pp. 1539–1547, 2008.

[30] K. Grolinger, M. Hayes, W. A. Higashino, A. L’Heureux, D. S. Allison, and M. A. Capretz, “Challenges for mapreduce in big data,” in *2014 IEEE World Congress on Services*, pp. 182–189, IEEE, 2014.

[31] J.-X. Dong, A. Krzyzak, and C. Y. Suen, “Fast svm training algorithm with decomposition on very large data sets,” *IEEE Transactions On Pattern Analysis and Machine Intelligence*, vol. 27, no. 4, pp. 603–618, 2005.

[32] M. Kawulok and J. Nalepa, “Dynamically adaptive genetic algorithm to select training data for svms,” in *Ibero-American Conference on Artificial Intelligence*, pp. 242–254, Springer, 2014.

[33] L. Guo and S. Boukir, “Fast data selection for svm training using ensemble margin,” *Pattern Recognition Letters*, vol. 51, pp. 112–119, 2015.

[34] H. Brighton and C. Mellish, "Advances in instance selection for instance-based learning algorithms," *Data Mining and Knowledge Discovery*, vol. 6, no. 2, pp. 153–172, 2002.

[35] H. G. Jung and G. Kim, "Support vector number reduction: Survey and experimental evaluations," *IEEE Transactions on Intelligent Transportation Systems*, vol. 15, no. 2, pp. 463–476, 2014.

[36] H. Liu and H. Motoda, "On issues of instance selection," *Data Mining and Knowledge Discovery*, vol. 6, no. 2, pp. 115–130, 2002.

[37] E. Leyva, A. Gonzlez, and R. Prez, "Three new instance selection methods based on local sets: A comparative study with several approaches from a bi-objective perspective," *Pattern Recognition*, vol. 48, no. 4, pp. 1523 – 1537, 2015.

[38] S. García, J. Luengo, and F. Herrera, *Instance Selection*, pp. 195–243. Cham: Springer International Publishing, 2015.

[39] M.-A. Carbonneau, E. Granger, A. J. Raymond, and G. Gagnon, "Robust multiple-instance learning ensembles using random subspace instance selection," *Pattern Recognition*, vol. 58, pp. 83 – 99, 2016.

[40] Z.-Y. Chen, C.-F. Tsai, W. Eberle, W.-C. Lin, and S.-W. Ke, "Instance selection by genetic-based biological algorithm," *Soft Computing*, vol. 19, no. 5, pp. 1269–1282, 2015.

[41] D. A. Silva, L. C. Souza, and G. H. Motta, "An instance selection method for large datasets based on markov geometric diffusion," *Data & Knowledge Engineering*, vol. 101, pp. 24–41, 2016.

[42] C. Domeniconi, D. Gunopulos, S. Ma, B. Yan, M. Al-Razgan, and D. Papadopoulos, "Locally adaptive metrics for clustering high dimensional data," *Data Mining and Knowledge Discovery*, vol. 14, no. 1, pp. 63–97, 2007.

[43] M. Wu and B. Schölkopf, "A local learning approach for clustering," in *Advances in neural information processing systems*, pp. 1529–1536, 2006.

[44] R. Chitta, A. K. Jain, and R. Jin, "Sparse kernel clustering of massive high-dimensional data sets with large number of clusters," in *Proceedings of the 8th Workshop on Ph.D. Workshop in Information and Knowledge Management*, PIKM '15, pp. 11–18, ACM, 2015.

[45] N. Zhou, Y. Xu, H. Cheng, J. Fang, and W. Pedrycz, "Global and local structure preserving sparse subspace learning: An iterative approach to unsupervised feature selection," *Pattern Recognition*, vol. 53, pp. 87 – 101, 2016.

[46] G. Jawaheer, P. Weller, and P. Kostkova, "Modeling User Preferences in Recommender Systems: A Classification Framework for Explicit and Implicit User Feedback," *ACM Transactions on Interactive Intelligent Systems*, vol. 4, no. 2, pp. 8:1–8:26, 2014.

[47] S. Akuma, R. Iqbal, C. Jayne, and F. Doctor, "Comparative analysis of relevance feedback methods based on two user studies," *Computers in Human Behavior*, vol. 60, pp. 138 – 146, 2016.

[48] L. O. Colombo-Mendoza, R. Valencia-García, A. Rodríguez-González, G. Alor-Hernández, and J. J. Samper-Zapater, "RecomMetz: A context-aware knowledge-based mobile recommender system for movie showtimes," *Expert Systems with Applications*, vol. 42, pp. 1202–1222, February 2015.

[49] S. Wang, B. Zou, C. Li, K. Zhao, Q. Liu, and H. Chen, "CROWN: A Context-aware RecOmmender for Web News," in *2015 IEEE 31st International Conference on Data Engineering*, vol. 2015-May, pp. 1420–1423, IEEE, April 2015.

[50] M. Hosseinzadeh Aghdam, N. Hariri, B. Mobasher, and R. Burke, "Adapting recommendations to contextual changes using hierarchical hidden markov models," in *Proceedings of the 9th ACM Conference on Recommender Systems*, RecSys '15, pp. 241–244, ACM, 2015.

[51] T. Hussein, T. Linder, W. Gaulke, and J. Ziegler, "Hybreed: A software framework for developing context-aware hybrid recommender systems," *User Modeling and User-Adapted Interaction*, vol. 24, pp. 121–174, February 2014.

[52] X. Liu and W. Wu, "Learning context-aware latent representations for context-aware collaborative filtering," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, pp. 887–890, ACM, 2015.

[53] G. Adomavicius, R. Sankaranarayanan, S. Sen, and A. Tuzhilin, "Incorporating contextual information in recommender systems using a multidimensional approach," *ACM Transactions on Information Systems*, vol. 23, pp. 103–145, January 2005.

[54] P. G. Campos, I. Cantador, and F. Díez, "Exploiting time contexts in collaborative filtering: An item splitting approach," in *Proceedings of the 3rd Workshop on Context-awareness in Retrieval and Recommendation*, CaRR '13, pp. 3–6, ACM, 2013.

[55] W. Yao, J. He, G. Huang, J. Cao, and Y. Zhang, "A graph-based model for context-aware recommendation using implicit feedback data," *World Wide Web*, vol. 18, pp. 1351–1371, September 2015.

[56] T. Pessemier, S. Dooms, and L. Martens, "Context-aware recommendations through context and activity recognition in a mobile environment," *Multimedia Tools Appl.*, vol. 72, pp. 2925–2948, October 2014.

[57] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, pp. 30–37, August 2009.

Curriculum Vitae

Name:	Dennis Eduardo Bachmann
Year of Birth:	1986
Post-Secondary Education and Degrees	The University of Western Ontario London, ON 2015 - 2016 MESc
	Positivo University Curitiba, PR - Brazil 2009 - 2012 Bsc
Related Work Experience:	Teaching Assistant The University of Western Ontario 2015 - 2016
	Co-founder and Director of Technology BMS Solutions (Brazil) 2012 - 2014
	Intern AmBev (Brazil) 2011 - 2012
	Co-founder and Lead Programmer Bachmann Security (Brazil) 2008 - 2011
	Intern IBM (Brazil) 2007