MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering

Major: Information and Communication Technologies

Products Recommender System improved with social network information

Master thesis

Simone Cogno
simone.cogno@master.hes-so.ch

Professors

Ghorbel Hatem, *HE-ARC Data analytics group*
Hatem.Ghorbel@he-arc.ch

Punceva Magdalena, *HE-ARC Data analytics group*
magdalena.punceva@he-arc.ch

Expert

David Jacot, *Swisscom SA*
david.jacot@gmail.com

Client and location

Saint-Imier, HE-ARC Data analytics group

HES-SO//Master, February 4, 2016, Version 2.2
Av. de Provence 6, CH-1007 Lausanne

Accepted by HES-SO//Master (Switzerland, Lausanne) on a proposal from

Ghorbel Hatem, *HE-ARC Data analytics group*
Hatem.Ghorbel@he-arc.ch

Advisor
Schroeter Nicolas,
Nicolas.Schroeter@hefr.ch

Head of MSE
Moghaddam Fariba,
fariba.moghaddam@hes-so.ch

Abstract

Recommender systems are obtaining a lot of importance in e-commerce and a variety of other world wide web applications. Some of the most common are probably movies, books, music, news, documents, and products in general.

This thesis is focused on the creation of a Products Recommender Systems (RS). We provide a state of the art of the main recommendation techniques such as Content-based (CB) or Collaborative filtering (CF). Besides, we have explored some new techniques which can take advantages of social network information to enhance the recommendation quality.

We have decided to implement a Hybrid Recommender System using the Featured-Weighted User Model (FWUM) approach described in [1] in addition to the Trust Walkers algorithm described in [2]. Afterwards, we have designed the architecture of the system, and we have provided an overview of several Big Data technologies such as Hadoop, Spark, and Cassandra including some of their advantages and drawbacks related to our project.

In the final part, we provide a description of the main steps for the implementation of the two algorithms. Then, we spoke about several optimizations adopted and the solution to the problems we had met. To test our algorithm we collected a books dataset from Goodreads.com by using their public API [1] and a dataset of products from Epinions.com. The two datasets contain a list of items, a list of users and their ratings. They also provide social relations between the users such as trust or friendship.

The tests were performed using some of the most common measures for evaluating the quality of a Recommender System such as Precision and Recall.

The results of the Hybrid RS showed that for the 45% of users the recommendation has a precision between 10% and 60% by using the Goodreads dataset. We have observed, instead, more limited results when we have used the Epinions.com dataset.

[1] Goodreads API, 2015, `https://www.goodreads.com/api/`

[9] Epinions.com dataset, `http://liris.cnrs.fr/red/`

Acknowledgments

I would like to thank and express my gratitude to the following people who have been fundamental in the successful completion of this project.

Firstly, I would like to thank Mr. Ghorbel Hatem for his monitoring during the whole project and for his many advices that helped me to work in the proper direction.

Thanks to Ms. Magdalena Punceva for always been available to meet me and for answering my many questions. Thanks also for having helped me to find interesting articles and supporting me to understand them.

Mr. David Jacot, of Swisscom, for their useful observations that allowed me to improve the design and the quality of my work.

I would also say thanks to all the mentioned people for their availability throughout all the project.

A special thank also goes to my girlfriend, Sara Radaelli, for encouraging me to choose this challenging project as well as for always supporting me.

Finally, I would like to thank all my colleagues of room C10.08 for the good times and the exchange of knowledge we had during the whole project.

Contents

Abstract **i**

1 Introduction **1**
1.0.1 Objectives . . . 1
1.0.2 Our Approach . . . 2

2 Analysis **4**
2.1 Related work . . . 4
2.1.1 Content-based RS . . . 4
2.1.2 Collaborative Filtering . . . 6
2.1.3 Hybrid RS . . . 7
2.1.4 Social Recommender System (SRS) . . . 8
2.1.5 Sentiment analysis . . . 10
2.1.6 Evaluating a Recommender System . . . 10
2.2 Dataset selection . . . 12
2.3 Distributed databases . . . 13
2.4 Data analysis frameworks . . . 14
2.5 Conclusion . . . 15

3 Design **17**
3.1 Architecture . . . 17
3.1.1 Goodreads dataset . . . 18
3.1.2 Epinions dataset . . . 22
3.2 FWUM algorithm . . . 22
3.2.1 Construction of the content-based user profile step . . . 22
3.2.2 Feature weighting step . . . 23
3.2.3 The user's neighborhood formation step . . . 24
3.2.4 Top-N list generation step . . . 24
3.2.5 Parameters . . . 25
3.3 Trust Walkers algorithm . . . 25
3.4 Combining the FWUM and the Trust Walkers (TW) . . . 27
3.5 Conclusion . . . 27

4 Implementation **29**
4.1 Cassandra configuration . . . 29
4.2 Spark cluster set-up . . . 30
4.2.1 Monitor the running algorithm . . . 31
4.3 Goodreads Python API . . . 32

4.3.1 Problems encountered . . . 33
4.4 FWUM algorithm . . . 34
4.4.1 Construction of the content-based user profile step . . . 34
4.4.2 Feature weighting step . . . 35
4.4.3 The user's neighborhood formation step . . . 35
4.4.4 Top-N list generation step . . . 36
4.4.5 Metrics . . . 37
4.4.6 Optimization . . . 38
4.4.6.1 Matrix multiplication . . . 38
4.4.6.2 Database size reduction . . . 39
4.4.7 Problems encountered . . . 39
4.4.7.1 Waste of time when a node crash . . . 39
4.4.7.2 Out of memory errors . . . 39
4.4.7.3 Inclusion of the dependencies . . . 40
4.5 Trust Walkers algorithm . . . 40
4.5.1 Problems encountered . . . 41
4.5.1.1 Items similarity colculation . . . 41
4.6 Web UI . . . 42
4.7 Conclusion . . . 42

5 Tests **45**
5.1 Metrics and visualization of the results . . . 45
5.2 Parameters optimization . . . 46
5.3 Scalability test . . . 48
5.4 Test with Goodreads dataset . . . 49
5.4.1 Test description . . . 49
5.4.2 Obtained results . . . 49
5.4.3 Discussion of the results . . . 52
5.5 Test on Epinions dataset . . . 53
5.5.1 Test description . . . 53
5.5.2 Obtained results . . . 53
5.5.3 Discussion of the results . . . 54
5.6 Conclusion . . . 54

6 Conclusion **57**
6.0.1 Issues and difficulties . . . 59
6.0.2 Future enanchements . . . 60
6.0.3 Personal conclusion . . . 60

7 Declaration of honor **63**

Acronyms **65**

Bibliography **67**

List of Figures **69**

List of Tables **70**

A CD-ROM Content **71**

B Recommanded reading **73**

C Source code repository **75**

D Planning **77**

E Code samples **79**
E.1 Word count with Hadoop 79

F Installation guide **83**
F.1 Goodreads Python API 83
F.2 Install and configure Cassandra 83
F.3 Install and configure Spark 84
F.3.1 Spark cluster set-up 84
F.3.2 Launch a Spark EC2 cluster 85
F.3.3 Connect to the Master driver 85
F.3.4 Build and include the dependencies 85
F.3.5 Setting up the Master 86
F.3.6 Run a Spark application 86

G Project specification **87**

Introduction

Recommender systems are obtaining a lot of importance in e-commerce and a variety of other world wide web applications. Some of the most common are probably movies, books, music, news, documents, and products in general.

During the last years, we also observe an huge increment of application that uses the social network to improve the user experience and some of them are also publicly available.

We can mention the famous Twitter website that allows to retrieving a lot of social network data through their public API [1]. There are also several datasets available online which provides these data for research propose, for example, the Epinions.com dataset[2].

The most common techniques used to create a Recommender System (RS) are the Content-Based (CB) and Collaborative Filtering (CF). CF approach builds a model from the past rating history of users as well as other similar decisions made by other users. This model is used to retrieve a list of items that the users may be interested in. A Content-based RS use the past user reviews to describe a user profile using several discrete characteristics. The items having similar properties are then recommended to the users. The CB and CF can also be combined in a Hybrid approach or improved with social informations.

The aim of this project is to analyze several techniques used for creating a RS including some new approaches that take advantages of the social network information to enhance the recommendation quality.

After this research, an implementation of the most interesting algorithm will be created and tested for demonstrating the correctness and the performance of the system.

1.0.1 Objectives

The primary objective of this thesis is to create a Products Recommender System. A state of the art of the most common technique will be established as well as a research of the public datasets available. An algorithm has to be chosen for implementing a prototype of a Recommender Systems using the dataset founded. Ideally, the algorithm has to be implemented with Hadoop or Spark in order to allow us analyze a big dataset by taking advantage of the computational power of several machines in parallel. The algorithm implemented has to be tested with the several standard metrics for providing an estimation of the quality of the Recommender System.

As secondary objectives, we also wish to create a stand-alone web application that show the recommendation made to the users. Furthermore, we wish to adapt the algorithm for the RecSys challenge 2016[3].

[1] Twitter REST API, `https://dev.twitter.com/rest/public`
[2] Epinions.com dataset, `http://liris.cnrs.fr/red/`
[3] RecSys Challenge 2016, `http://recsys.acm.org`

1.0.2 Our Approach

In the first part of the project, we have analyzed several techniques used to creating a Recommender System. Including some of the commons techniques as well as some new approaches that take advantages of social network information (Chapter 2.1). In our approach we have chosen to implement a Hybrid Recommender Systems (HRS) using the method described in [1].

An analysis of the publicly available datasets has been made for retrieve a dataset with enough information about the social connection between the users (Chapter 2.2). In particular we have collected a book dataset from Goodreads.com by using their public API [4] and from Epinions.com[5]. The two datasets contain a list of products (or books), a list of users and their ratings. They also provide social relations between the users such as trust or friendship.

To implement our algorithm we have used Spark as the analysis framework and we have used Cassandra as a persistent database. In addition, we use AWS S3 for read and store HDFS file where Cassandra was not needed.

The tests were performed using an EC2 Spark cluster. To measure the performance of our RS we have used some of the most common metrics for evaluating the quality of a Recommender System such as Precision and Recall.

[4]Goodreads API, 2015, `https://www.goodreads.com/api/`
[5]Epinions.com dataset, `http://liris.cnrs.fr/red/`

Analysis

2.1 Related work

2.1.1 Content-based RS

Content-based RS take advantages of the past ratings of users to build an user profile using several discrete characteristics. This model is then used to recommend new items having similar properties. To performing such a recommendation we have to computer *Item* and the *User profile* and compare their properties to retrieve items that are relevant to the user based on his taste.

In the approaches presented from [3, Jeffrey D. Ullman] uses the properties of items to construct a vector. This method is called "Vector model representation" and it is used to construct the *item* and the *user* profile. For example, we can build the components of the item vector using their category, the frequency of words ()used in the document in [4]) or other useful information.

A book can have a vector component for their different genre. As an example, we can take "adventure", "romance", "comedy" as the set of possible classes of books. The vector of a romance and a comedy book will be {0,1,0} and {0,0,1} respectively. We can clearly describe the items with as many components we need, as long as all the vectors have the same number of components.

To compare two items, we can use several similarity measures. One of the most used is the *cosine similarity* but for simple use cases the *Jaccard similarity* can be used as well.

The cosine similarity is described in the equation 2.1. As we can see, we compute the product of two item vector A and B divided by their module. This calculation gives the result of the angle between the two vector. If the angle between the two items is small, they are very similar. Otherwise, they differ a lot. The cosine similarity gives values between 0 and 1.

$$sim = cos(\Theta) = \frac{A \bullet B}{\|A\| \bullet \|B\|} \tag{2.1}$$

We have seen how to create the Item profile and how to calculate the similarity between two items. Now we have to construct the user profile to compare it to items. The method presented in [3, Jeffrey D. Ullman] explain that we can build the user profile according to the previous ratings given by the user on the items. The value can be a boolean for describing if the user has bought an item or not, otherwise can be an integer number representing the stars that the user gave as a rating for an item.

With these numbers, we can construct the utility matrix that represents the user ratings on a set of items. The blanks entry describe the situation where the user has not rated the item.

From the utility matrix, we can retrieve the list of items that a user has bought or to that he has given them a rating. Then we can make an average of the components in the item profile for constructing the vector representing the user profile.

We can say that 20% of items bought by the user are in the category "romance". In that situation, the user profile vector will have 0.2 as the value of the "romance" component.

When we have computed the user profile vector, we can now compare the user to the items since they have the same component types. The cosine similarity can be a good choice for measuring this similarity.

By performing this technique, we can easily compute the similarity between the characteristics of a user compared to particular items in the system. Doing so, we can finally recommend some items to the user which have not been rated by him yet.

The advantage of this approach is that is quite simple to implement and, if there are enough information on the dataset, the quality of the recommendation is pretty good.

The disadvantage of this approach is the complexity of the computation when we want to discover the items similar to the interest of a user. In fact to make this calculation we have to traverse all the items in the dataset and to choose the most similar. If the dataset is large, this can be an expensive calculation. For avoiding this limitation some solution have been purposed on [3, Jeffrey D. Ullman]. One approach is to cluster similar items to have far fewer items to compare. The clusters can be recalculated systematically with a specified interval of time and not on every recommendation to a user. In this case, we make the assumption that the dataset of user ratings varies slowly. Another limit of this algorithms is that if the user hasn't rated many items (cold start) the system cannot calculate the user interest precisely, therefore the recommendation can be of poor quality. Collaborative filtering can improve the cold start problem by focusing on the similarity between the users. Thanks to this approach, the system can recommend items to a user according to the ratings made by similar users (see section 2.1.2). In conclusion, this method is very useful if we want to have a RS that works well and does not require a complex algorithm. Also, it can be a good starting point to combine it with other approaches. To resume the pros and cons of the Content-Based Recommender Systems we can see the table 2.1.

Table 2.1: Pros and cons of the Content-Based RS [1]

Pros	***Cons***
It do not need data of other users	Retrieve the appropriate features is difficult if we have few data
It's able to recommend new and unpopular items	Cold start problem for the new users (user profile)
It's possible to provide explanations about the recommendation (features)	Does not recommend items outside the user profile and the judgments of others users are not taken into account (overspacialization)

[1] Mining of Massive Datasets, RS presentation, `http://www.mmds.org/mmds/v2.1/ch09-recsys1.pptx`

2.1.2 Collaborative Filtering

The content-based recommendation systems use the information about the interests of the user for recommending items that he may like. By using CF we don't only consider the ratings of a user but also those of similar users in the systems. In this way, we can recommend items that are very different to the preference of the users, but can be interesting since similar users have rated it positively.

To measure the similarity between two users, in [3, Jeffrey D. Ullman p.320] they use the cosine similarity from the user-rating matrix, also called *utility matrix*. Two users are similar if they rated a lot of items in common, with a similar rating.

From the list of similar users, we can retrieve the items they like the most. Then we can recommend these items to the final user.

In [3], they also make a normalization of user rating for better represent the behavior of the user. A 5-stars rating from a user that have rated items only with 4 or 5 stars is different from the same rating from another user that have rated items only with 1-2 stars. To take into account the behavior of a user we normalize the value by subtracting the average rating of the user from the value in the utility matrix. In this way, we have a positive and negative value representing low rating and high rating respectively.

An improvement of this method is to cluster similar items and users before executing the recommendation process. This method can avoid the problem of the sparse utility matrix due to the few ratings that users make on items.

In general, the collaborative-filtering approach is very effective because it helps the people to discover new items that are not similar to their usual interests. In the other hand, this approach uses only the user ratings and does not take into account other information about the content of the items. The utility matrix can be very sparse, especially at the beginning, and the prediction can be imprecise. A hybrid approach can be useful for filling this lack by using both CB, CF and other methods.

The paper [5, J. O'donovan and B. Smyth] explores another technique to refine the CF technique. Basically, they try to give a measure of "Trust" between users. Let's consider a user v similar to u that is taken into account for the prediction of a item i. The measure of trust between user u and v is calculated by the percentage of correct prediction that user v has made for predicting some items at user u. User v makes a correct prediction only if his rating on item i is similar to the average (or another aggregate function) of the ratings made by the group of users similar to u.

For the next prediction the contributions of user v are weighted according to the confidence that the user u has on him. For example, if the user v make 20 correct prediction for user u over 100 total contribution, the value of confidence between u and v is 0.2.

Here the measure of confidence is calculated from implicit information. A more reliable measure of confidence can be acquired by explicit ratings that users give to other users. As an example, in the Amazon[2] website, we can rate a seller from 0 to 5 stars, for denote how much trust we have on him. Using implicit information, however, can enhance the recommendation quality and can be useful for filtering some "fake" users that have made

[2]Amazon,www.amazon.com

ratings in an inconsistent way. In conclusion, using it with the regular CF approach, can improve the recommendation quality.

In general, the advantage and disadvantage of performing a Collaborative Filtering are showed in the table 2.2. The main difficulty is to have enough data of the users ratings for avoiding the Cold start problem.

Table 2.2: Pros and cons of the common Collaborative-filtering approach [1]

Pros/ Cons
+ It do not need to items feature selection
+ It can take into account the judgments of other users and recommend items outside the user profile
- Need enough users to find similar users (Cold start)
- Hard to find users that have rated the same item (Sparsity)-
- Not possible to predict items that are not been previously rated (First rater)
- Tends to recommend popular items and omit the new ones

2.1.3 Hybrid RS

In the Hybrid RS, the previous described CB and CF are combined to improve the recommendation quality. Often a hybrid system uses the results of CF and run CB on it. In [1, Panagiotis Symeonidis et al.] they make use some notion of Information Retrieval for including the Term Frequency Inverse Document Frequency (TFIDF) as a measure for similarity and weighting scheme in CF. This technique is used for weighting the words of documents according to their rarity. In general, the rarity of a word in the entire collection of documents and their frequency in a singular document denote to their significance to compare two paper. In fact, two documents that have both some rare words have a good chance to talks about the same subject. Inversely, two papers that have some very popular words in common are not necessarily speaking of the same topic.

This principle can also be translated for recommender systems. If, for example, two users have liked a book of the category "Fantasy", doesn't mean that they are similar. This is because the feature "Fantasy" is very popular among users and most of them have rated it (Ex. most of the peoples like Harry Potter or the Lord of the Ring). On the other hand, if two users have both liked books of a very rare category, let's say "Egyptian literature", they have more chance to be similar because only a few users are interested in this category. It is, therefore, reasonable to weight the item features according to their rarity in the whole dataset.

In [1, Panagiotis Symeonidis et al.] the *TFIDF* is converted into Feature Frequency and Inverse User Frequency (IUF) for a resulting *FFIUF* measure. Where FF(u, f) is the number of times a feature f occurs in the profile of user u. Moreover, the UF(f) is the number of users that have the feature f in their profile at least once. The IUF is the inverse

[1] Mining of Massive Datasets, RS presentation, `http://www.mmds.org/mmds/v2.1/ch09-recsys1.pptx`

of this value and can be calculated with the equation 2.2.

$$IUF(f) = \log(\frac{\mid U \mid}{UF(f)}) \tag{2.2}$$

Here the $\mid U \mid$ value denotes the total numbers of users in the dataset. Finally, the equation 2.3 calculates the weight W of feature f for user u. The algorithms that use this measure for weighting the features is named FWUM.

$$W(u, f) = FF(u, f)IUF(f) \tag{2.3}$$

From the evaluations made on [1, Panagiotis Symeonidis et al.] we can see that FWUM performs better that CF and CB approach, also for a small dataset.

This method is very interesting due to its implementation of both CB and CF techniques with some improvements on them. Moreover, it adds some complexity on the construction of the algorithm but it recovers this complexity in term of quality of the prediction

It is important to note that this algorithm does not take into account any social information. By integrating social information it is possible to improve the algorithm for example to the cold-start problem.

For a complete explanation of this algorithm, we have described it in the next chapter at the section 3.2.

In the next section, we will resume some articles in the context of SRS and we will explore some of the principles used for this type of recommendation.

2.1.4 SRS

In this section, we describe some techniques for SRS that we have explored in several articles. The main goals for a SRS is to take advantage of the social network information for predict interesting items to the users.

In [6, W. Carrer-Neto M.L. Hernández-Alcaraz R. Valencia-Garc ıa F. Garc ıa- Sánchez] they make the assumption that user friends have similar interests and similar behaviors. They then recommend the items rated by the nearest friends. This assumption has obviously some limits if we compare it to real data as some friends can have different interests. Then they enhance the amount of data by combining the friend network with other similar users in the whole system with a knowledge-based technique. In the final system, they also provide that the user can weight the importance of the recommendation based on the network of friends and the recommendation based on the knowledge-based approach. An open-minded user can give more relevance to the friends recommendation. Otherwise, he can decrease their weight.

This technique can be combined with CB and CF approaches and has the advantage to be simple. The disadvantage is that can be imprecise when compared with real data. The user's friends can be very different to the user himself. For example on Facebook, a user can have hundreds or even thousands of friends and is oblivious that not all of them have similar tastes.

In [7][8] they take into account the differences between user and his friends. This is done by weighting the contribution of a user in the recommendation process according to the similarity between the user and his friends. The similarity measure is based on the previous ratings made by users. By taking into account the contribution of the network of friends it is possible to enhance the recommendation quality, while maintaining the simplicity of the algorithm. In fact, this method has a similar computational complexity of CF algorithms. Besides, compared to a CF approach, this can be useful for cold start problem when we have few data (ex. ratings) for a new user. By considering his friends, we can certainly increase the amount of data available.

In [8, Alexandra Olteanu et al.] they explore some social affinities that can be useful for the recommendation purpose. They estimate the social affinity in a friends graph by using Random Walks (RWs). This technique has been used for both friend recommendations (paper [9], [10]) and item recommendation (paper [11],[2],[12]). In short, for predicting item i for user u they perform several RWs starting from u until they reach an user v that have rated item i or when performing a maximum number of steps k-max. In [2] they also take into account when user v has rated an item similar to i. The similarity between items is calculated with the Pearson Correlation Coefficient (PCC) on the user ratings, but also the Vector Space Similarity (VSS) can be used as well. They call this algorithm *TrustWalker*. As presented in [2] TrustWalker has a good improvement compared to other method like CB or CF, especially in the cold-start situation. The downside is the complexity of the computation for a large graph, but we can limit the steps made by RWs by set k-max=6, according to the "six-degrees of separation" rule [13], without loosing important information.

A different approach has been presented in [14]. They partition the user-items rating matrix according to contextual information associated with each rating (data, hour, physical position, etc.). By doing this, they increase the prediction accuracy of relevant items. They also use an additional social regularization term that is calculated based on the similarity between users. The similarity measure that they choose is the PCC, computed on the previous rating made by users combined with contextual information that are associated with each rating. The disadvantage of this method is that our dataset does not have a lot of information on the context in which ratings have been made.

Until now we have explored algorithm based on explicit information on the social network as the network of friends. In [15] they explore some techniques for calculating the social affinity between two users based on the activity that they perform in the system. With these information we can build a *implicit* social network based on the tie strength. In this approach, they use a bipartite graph for calculating these measures. This graph contains two type of node and links between them. For example, *users* and *events*. Where an *event* can be any activity that two or more users made together. They define some axioms that the measure of strength must follow, and they expose some functions that satisfy the most of them. Some of the more interesting measures are the *Common Neighbors*, *Katz Measure* and *Random Walk with Restarts*. But several other functions can be used as well.

2.1.5 Sentiment analysis

In the previous sections, we made the assumption that the evaluations made by the users are effectively their real thought about an item. In reality, the opinion of users can be quite different for the value of the rating. For discovering their real opinion, we can take advantage of the reviews text. The process that measures the feeling of a user starting from a text or a sentence is called *Sentiment Analysis* and refer to the use of *Natural Language Processing*.

Four our particular use case this technique can be useful for improving the reliability of the rating on items. In fact, the mean of all the rating can be a very significant value but sometimes don't reflect the real evaluation of the items. We can measure the opinion of a user on an item by analyzing the terms and the sentences wrote in the reviews. The resulting user opinion to an item can be very useful for recommending quality items.

It is important to note that the branch of Natural Language Processing is very vast, and it is still under development and research. Building an algorithm for performing Sentiment Analysis on text can be a very expensive task, and it is not the purpose of this work. In our project, we want to retrieve that information only for improve the quality of our dataset and for testing the usefulness of using it on a RS. Therefore, we will use some tools (ex *textblob*[3].) that can helps us to retrieve these information.

2.1.6 Evaluating a Recommender System

Evaluating a RS is a necessary task that has to be accomplished for testing the correct behavior of an algorithm.

One of the criteria for evaluating a RS is to calculate the prediction quality by using one or multiple of these metrics[16, J. Bobadilla et al. cp. 4]:

- Mean Absolute Error (MAE)
- Root Mean Square (RMSE)
- Normalized Mean Average Error (NMAE)

These measures are calculated by measuring the error between the real data and the predicted ones.

For evaluate recommendation quality we have some commonly used metrics.

- Precision
- Recall
- Receiver Operative Characteristics (ROC)
- F1 score

Most of these measures can be retrieved starting from the Confusion Matrix that it is determined in the recommendation process. This matrix provides some value about True

[3]TextBlob, https://textblob.readthedocs.org/en/dev/quickstart.html#textblobs-are-like-python-strings

Positive, True Negative, False Positive and False Negative prediction. With these measures, we can retrieve all the metrics listed above.

Other useful metrics for evaluating a RS can be the consideration of the Diversity and Novelty [16, J. Bobadilla et al. cp. 4]. Where the diversity is the measure of the commonly recommended items presented to the user and Novelty is the measure of the promotion of less widely known (so-called long tail) items in the whole dataset [17].

For measuring all these metric, the it is recommended to use the k-fold cross validation because of its advantages on low size dataset and for limiting the over-fitting problems [16, J. Bobadilla et al. cp. 4].

2.2 Dataset selection

For this project, we need a dataset that provides enough information about the users and their activities. In particular, we need information about what the users like and their interests. Additionally, we are also interested to retrieve a dataset that provides some social relations between the users. We analyze several datasets founded online and we create the table 2.3 that summarize the key points for our project. The column "*User information*" tell about what information are present in the user profile such as Name, Age, and other details. This can be useful to describe a user and to compare two different users. The column "*Items information*" tell us about what information we have on every item in the dataset. These information can be text, votes, categories of the items, etc. The column "*Reviews information*" describe the information that links an user to an item such as the history of the purchased items, their reviews, their comments, etc.

Dataset	Type of datas	User information	Items informations	Review information	Social information
Amazon[1] *product*	*Product review*	*Name, age, born date*	*Type*	*Buy history and reviews*	*No*
MovieLens[2]	*Movie Recommendation*	*user id, age, gender, occupation, zip code*	*Publishing date, 19 genres*	*Movie ratings*	*No*
Million song[3]	*Song dataset*	*Name, age, born date*	*Upvotes, author, genre, lyrics, tag*	*Song play count*	*No*
Delicious Bookmarks [4]	*Boorkmarkt*	*Name, age, born date*	*Text, tag*	*Preferred bookmarks urls*	*Fun relations*
Book-Crossing[5]	*Books*	*Location, age,*	*Book-Title, Book-Author, Year-Of-Publication, Publisher*	*Book rating*	*No*
Last.fm [6]	*Songs*	*Gender, age, country, song-num plays*	*Artist name, track name*	*Hystory of played songs*	*Following relations*
Goodreads.com[7]	Books	Name, age, born date	category tags, votes, rating, text reviews	Books rating, reviews	User friends
Twitter (API) [8]	Tweets	Name	Text, Nb. of favorite, retweets	Tweets hystory	User friends, followers
Epinion.com [9]	Product	Name, age	Categories	Users reviews	Trusted users

Table 2.3: Comparative table of datasets

As we can see from the table, we had difficulties to find a dataset containing some social information. Twitter offers a *Public API* [8] where we can retrieve the user's friends, their last tweets, the details of a tweet. Twitter impose rate limit of 15 calls every 15 minutes and, with every call, we can retrieve up to 3000 tweets from a user and up to 5000 friends per user. Another option is to connect to their *Streaming API* [10] and save the real-time data from Twitter.

With the *Goodreads HTTP API* [7] we have a rate limit of 1 call pro seconds and for every call, we can retrieve up to 100 friends of a user and the details of books. The reviews information have to be crawled from the reviews page of the site without any limit.

[1]Amazon product dataset, http://jmcauley.ucsd.edu/data/amazon/
[2]MovieLens, http://www.grouplens.org/node/73
[3]Million song, http://labrosa.ee.columbia.edu/millionsong/
[4]Delicious, http://grouplens.org/datasets/hetrec-2011/
[5]Book-Crossing, http://www2.informatik.uni-freiburg.de/~cziegler/BX/
[6]Last.fm dataset, http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-1K.html
[7]Goodreads API, 2015, https://www.goodreads.com/api/
[8]Twitter Public API, 2016, https://dev.twitter.com/rest/public/rate-limiting
[9]Epinions.com dataset, http://liris.cnrs.fr/red/
[10]Twitter Streaming API, 2016, https://dev.twitter.com/streaming/overview

The Epinion.com dataset has a dataset ready for use but it provides little information about an item. It provides only a general category of the items without any other details.

For the purpose of this project, we have chosen to use the Goodreads API because it provides a lot of useful information for a RS. In addition, we have used the Epinions.com [9] dataset to test our algorithm as well as to compare the recommendation performances.

2.3 Distributed databases

For the purpose of this project, we had to choose a database that could offer good reading performances and able to stock a large dataset. We also wanted that it could be extensible and distributed across several machines as we didn't know the exact size of the final dataset. Furthermore, as we have to build the dataset from zero, the databases has to be optimized for reading and writing.

Since there are a lot of options and the aim of this project was not to analyze every distributed database in details, we have made a short overview of some of them. We have considered the following distributed databases technologies:

- Amazon DynamoDB [4]
- Apache Hive[5]
- Apache Cassandra[6]
- Mongo DB[7]
- HBase[8]

The Amazon DinamoDB is a fully managed NoSQL database service that is high scalable. The advantage of using this database is that we do not have to worry about the management of the provisioning, the configuration, and the cluster scaling. The downside of this database is a proprietary service owned by Amazon.com and it has several cost related to the quantity of data and the number of read and write. In addition, we do not have any control on the database and, for an academic project like this, it is not really useful for learning.

Apache Hive is a data warehouse infrastructure built on Hadoop and optimized for analysis, queries and summarization of the data[9]. This is a good option, the only downside is that it is not meant to be used as a live database.

Mongo DB is another good options as it is a NoSQL document-oriented database that can be distributed on different nodes and provides hight performance, hight availability and automatic scaling [1]. Its advantages of using document are that every record corresponds to the real representation of the object in many programming languages. Compared the

[4]Amazon DynamoDB, 2016. http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

[5]Apache Hive, 2016. https://hive.apache.org

[6]Apache Cassandra, 2016. http://cassandra.apache.org

[7]Mongo DB, 2016. https://www.mongodb.org

[8]Hbase, 2016. https://hbase.apache.org

[9]Wikipedia Apache Hive, Measuring-Tie-Strength-in-Implicit-Social-Networks

DBMS systems a document-oriented database can avoid the need of expensive joins as the information needed can be embedded directly in a document or arrays. Another advantage is that the database schema is very dynamic and supports fluent polymorphism [1]

Cassandra and HBase are two databases modelled from the Google's BigTable[10]. Whereas Cassandra also descends from Amazon Dynamo, Hbase describes itself as an "open source BigTable implementation". So they have a lot of characteristics in common but also many differences.

Both of them are open source distributed database designed to handle a large amount of data. They can be distributed across many servers and provides high availability, fault tolerance, and high scalability.

A downside of HBase is that it requires enough hardware to run well because of its multiple management daemons processes such as Namenode, HBaseMaster, Zookeeper. etc[11]. It is recommended to run it on a minimum of 5 nodes. Cassandra, as far as we know, does not seems to have any requirements in this subject. HBase and Cassandra claims to have linear performance gains in read and write through adding nodes [12][13].

Was difficult to choose one of these distributed databases after this short analysis, because most of them are amply sufficient for the purpose of this project and because no test has been made to provides a real comparison between them.

We have finally chosen Apache Cassandra because it offers a lot of good features and documentation that helps to be deployed and configured in a reasonable amount of time. We have also discovered a good community on Stackoverflow.com. We also like some of its characteristics such as the flexible schema design that can be changed online with the possibility to store structured, semi-structured and unstructured data. In addition, we like the possibility to use the CQL language similar to SQL for easily manage the database, and we have found several Python and Spark drivers that provides Object mapping features. More information on how we have configured and used Cassandra can be found in Chapter 4.1.

2.4 Data analysis frameworks

For the data analysis, we had to choose between Hadoop or the more recent Spark library. We decided to use Spark because of it's several advantage compared to Hadoop. To mention some of them, with Spark we can compute an iterative algorithm and perform several map-reduce steps consecutively. In Hadoop, we can only make one map-reduce operation per jobs, and then we have to flush the entire data on disk and start another job from it. This is obviously a very expansive process compared to Spark when we can maintain the data in memory between several map-reduce operations. For some calculations it has been proven

[1] What is MongoDB, 2016, https://docs.mongodb.org/manual/core/introduction/

[10] Google's Big Table, 2015, https://en.wikipedia.org/wiki/Bigtable

[11] Hadoop Tutorial: HBase Part 1 – Overview, 2016, http://www.slideshare.net/martyhall/hadoop-tutorial-hbase-part-1-overview

[12] An Overview of Apache Cassandra, Datastax, 2016, http://www.slideshare.net/DataStax/an-overview-of-apache-cassandra

[13] Big data showdown: Cassandra vs. HBase, 2016, http://www.infoworld.com/article/2610656/database/big-data-showdown--cassandra-vs--hbase.html

that Spark can be 100x faster that Hadoop for example in the computation of a Linear regression[14].

Spark uses Resilient Distributed Dataset (RDD) as the principal data structure for manage data. An RDD can contain enormous data because Spark is able to distribute it across the machines of the cluster. An RDD supports two type of operations: *transformations* and *actions*. Some useful transformations are for example *join()*, *map()*, *groupBy()* that creates a new RDD with another internal structure.

We can then retrieve a result from an RDD by executing some actions on it, for example, we can count the number of rows contained in one RDD by executing the *count()* function.

In the following example we can see how a Word Count computation can be very easy using a Spark program:

```
val text_file = spark.textFile("hdfs://words.txt")
val counts = text_file.flatMap(lambda line: line.split(" "))
                      .map(lambda word: (word, 1))
                      .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("hdfs://word_count.txt")
```

As we can see, with Spark, we can write such a program in 5 lines (or less). To do the same program in Hadoop, we require at least two classes with around 250 lines of code (see annex E.1) that make it less readable and harder to manage.

Mainly for these reasons, we finally choose Spark for the analysis framework. The only "downside" is actual that, to take advantage of the best feature of Spark, we have to learn Scala. In fact, some of its libraries are only available in that language (ex. GraphX). This is not a downside, instead, if we are interesting to learn it. In fact, it is a very interesting and powerful language that is also more and more used in the market.

2.5 Conclusion

All the algorithms explored in the section 2.1 are useful for accomplishing the main goals of this work. However, for the final algorithm, we had to choose only a one or two of these algorithms. The most interesting ones.

The objective was also to build an algorithm that can be used with a various types of data. In our dataset, we have several kind of data such as structured data, social activity information and friends relations. The algorithm has to work by taking advantages of these data but also has to consider possible unstructured data that can be present in other datasets (ex. text).

For structured data we can use CB and CF or an Hybrid approach. As discussed in section 2.1.1 and 2.1.2, CF and CB take into account only user ratings data and uniformly weighted features respectively. The Hybrid approach can combine this two information in addition to weighting the item features thanks to a principle of Information Retrieval. As resulting from our analysis and the results obtained in [1], we can conclude that this

[14]Officla Spark website, Speed comparaison, 2016, `http://spark.apache.org`

algorithm is the most effective among the others. It can be used, furthermore, for structured and unstructured data (ex. text). Besides, to taking advantage of social network information, we can use some of the approaches discussed in section 2.1.4. As explained previously, [6, W. Carrer-Neto M.L. et al.] make too naive assumptions about the similarity of friends. The approach described in [14] use, instead, contextual information for enhancing the prediction in the recommendation process. This process can be interesting but, unfortunately, we do not have a lot of such information in our dataset. As resulting from the analysis, the remaining and most useful approach, is the one described in [7][8] based on the "TW" algorithm described in [2] where they weight the contribution of friends in the recommendation process based on the user similarity or with their social affinity. In fact, they enhance the reliability of the similarity measure between users using several tie-strength metrics described in [15].

To testing the algorithm, we choose the Goodreads and the Epinion dataset (section 2.2). We also choose Cassandra as distributed database after the overviews of the principals characteristics of the other ones such as MongoDB, HBase, Apache Hive and Amazon Dynamo (section 2.3). Since most of these databases provide more than enough features for our use case, the choice has been made by regarding three principals aspects: the facility to set-up, the tools and APIs available and the quality of the documentation. We found that Cassandra is easy to install and to set up and provides an excellent integration with Spark. We also found that their documentation is very clear, and have a great community on StackOverflow.com.

For the analysis framework, we made a comparison between Spark and Hadoop by regarding at some key points such as speed and ease of use. Our analysis shows (section 2.4) that Spark provide a better speed thanks to their optimized use of the memory [15]. In addition, it's more appropriate to implement a complex algorithm thanks to the RDD structure that provides lots of built-in optimized features.

To improve the reliability measure of a user interest about an item we can also use some Sentiment Analysis tools by analyzing the text reviews using the textblob tool[16].

In the next chapter, we will explain the details of the algorithm chosen as well as the architecture of the systems and the communication between all the components.

[15] Spark Official, `http://spark.apache.org`

[16] TextBlob, `https://textblob.readthedocs.org/en/dev/quickstart.html#textblobs-are-like-python-strings`

Design

In this chapter will be described the design phase of the project. In section 3.1 we will illustrate the architecture with several explanation about the functioning of the systems and the communication between each component. In section 3.1.1 we speak about the final schemes of the Goodreads and Epinions datasets that will be used in the implementation phase. In section 3.2 and 3.3 we will describe more in details the algorithm that has been chosen in the analysis part (chapter 2.5)

3.1 Architecture

The architecture of the system that will be set up is showed in figure 3.1. We can see three key blocks: (1) the Goodreads HTTP API and the Python API (2) the Cassandra cluster and (3) the Spark cluster.

The functioning of our system is pretty straightforward. Firstly we download the dataset from Goodreads.com into the Cassandra database. Spark will then read this database, containing the users ratings information and analyze it with several EC2 [1] on-demand instances in the cluster. The results of the computation are then re-saved into the Cassandra database or on Amazon S3[2] for further analysis on the results data.

The Python API has the task to download the dataset from Goodreads API, filter the useful data and save them in the Cassandra database. In addition, it is responsible to maintaining the consistency of the data in the database.

The Cassandra cluster is configured to optimize the most useful queries. Their schema has been optimized for this propose (more information on section 3.1.1)

The Spark cluster is constructed on demand using several m4.large EC2 instances. It uses the Spark Cassandra Connector [3] to read the data from Cassandra and convert it to a Spark RDD. To improve the performances, it can also read and save the data from S3 when Cassandra is not needed. In Spark, we use the Mllib and the GraphX libraries to take advantages of their built-in functions.

In the next sections, we describe the details and the explications on how the Goodreads and Epinions.com dataset were construed as well as the choice of the libraries and drivers.

[1] Amazon EC2,https://aws.amazon.com/fr/ec2/
[2] Amazon S3, https://aws.amazon.com/it/s3/
[3] Spark Cassandra Connector, https://github.com/datastax/spark-cassandra-connector

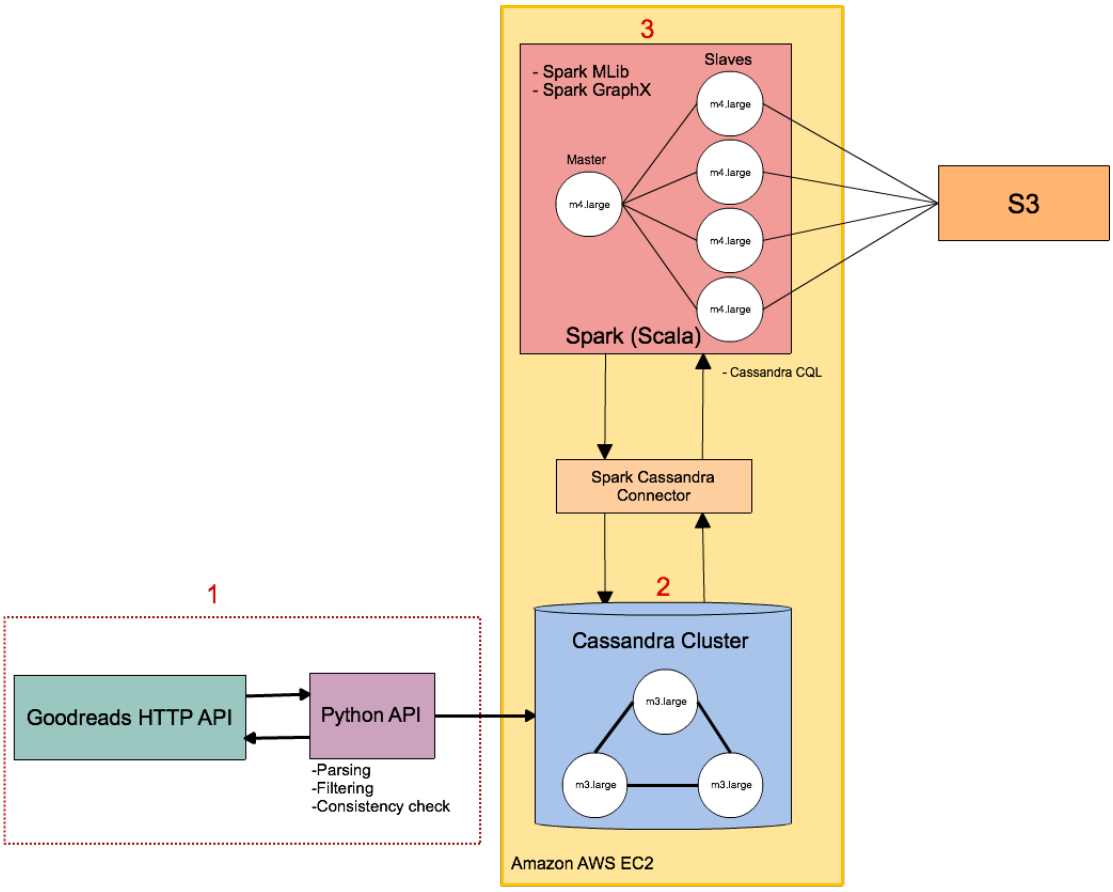


Figure 3.1: Architecture of the system

3.1.1 Goodreads dataset

In the analysis part on Section 2.2 we have chosen to retrieve the data from Goodreads.com[4].

Goodreads.com is a social network dedicated to books. In 2013, they had more than 25 million users and several millions of books. They provide a Web API to access to the resources of the website.

The Goodreads API is available by a web interface on the official site of Goodreads.com and eventually there are some API for Python that can be used for a primary usage. To downloading a quite large dataset we have to optimize the requests made to Goodreads.com, and it is necessary to create a custom API.

To doing this, it has been chosen to use Python as the main language because of the good integration with Cassandra and for the several wrappers available for the Goodreads API that we can take as a starting point.

In particular, we will use some code of "Sefakilic Goodreads api" [5].

The rest of the Python API will be implemented from scratch to having a better performance by optimizing the number of network requests.

[4]Goodreads.com, `http://goodreads.com`
[5]Sefaklic Goodreads Wrapper `https://github.com/sefakilic/goodreads`

The data downloaded from Goodreads API will then be saved in the Cassandra cluster by using a Python Cassandra Driver[6].

We expect to have a least 1000 users with related reviews and categories to beginning the recommendation tests.

From the Gooreads API, we can retrieve the information about the books, the users, and their reviews. In addition, it's possible to extract the friends relation among the users.

The shema of the Goodreads database is composed by the following tables:

- Books
- Authors
- Reviews
- Shelves
- Users

For every book, we can retrieve its ISBN, the authors of the book as well as the user reviews. Besides, a book has a list of "shelve" that describe its category and personal feedback by users.

As described before a "shelve" is similar to a tag that every user can add to a book. Every user can furthermore vote the shelves to give them more relevance. This information is very useful for our use case because they characterize a book in a better way than a simple category assigned by the publisher.

Every user can also make several reviews on books both with a star rating or with text. The API of Goodreads did not allow, at this time (2015), to retrieve the reviews give users but we can do this quite easily by crawling the official website.

The first version of the database schema is presented in figure 3.2. In this version, every element of the dataset has a separate table associated with several relational tables to maintains the relationship between the elements.

[6]Python Cassandra Driver, `https://datastax.github.io/python-driver/`

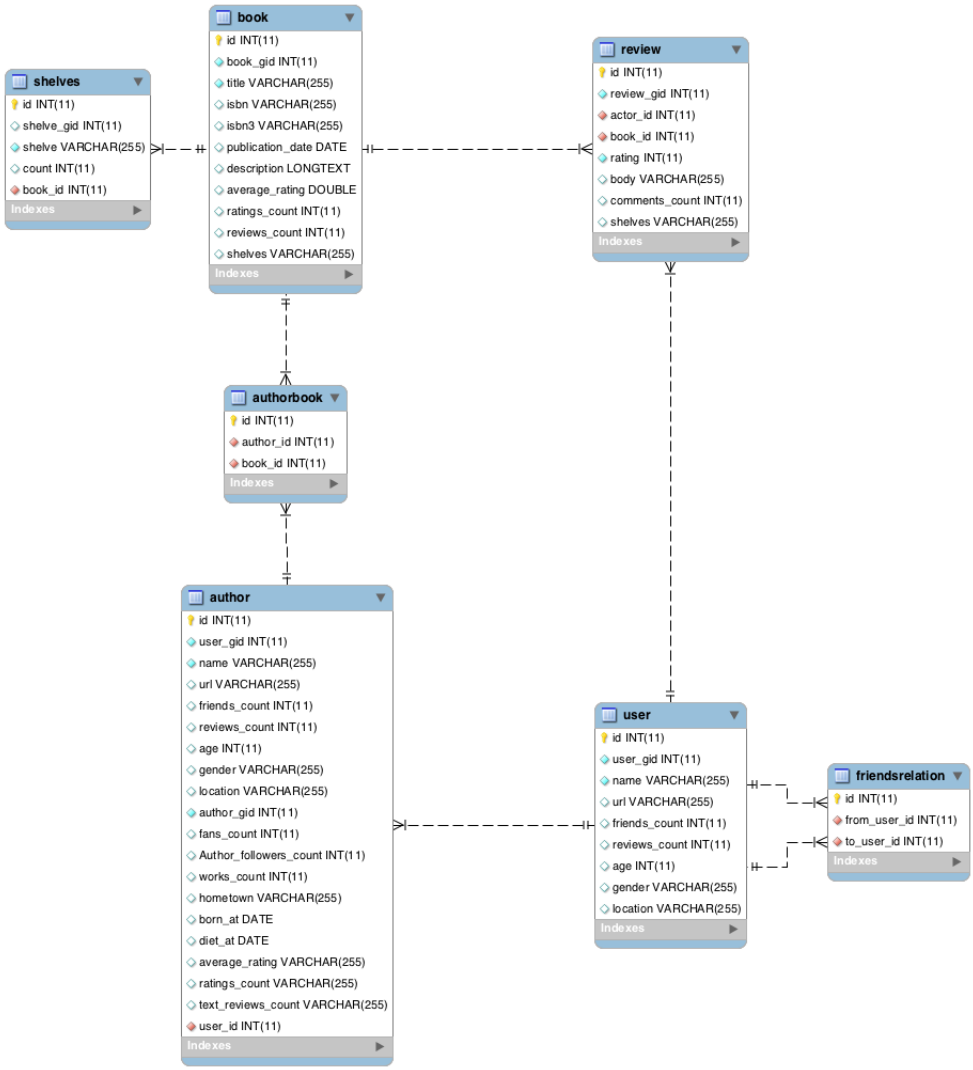


Figure 3.2: First version of the database shema of goodreads dataset

The first release of the database has a good structure, but we can see that a lot of joins is required if we want to retrieve all these information. If we want, for example, to retrieve all the reviews made by a user with the related books and shelves we have to perform the following steps: (1) join from the table *shelves* to *book* (2) perform a join from the table relation *AuthorBoook* and the two table *book* and *author* and (3) made a join from *review* to *user*. As our database can have a dimension of 15 GB, these joins are quite expensive. In addition, in the context of a recommender system, this query will be performed very often.

An improvement of the schema has been created by storing the list of reviews for every user within the same table. By using Cassandra database we can save them in a User Defined Type (UDT) list in a column of the user table (figure 3.3).

In this way If we want to retrieve the reviews made by a user we only have to retrieve the entire row included the list of the reviews. This can be done without any additional joins.

Of course storing the list of reviews for every user needs that some of the data will be duplicated, and we expect an increment of the dataset size. On the other hand, we have a good

improvement in reading the database with this particular query on the user table.

The modified and optimized schema is showed in figure 3.3.

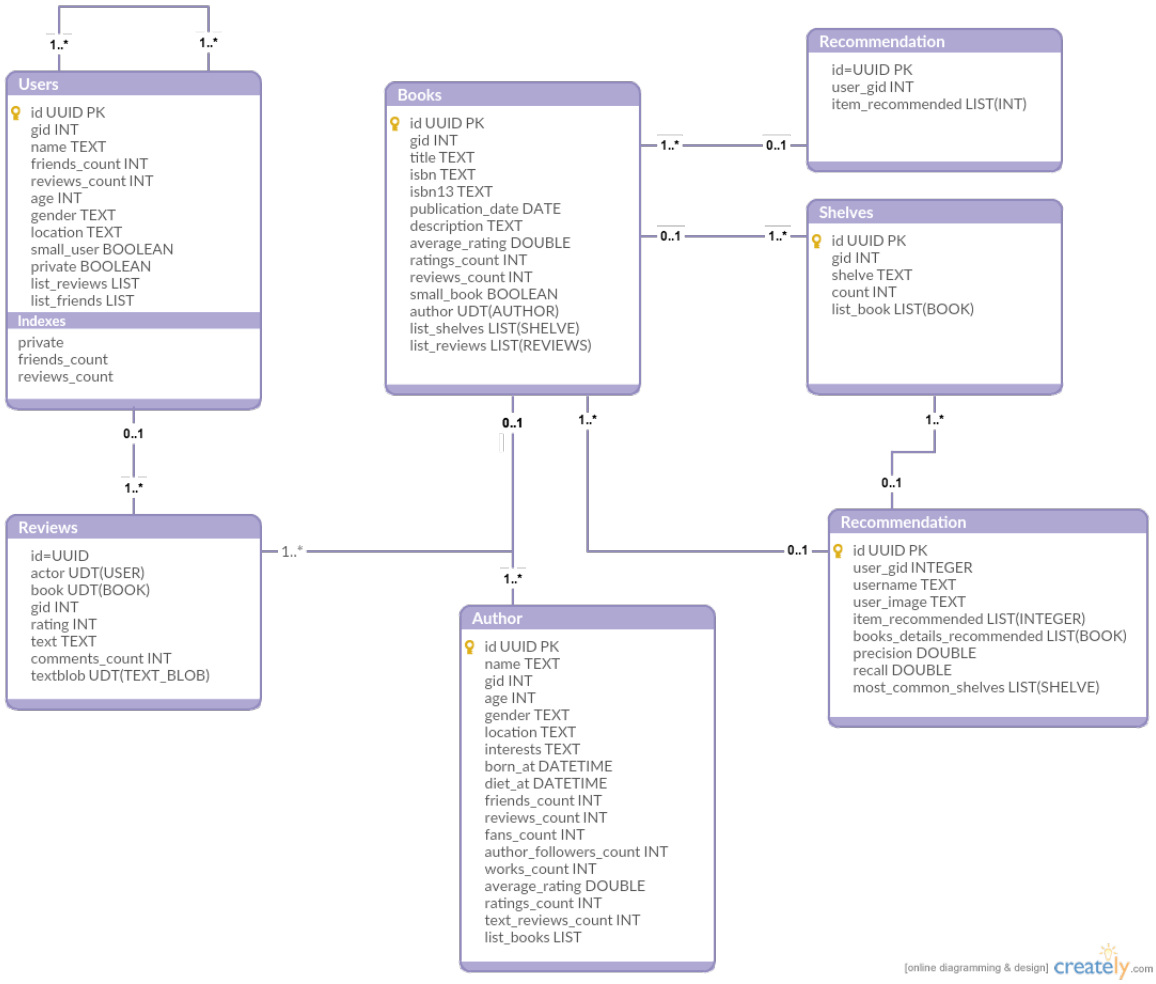


Figure 3.3: Second version of the database shema of goodreads dataset

As we can see from the figure 3.3 we kept the most relevant tables, but we have structured the relations in several lists internally in the each table. In this way, it is possible to perform several optimized queries. For example, we can retrieve all the reviews for a particular user without the need to do joins. In fact, the list of the reviews is already saved in the users table in the "list_reviews" field.

We also added a new table "Reccomendation" where we store the recommended items with the related Precision and Recall of the recommendation. This table is actually used for the little web application that demonstrates the results of the Recommender System.

3.1.2 Epinions dataset

In figure 3.4 is showed the schema of the Epinions dataset [1]. For constructing the dataset compliant with the input data of our algorithm, we join the user, reviews, and the product table to retrieve all the needed data. We export the data to a large CSV file. For the TrustWalkers algorithm, we join the data with the "trust" table as it contains the trust relations between the users.

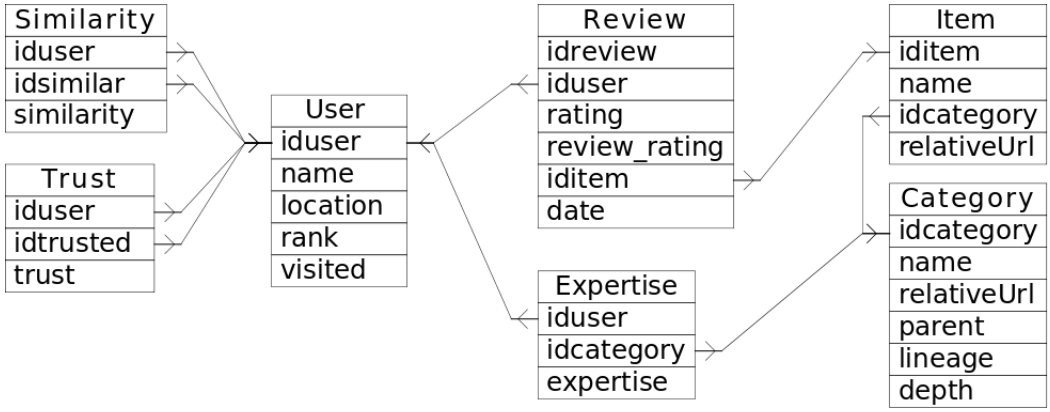


Figure 3.4: Database shema of Epinions.com dataset [1]

3.2 FWUM algorithm

In the previous chapter, we already discussed several techniques that we can use to performing a recommendation of products. One of the choices that we made in the analysis part is to use, for the first part of the algorithm, a hybrid approach described in [1]. In this algorithm, we use both items feature and users rating information to making the recommendation. More specifically we weighting the items feature based on their frequency in the users profiles.

The steps of the proposed algorithm are the following:

1. Construction of the content-based user profile by using both collaborative and content features
2. Feature weighting steps: We weighting the features based on their importance and rarity among the other users
3. Retrieve the neighborhood of users by calculating their similarity
4. Retrieve the Top-N items for that will be recommended to the users

3.2.1 Construction of the content-based user profile step

As we can se in figure 3.5 we have three matrices that represent the user ratings matrix in figure 3.5a, item features matrix in figure 3.5b and the user features in figure 3.5c.

[1]Epinions.com dataset, `http://liris.cnrs.fr/red/`

	I_1	I_2	I_3	I_4	I_5	I_6
U_1	-	4	-	-	5	-
U_2	-	3	-	4	-	-
U_3	-	-	-	-	-	4
U_4	5	-	3	-	-	-

(a) User-item matrix

	F_1	F_2	F_3	F_4
I_1	0	1	0	0
I_2	1	1	0	0
I_3	0	1	1	0
I_4	0	1	0	0
I_5	1	1	1	0
I_6	0	0	0	1

(b) Item-feature matrix

	F_1	F_2	F_3	F_4
U_1	2	2	1	0
U_2	1	2	0	0
U_3	0	0	0	1
U_4	0	2	1	0

(c) User-feature (user profile)

Figure 3.5: Initial matrices[1]

The user-feature matrix is calculated by counting the feature of every rated items by users. For example if we look at the ratings of *U1* we can aggregate the feature of items *I2* and *I5* that have respectively the features *F1,F2* and *F1,F2,F3*. Therefore, in the user-feature matrix *U1* has *2* times *F1* and *F2* and *F3* once. These values represent the correlation of user to every features.

3.2.2 Feature weighting step

At this point we want to weight features in order to retrieve: (1) the most representative features for a user and (2) those that better distinguish one user from others.

To doing that, we use a model inspired by the Information Retrieval (IF) field by using the popular TFIDF calculation. In our model the Term Frequency (TF) is referred to the frequency that a feature f has on user U denoted as Feature Frequency (FF). These values are calculated to providing an intra-user similarity measure. Furthermore, inter-user dissimilarity is calculated by the inverse frequency of a feature f among all users in the dataset. This factor is referring to IUF. Thus, FF(u,f) is the number of times that a feature f occurs in the profile of user u and UF(f) is the number of users in which feature f occurs at least once. Finally IUF can be computed with following formula:

$$IUF(f) = \log \frac{|U|}{UF(f)} \tag{3.1}$$

Where in equation 3.1 the $/U/$ is the total number of users.

If the IUF(f) is hight the feature f is rarely presents on users profiles, inversely is a very common feature among all users profile. By using this factor we can finally weight every feature using FF and IUF as described by equation 3.2:

$$W(u, f) = FF(u, f)IUF(f) \tag{3.2}$$

This feature weighting model gives more importance to features that frequently occur in the user profile. On the other hand, it gives less importance to very common features that occur in many user profiles.

This makes sense if we think about two users that both have liked a book of the category "Fantasy". They are not necessarily similar because this feature is very popular among users and most of them have rated it (Ex. most of the peoples like Harry Potter or the Lord of the Ring). On the other hand, if two users have both liked books of a very rare category, let's say "Egyptian literature," they have more chance to be similar because only a few users are interested in this category. It is, therefore, reasonable to weighting the item features according to their rarity in the whole users set.

After the weighting process, we have a new matrix represented in figure 3.6a. As we can see F4 for U3 has a greater weight even if occurs only once in the user profile. This is because F4 is a rare feature among users, this cannot be seen on the previous user-feature matrix in figure 3.5c.

Once the characteristics of the users profiles are weighted correctly, we can use the user-feature matrix to calculating the similarity between users by using the cosine similarity measure. Cosine similarity gives particularity good measure as it takes into account the weight of every feature to compare the two user vectors. The user-similarity matrix resulting can be seen in figure 3.6b.

	F_1	F_2	F_3	F_4
U_1	0.60	0.24	0.30	-
U_2	0.30	0.24	0	-
U_3	-	-	-	0.60
U_4	0	0.24	0.30	-

(a) User-item matrix

	U_1	U_2	U_3	U_4
U_1	-	0.96	0	1
U_2	0.96	-	0	1
U_3	0	0	-	0
U_4	1	1	0	-

(b) User-feature (user profile)

Figure 3.6: User profile and user similarity matrices after the feature weighting step[1]

3.2.3 The user's neighborhood formation step

To provide a recommendation, we need to retrieve a set of similar users. By using the matrix in figure 3.6b we can easily retrieve the most similar users for a given user u as we know that a similarity near to 1 means that there is a high similarity between the two users.

3.2.4 Top-N list generation step

In the approach described in [1] they take into account the positively rated items inside the found neighborhood. We can see the whole process in the figure 3.7. More precisely, for every item evaluated by similar users, they count its user features frequency in the neighborhood. Therefore, they weight every item by the sum of the frequency of every feature that it consists of. Then, we retrieve the N items with the biggest influence on the whole sets, so these with the highest sum of the features frequency. For more details, these steps are described in [1, p. 7].

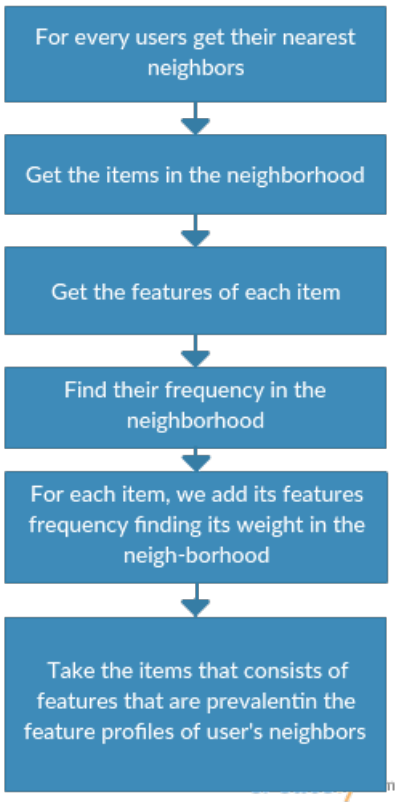


Figure 3.7: Top-N items list generation process

3.2.5 Parameters

The FWUM algorithm has four main parameters that we can adjust to maximizing the recommendation quality. In particular, we can adjust the *Number of recommended items* to present to the users and the *Number of similar users* that we take into account in the Top N generation step (section 3.2.4) for the construction of the neighborhood. We can also set up a *Minimum user similarity* to filter the users with the lowest similarity. Then we can set up the *Minimum shelve rating count* threshold to filter only the most popular shelves.

3.3 Trust Walkers algorithm

To take into account social information and try to improve the quality of the recommendation we decided to use the Trust Walkers algorithm described in [2] .

The objective of this algorithm is to retrieve the ratings for items that the user haven't rated yet. To do this, we take advantage of the social network of friends (or a trust network) to retrieve the friends ratings on the items and estimate their ratings. In short, to predicting item i (figure 3.8) for user u they perform several RWs starting from u until they reach an user v that have rated item i or when performing a maximum number of steps *k-max*. We also take into account when user v have rated an item similar to i. The similarity between items are calculated with the PCC on the user ratings, but also VSS can be used as well.

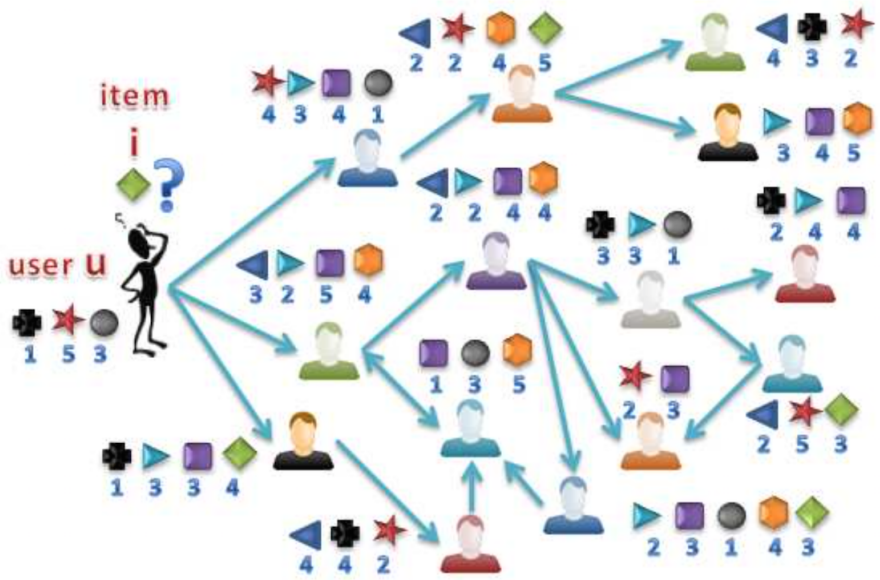


Figure 3.8: Representation d'un social network or a trust network [2, p. 2]

The main steps to doing this are: (1) Perform several RWs to finding user v that have expressed rating on item i or items similar to i (2)For each Random Walk (RW) retrieve an estimated rating for item i (3) Repeat step 1 and 2 several times and aggregate the rating retrieved from the RWs to predicting the rating for item i.

To perform a RW in the social network graph we start from user u and continue to randomly selected neighbors v of u . With a probability of $\phi_{u,i,k}$ we stay at this node and we take the rating of items i or of the most similar item to i rated by v. With the probability of $1 - \phi_{u,i,k}$ we continue our random walk and walk to another direct friend or trusted neighbors of v.

To calculate the similarity between the items we use the cosine similarity of the item i and the items rated by v. Alternatively, we can use the PCC if it gives better results or if we do not have enough content-based information on the dataset.

The $\phi_{u,i,k}$ is the probability of staying at node v and it is calculated by the maximum similarity of items rated by user v with target item i. The factor k then regularizes this value by using a sigmoid function to retrieve a value between 0 and 1. The factor k determines the distance to the source and, when k is big, the probability of staying at node u and terminate the random walk, is very high.

The formula to calculate $\phi_{u,i,k}$ in every node is the following:

$$\phi_{u,i,k} = \max_{j \epsilon RI_u} \quad sim(i, \quad j) \times \frac{1}{1 + e^{-\frac{k}{2}}} \tag{3.3}$$

There is a chance for a RW to continue forever. To avoid this case we stop the random walk when we are very far from the source (k > *max-depth*). We can set the max-depth=6 according to the "six-degrees of separation" rule [13], without loosing important information.

After performing several random walks, we can aggregate the results to retrieve the esti-

mation of the rating for items i. In our case, we compute the average of the random walks results.

The number of random walks to retrieve a reliable result depends on the dataset but we can take into account the variance as follows:

$$\sigma^2 = \frac{\sum_{i=1}^{T} (r_i - \bar{r})^2}{T} \tag{3.4}$$

Where the the r_i is the result of the i^{th} random walk and $\bar{r}$ is the average of the results returned by random walks.

We can then terminate the Trust Walkers when the variance between the last random walk and the current random walk does not change the results significantly, or in other term, the variance is very small or where $|{\sigma_{i+1}}^2 - \sigma_i| \leq \epsilon$ where ϵ is a constant value.

3.4 Combining the FWUM and the TW

To improve the performance of the recommendation, there is the possibility of combining the FWUM and the TW. An idea of doing this is to combine the two algorithms as follows. Firstly we compute the recommendation with the FWUM. In this case, the number of recommended items can be large. Subsequently, we compute the TW on it to try to estimate their rating values by the judgments of the neighbors in the social/trust network. The items with the highest predicted rating can then be recommended to the user. Another option would be to run the two algorithms and merge their best results in the final recommendation. In addition, we can add a weight to adjusting the contribution of the TW and the FWUM in the recommendation results as proposed in [18].

3.5 Conclusion

In this chapter, we have spoken about the architecture of the systems and the two algorithms that will be implemented in the next phase. In the section 3.1 we have described the functioning of the systems and the database schemes to the Goodreads and the Epinions datasets. In particular, we have spoken about several optimization made on the Goodreads dataset to improving the reading performance by excluding the need of performing some expansive joins between the tables.

In section 3.2 and in section 3.2 we explain the details of the *FWUM* and the *Trust Walkers* algorithms.

Implementation

In this chapter we will explain the steps to configure Cassandra (section 4.1), how to populate the database(section 4.3) and the most important steps of the implementation of the two algorithm: the FWUM and the TW. We also explain how to launch a Spark application into a cluster, and we show the Web UI that has been established to better visualize the results.

4.1 Cassandra configuration

As we discussed in chapter 2.3 we use Cassandra as the database to stock all the data retrieved from Goodreads.com. At the beginning to install and configure Cassandra we used the instructions found on the Datastax website in the article: "Installing DataStax Distribution of Apache Cassandra 3.x on Debian-based systems"[1]. After some research, we found a more proper way to install it by using a ready EC2 AMI and by configuring it by the guide found in the article "Installing a Cassandra cluster on Amazon EC2"[2].

By following this instruction we can launch a Cassandra cluster in three steps:

1. Launching the DataStax Community AMI
2. Creating an EC2 security group
3. Creating a key pair
4. Connecting to your DataStax Community EC2 instance
5. Clearing the data for an AMI restart
6. Expanding a Cassandra AMI cluster

As we can see the steps for launching a Cassandra node are similar to those used to launch a normal Elastic Compute Cloud (EC2) instance from AWS. For adding more node (point 6) we can use the OpsCenter fount at *http://INSTANCE_IP:8888* and use a graphical wizard to add and launch a new node, for this part all the configuration are made transparently to the administrator.

When the cluster is ready we can create our tables by initializing first our keyspace by the following CQL command:

```
calsh> CREATE KEYSPACE prs
WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
```

[1]Installing DataStax Community on Debian-based systems, 2016, `https://docs.datastax.com/en/cassandra/2.0/cassandra/install/installDeb_t.html`

[2]Installing a Cassandra cluster on Amazon EC2, 2016, `http://docs.datastax.com/en/cassandra/2.1/cassandra/install/installAMI.html`

The "Simple Strategy" means that we will use only one datacenter and the "replication_factor=3" means that we have three replicas of every row placed on different nodes. With the Simple Strategy, the replicas are placed on the next node of the cluster clockwise. If we want to use multiple datacenters then, we can use the keyword "NetworkTopologyStrategy" that places the replicas on different datacenter for better fault tolerance (even if an entire region fail).

We can then the create a table by the following CQL command:

```
calsh>CREATE TABLE users (
id uuid PRIMARY KEY,
name varchar,
...
);
```

In this example, the primary key is the id of the users an by default will be the Partitioner key that Cassandra uses to partitioning the data on the cluster. In short, Cassandra calculates a hash function on that key to distributing every row across different nodes. To have some balanced nodes, we need a partitioning key to be as random as possible. That's why we use a random UUID key in our tables.

Optionally we can use the clustering key to ordering the results by a particular field. In this way, using the clustering key in some specific queries can be more efficient.

Finally, we populate our database using our proper Python API and the Python Cassandra Driver[3]. We will speak about this in Section 4.3

4.2 Spark cluster set-up

To execute a Spark application on a EC2 Cluster we need to do the following steps:

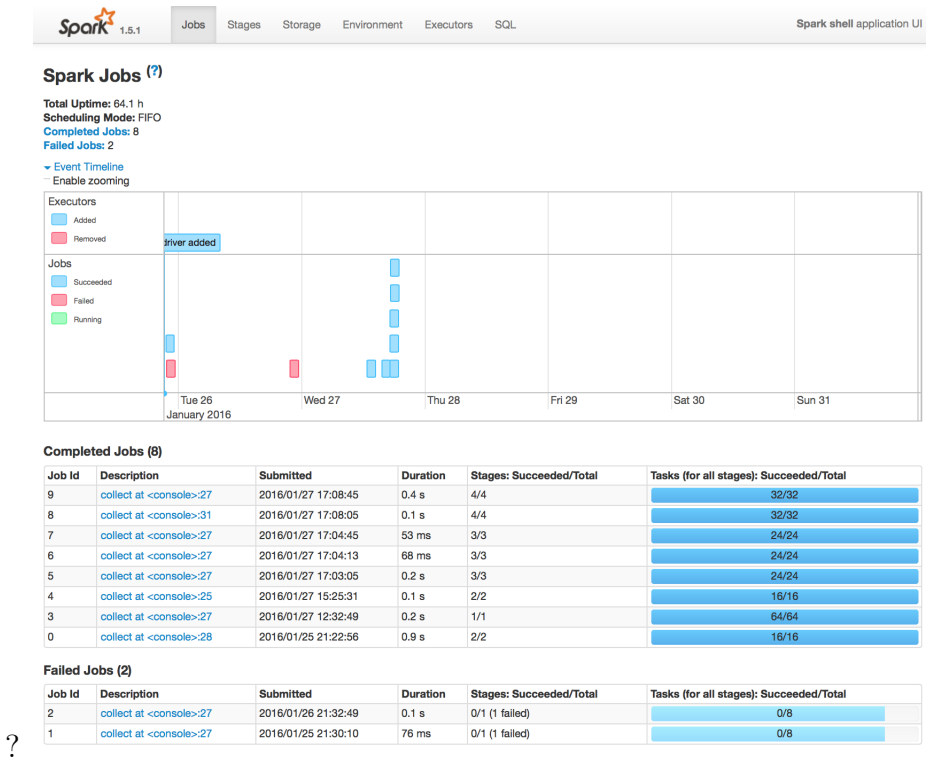
1. Launch a Spark EC2 cluster
2. Connect to the Master driver
3. Build the source code in local and create a jar containing all the necessary dependencies
4. Copy the compiled JAR to the Master of the EC2 cluster
5. Configure the credentials on the Master to connecting on AWS S3
6. Run the spark application
7. Monitor the running application on the Spark UI

Every steps have been described in details on the "Installation guide" in the annex F.

[3]Python Cassandra Driver, 2016, http://datastax.github.io/python-driver/api/

4.2.1 Monitor the running algorithm

We can monitor a Spark application at the *http://MASTER-IP:8080*. In the jobs panel showed in figure 4.1 and accessible from the address *http://MASTER-IP:4040/jobs/* we can see the time-line of all the events in the applications and the list of the jobs queue.



?

Figure 4.1: Spark UI

For every stage we can see their *event timeline*. In Figure 4.2 we can see that this stage is spread out across several machines (3 showed). Each bar represent a single task in the stage. We can see from this example that the majority of the time is taken for the computation (in green) and not in I/O operation (in red). In addition, we can see that the executor are running up tp 4 tasks in parallel using their cores. The Figure 4.3 shows the list of operation for a specific job. In this case, we do a simple join of two lists, and we return the unique values. Thus, Spark performs a two parallelization of lists with *parallelize()*, then perform the join and finally execute the *distinct()* operation.

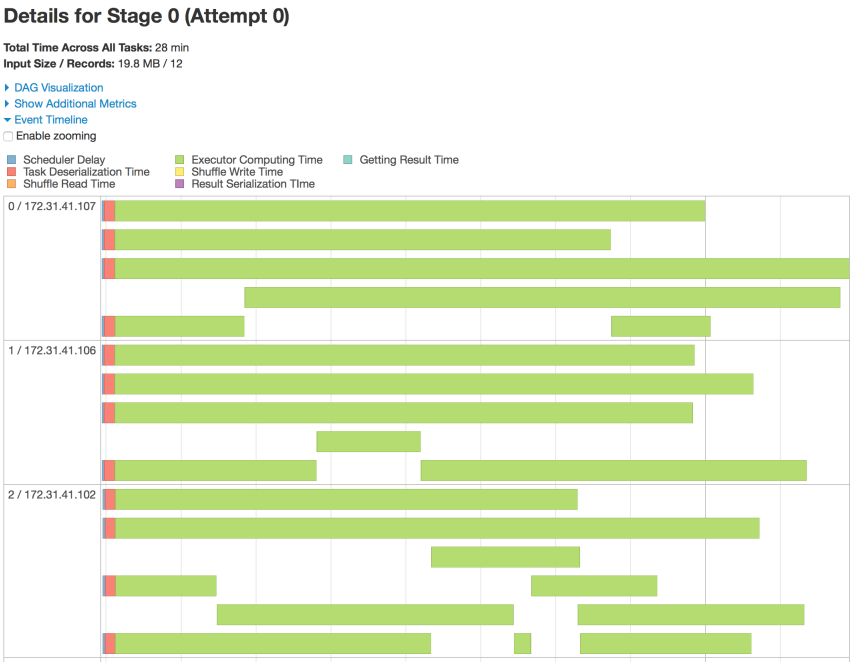


Figure 4.2: DAG representation

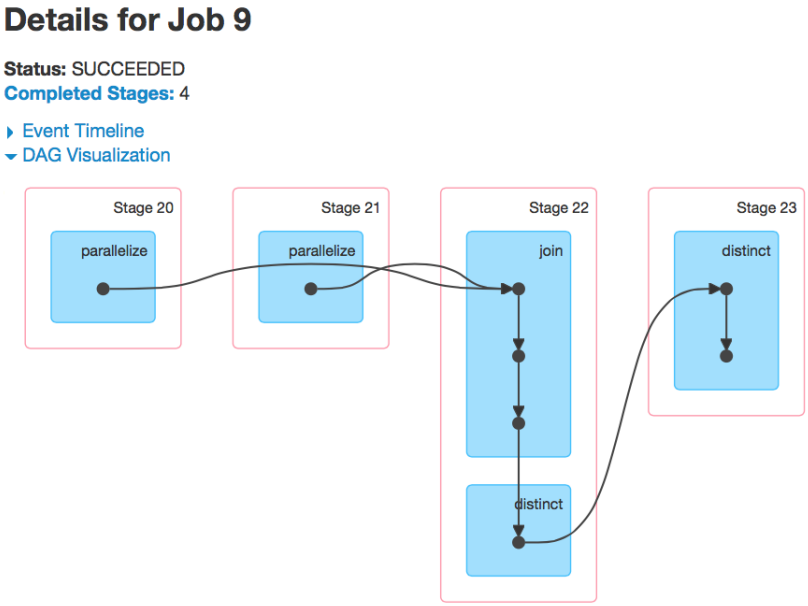


Figure 4.3: Events-timeline

4.3 Goodreads Python API

To constructing the Goodreads dataset we build a Python API based on the code of the "Sefakilic Goodreads API" [4] that provide to us a starting point for the development. As described in the design chapter on section 3.1 we develop a new API because we have to optimize the number of requests pro seconds and respect the limit of 1 request pro second

[4]Sefaklic Goodreads Wrapper `https://github.com/sefakilic/goodreads`

imposed by the Goodreads HTTP API[5]. We also use the Python Cassandra Driver [6] that allows to work with Python object and save them on Cassandra Cluster.

To create a table we define a class on Python and we only need to declare it on the following manner:

```
class Reviews(Model):
    id=UUID(primary_key=True, default=uuid.uuid4)
    actor=UserDefinedType(user, default=user())
    book=UserDefinedType(book, default=book(author=author()))
    ...
```

Where the rating and the text field are of a pre-defined Cassandra types. The id, actor and book field are, instead, some *User Defined Types* that we can define like a standard Python class by just extending the *UserType* class.

In the code above we declare the user defined type "*user*":

```
class user(UserType):
    id=UUID(primary_key=True, default=uuid.uuid4, partition_key=True)
    name = Text(required=True)
    ...
```

To create a new record we only have to perform the following operation:

```
user=Users(id=uuid.uuid4(), name='Simone', ...)
review=Reviews(id=uuid.uuid4(), actor=user, ...)
review.save()
```

By looking at the UUID of the object, Cassandra can update or create a new row on the Table "Users". More information of the installation of the Goodreads Python API can be found on the "Installation guide" in the annex F.

4.3.1 Problems encountered

To optimizing the request pro seconds on the Goodreads API we have to re-implement the majority of the function of the *Sefakilic Goodreads API*. By doing this we encountered a lot of spacial case to treat. For example to retrieve the book from the Goodreads HTTP API at the address "https://www.goodreads.com/book/show.xml?id=BOOK-ID &key=KEY" the XML returned can have multiple representation depending on what information are present in the book. For this reason we had to treat all the possible cases, that toke several days of work, to avoid to have errors in the database creation phase.

Another problem that took some time to resolve concerns the authentication to retrieve some information in the Goodreads API. More specifically to retrieve the user friends we have to perform an authentication using auth 2 to access to this information.

[5]Goodreads API, 2015, `https://www.goodreads.com/api/`
[6]Python Cassandra Driver, `https://datastax.github.io/python-driver/`

4.4 FWUM algorithm

4.4.1 Construction of the content-based user profile step

As discussed in the design chapter of this document in section 3.1 we used Spark to implement the algorithm, perform the analysis of the data, and produce the recommendation. The main advantage of using Spark (as discussed in section 2.4) is that it use the RDD as the principal abstraction to representing a collection of immutable element that can be manipulated and operated in parallel. In addition they are distributed on the machine in the cluster and they are fault tolerant. In addition to that, Spark offers a lot of useful libraries that can be used to speed up the development. The details of the steps of the

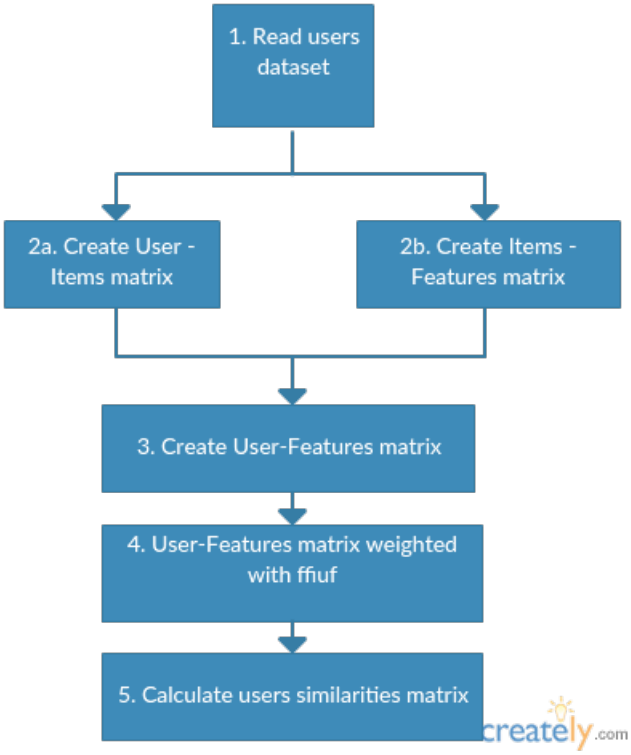


Figure 4.4: General steps for the hybrid algorithm construction

implementation of the FWUM algorithm are described in the chapter 3.2. In this chapter we want to give an overview on main steps performed with Spark. In the Figure 4.4 is illustrated the general process. In the first step we read the list of users from the database. In the beginning, we read directly from the Cassandra database. Then, to optimizing the reading speed (more details in section 4.4.6) we save the dataset directly on S3 to improve the reading performance using the *sc.objectFile()* method. The information of the users reviews is represented in an RDD of tuple with the id of the users as key and list of reviews as value. Every review is represented as a tuple of the item id and the array of categories. The final structure the users reviews has the following signature *RDD[(Int, Array[(Int, Array[Int])])]*.

We split then the reviews in train and test set. We choose 75% of the users reviews as the train set and the remaining 25% in the test test. The items are selected randomly. The structure of the RDD is composed by a tuple where the key is the user id, and the value is

a tuple of two arrays: the train and test set of reviews. The final RDD have the following structure: *RDD[(Int, (Array[...]), Array[..])]*.

We create then the User-Items and the Items-Features matrices. We use for this purpose the CoordinateMatrix class and the BlockMatrix class of the Mllib library of Spark. We have chosen to use the BlockMatrix class as they allow us to perform a matrix multiplication with two very large matrices. The matrix multiplication is very useful to creating the User-Features matrix in the step 3 by multiplying the User-items matrix by the Items-Features matrix. This method works very well for a little dataset but we see an exponential complexity when the dataset dimension increases. For this reason, on the optimized version of the algorithm, we implement ourselves the matrix multiplication by using only the necessary values. More details of this optimization are described in section 4.4.6. To perform this operation, we calculate the feature frequency of every user. This is actually the representation of the User-Feature matrix. In the code below we can see that we group the user features, and we calculate their frequency:

```
val user_ff: RDD[(Int, (Int, Int))] = rdd_splitted.flatMap{case (u, (train, test))=>
    val shelves=train.flatMap(_._2)
    shelves.groupBy(x=>x).map(x=>(x._1, x._2.length))
    .map{case (s,ff)=> (s,(u, ff))}
}
```

The User-Feature matrix is represented by an RDD of tuple with a shelve id, the user id and the shelve frequency.

4.4.2 Feature weighting step

The User-Feature weighted matrix is calculated as described in the algorithm details in section 3.2.2 where we explain how to weight every feature based on their FF and IUF measure. In short, we give a high weight to the characteristics that best represent a user profile and that best differentiate it from the others. Thus, we firstly calculate the Inverse User Frequency of each features:

```
val s_ffw: RDD[(Int, Double)]=user_ff.map { case (s, (u, ff)) => (s, u) }
    .groupByKey().map { case (s, s_u) =>
        (s, log10(users_size.toDouble / s_u.toArray.distinct.length.toDouble))
    }
```

We use these value to weight every features that below to the users.

```
//Weight every features
val user_ffw = user_ff.join(s_ffw).map { case (s, ((u, ff), fw)) =>
    (u, (s, fw * ff.toDouble))
}
```

4.4.3 The user's neighborhood formation step

Finally, we calculate the user similarities by calculating every user-user pair cosine similarity to construct the final User-User similarity matrix.

```
val similarities_pairs = user_ffw.cartesian(user_ffw)
    .filter { case ((u1, m1), (u2, m2)) => u1 != u2 }
    .map { case (((u1, a1), (u2, a2))) => (u1, (u2, cosineSimilarityList(a1.toList, a2.toList)))}
```

4.4.4 Top-N list generation step

By using the User-User similarity matrix we can perform the Top-N list generation step described in details in section 3.2.4. To retrieve the top-N items, we take the k most similar users by sorting them by the cosine similarity value calculated before:

```
//Get the N most similar users
val n_mostSimilarUsers = similarities_pairs.flatMap { case (u, sim_u) =>
    sim_u.toArray
        .sortBy(_._2).map(_._1)
        .reverse
        .take(k)
}
```

Then, we want to retrieve the best items we count its user features frequency in the neighborhood. Therefore, we weight every item by the summing the frequency of every feature that its consists of. Then we retrieve the N items with the biggest influence in the whole sets, meaning these with the highest sum of the features frequency.

With this code we retrieve the Feature Frequency in the Neighborhood:

```
val shelve_user_nbf: RDD[((Int, Int), Int)] = n_mostSimilarUsers.join(user_ff_list)
.map { case (siu, (iu, is_ff)) => (iu, is_ff) }
.groupByKey()
.flatMap { case (iu, is_ff_a) =>
    val map_is_ff = is_ff_a.flatten
                           .groupBy(_._1)
                           .mapValues(_.map(_._2).sum)
    map_is_ff.map { case (s, ff) =>
        ((s, iu), ff)
    }
}
```

The results is an RDD of tuple where the key is the feature id(Int) and as value, we have the user id that it below (Int) and its feature frequency value (Int).

The final step is to get the N items that consist of features that are prevalent in the feature profiles of the user neighborhood. to doing this, we sort the elements by the sum of their feature frequency value, and we take the top N items.

```
val top_N_items: RDD[(Int, Array[(Int, Int)])] = nb_shelve_user_items.join(shelve_user_nbf)
        .map{case ((is, iu), (ib, nbf))=>((iu, ib), nbf)}
        .reduceByKey((nbf1, nbf2)=>nbf1+nbf2) //Sum the feature frequency
        .map{case ((u, b), nbw)=>(u, (b,nbw))}
        .groupByKey() //Group by user
        .map { case (u, b_nbw) =>
            (u, b_nbw.toArray
                .sortBy(_._2) //Sort by the sum of feature frequency
                .reverse
                .take(N)) //Take the top N items
}
```

The structure of the results is an RDD of tuple where the key is the user id, and the value is a list of items recommended.

4.4.5 Metrics

After calculating the recommended items for every user, we can now calculate the Precision and Recall by comparing the list of recommended items and the test items. To doing this firstly we retrieve the test set that we have created before. Then, we calculate the number of items recommended that appears in the test set. These are called True Positive (TP). Then we calculate the False Negative and the False Positive. (more explanation of these measure are explained in the section 5.1.

To better understanding the meaning of these values we can see in table 4.1 how are calculated. These value are calculated in Spark as follows:

	Recommended	**Not recommended**
Preferred	True-Positive (TP)	False-Negative(FN)
Not preferred	False-Positive (FP)	True-Negative(TN)

Table 4.1: Confusion matrix of the recommendation results [19]

```
val user_tp = results.join(testItems).map{case (u,(b_res, b_test)) => (u, b_res.intersect(b_test).length.toDouble)}
val user_fn = user_tp.join(testItems).map { case (u, (tp, b_test)) => (u, b_test.length − tp) }
val user_fp = user_tp.join(results).map { case (uid, (tp, recom)) => (uid, recom.length − tp) }
```

The *results* variable store the top N items recommended to users, the *testItems* variable store the test items calculated in precedence. The Precision and the Recall metrics are calculated with the following formulas:

$$Precision = \frac{tp}{tp+fp}$$

$$Recall = \frac{tp}{tp+fn}$$

With Spark we calculate it as follows:

```
val user_precision = user_tp.join(user_fp).map { case (uid, (tp, fp)) => (uid, tp / (tp + fp)) }
val user_recall = user_tp.join(user_fn).map { case (uid, (tp, fn)) =>
    val recall = tp / (tp + fn)
user_precision.saveAsTextFile(ResultfolderName + "/user_precision")
...
```

In the chapter 5 we describe the results of this evaluation by performing a recommendation for more than 5000 users. The parameters of this algorithm are the *Number of similar users* (k , default value 40), the *Number of recomended items* (N , default value 20) and the *Percentage of shelves* (S, default value 10%). In section 5.2 we made some test to choosing the best parameters and optimizing the performance of the algorithm.

4.4.6 Optimization

4.4.6.1 Matrix multiplication

In the first version of our algorithm we used the BlockMatrix contained in the MLlib library. This is the only class that allows to perform a matrix multiplication with two Distributed Matrix. Unfortunately this class does not optimize the sparsity of a matrix. In our case this is very dangerous as we can have a matrix multiplication very large and sparse.For around 5000 users recommendation we can have an user-items matrix with a dimension of 5000x200'000 and an items-features composed of 200'000 rows and 20'000 columns with a total of 20'000 Billions of values. As the complexity of a matrix multiplication is O(nmp) we can see that it is very computational complex to perform such an operation. Thus, we decided to implement the matrix multiplication by using the basic operations provided by Spark. In this way we are able to consider only the real value of the matrix which are in the order of 5 Millions. We can see from the figure 4.5 that using our implementation of the matrix multiplication i much more efficient to compute the recommendation for more than 1000 users. For less recommendations, the performance is acceptable for both the algorithms.

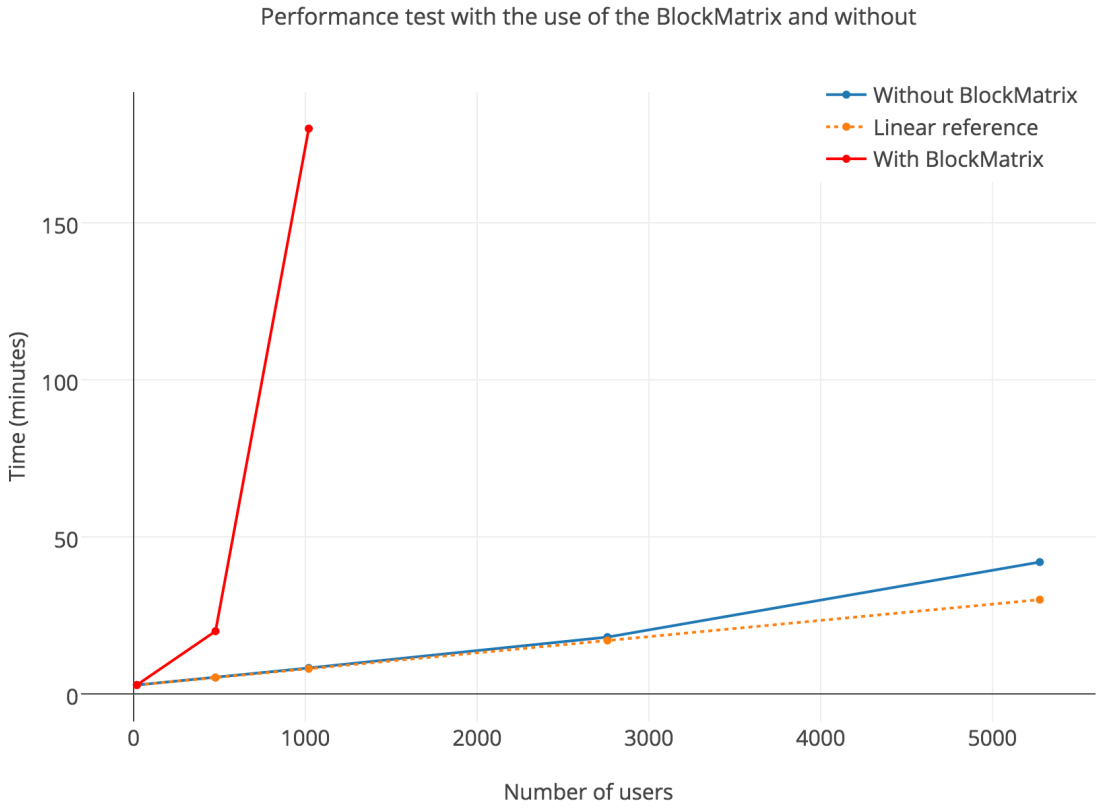


Figure 4.5: Comparing the performance of the algorithm by using the Block Matrix multiplication and without

4.4.6.2 Database size reduction

We have noticed that a huge amount of data were used for the "shelves" strings associated with every item in the Goodreads database. A "shelve" is represented with a text of 10 or more character. Since every item in the dataset has between 50 and 100 shelves, and there are more than 200'000 items in the database we have an enormous memory consumption because of these. Our solution was to represent each shelve by and integer number representing the string. This number is generated by the Scala library *hashCode()* method. This optimization has reduced the size of the dataset from 20GB to around 200MB. It's important to note that the values produced by the hash function are theoretically not unique. This means that there is a probability that some different shelves results in the same values. For this reason we made a test with 27653 shelves. We observer that only the 0.014% (4 of 27653) are duplicate values. Thus, we can say that this error is negligible in the case of this algorithm.

4.4.7 Problems encountered

In the testing phases of the FWUM algorithm, we have encountered several issues. In this section, we will explain all the solutions that we adopted to resolving them.

4.4.7.1 Waste of time when a node crash

When we perform an analysis with Spark, we have to read the dataset contained in the Cassandra dataset. In the beginning, we read directly from it by creating an RDD containing the dataset and we store it in the cache. We noticed that if a node fails and some cache memory were lost spark recompute it. In this case, Spark re-read the dataset from Cassandra. As seen in the debugging phase Spark loses a lot of time to re-read some piece of the RDD from Cassandra as it is on a different node.

We have resolved this issue by storing the RDD representation directly in the S3 service with the *.saveAsObjectFile("..")* method, and we read from it when required. In addition we cache the RDD in memory by calling *rdd.persist(StorageLevel.MEMORY_AND_DISK_2)*. This method tells Spark to store in memory the RDD with a replication factor of 2. By doing that, even if a node fails there is another replica of the RDD that can be used for the computation, without loosing time to re-compute the some missing information.

4.4.7.2 Out of memory errors

In some tests of the FWUM algorithm we have noticed some *Out Of Memory* error when we compute a large dataset. This error is produced when, in a single machine, we use too much memory. Even if in the Storage panel in spark every executor has enough memory, this does not mean the the heap space of all the executor is enough. In our case we did the following code.

```
.aggregateByKey(Map())((m1, v)=>{
v match { case (ib, is)=>m1+(is->(m1.getOrElse(...)}},
(m1, m2)=> m1++m2.map{case (k,v)=>k->(m1.getOrElse(...))}
)
```

This code goes Out of memory because the Map m1 is a local Map, this means that it has to be stored on a single machine. We can see from the code that at every iteration a new value is added to it, and this has caused the OOM error.

We resolved this issue by using the features offered on the RDD objects such as *groupByKey()* and *reduceByKey()* operations.

4.4.7.3 Inclusion of the dependencies

To run a Spark application into the Spark cluster we have to provide all the dependency of our application to the master. It is important also that we must avoid the duplication the dependency to when submitting the program. For example, all the library provided with spark (GraphX, Mllib, Spark core) are already present on the node of the cluster and the program crash if we include these as a dependency of the program. In opposite, we need them to compiling the program in local. To avoid this problem we use the sbt assembly plugin [7] and we set the dependency as "provided". In this the plugin don't include this library in the dependency but use it to compile our program.

4.5 Trust Walkers algorithm

The implementation of the Trust Walkers algorithm was quite different compared to the FWUM algorithm as it used a network structure instead of a matrix. For the context of this algorithm, we used the GraphX library to representing the trust network. Thus, we use an RDD of type Graph where each vertex represents a user and the edges represents the trust relations. The property of each vertex is the past ratings of the users divided into the train and test set.

As described in section 3.3, to predicting item i for user u we perform several RWs starting from u until they reach an user v that have rated item i or when performing a maximum number of steps *k-max*. We also take into account when user v has rated an item similar to i.

To construct the network of users we use the Graph class of the Mllib library of Spark. We initialize the graph by constructing an RDD of Edges and an RDD of vertices. With

[7] SBT assembly, `https://github.com/sbt/sbt-assembly`

these two RDD we can initialize the Graph. The properties in the RDD of vertices are the training reviews, the test reviews and a list of the random walks results with the k, the *source id* and the *RW result* value (initialized as -1, vertice id, -1).

```
//Construct the RDD containing the edge of the graph
val edges=user_id_trusted.map{case (uid, trusted)=>Edge(uid, trusted, "")}

//Initialize vertices properties in the last array we store (k, the source node id, estimated rating)
val train_set_users=user_reviews_splitted.map{case (uid, (train, test)) => (uid, (train, test}, Array[(Int, Long,
    Double)]((-1, uid, -1)))}

//Initialize the initial graph with the vertices and edges data
var initialGraph = Graph(train_set_users, edges)
```

The algorithm performs a RW by doing k random jumps to one of the neighbors of each vertex using the *collectNeighbors()* method on a Graph RDD. At each step, we save the intermediate results in the property of every vertex. When each Random Walk is completed, we collect the average of the results of the previous steps. Then, we calculate the mean of the random walks results to retrieving the estimated value of the item. This process has to be performed for every test item for which we want to retrieve their expected rating.

The pseudo code of this algorithm is described below:

```
performTrustWalkers(graph, listOfTestItems){
    estimatedValues=[istOfTestItems.lenght]
    for i <- 0 to Number_of_test_items:
        for rw <- 1 to Number_of_random_walks(max 1000):
            for k<- 0 to MaxSteps:
                graph.performRandomJump()
            graph.collectRWresult()
        estimatedValues[i]=graph.averagRWsResults(i)
    return estimatedValues
}
```

We terminate the overall process when there is no change in the results or when we performed a maximum number of random walks. In our project, we set this value at 1000.

After the termination condition is satisfied, we calculate the mean of every intermediate result of the RWs to retrieve the final prediction of the value of item i for user v.

This algorithm has been tested with a small dataset constructed by hand to testing the correctness of the computation but has not been possible to perform a full test of it. To evaluating this algorithm, it is necessary to use the RMSE metric.

4.5.1 Problems encountered

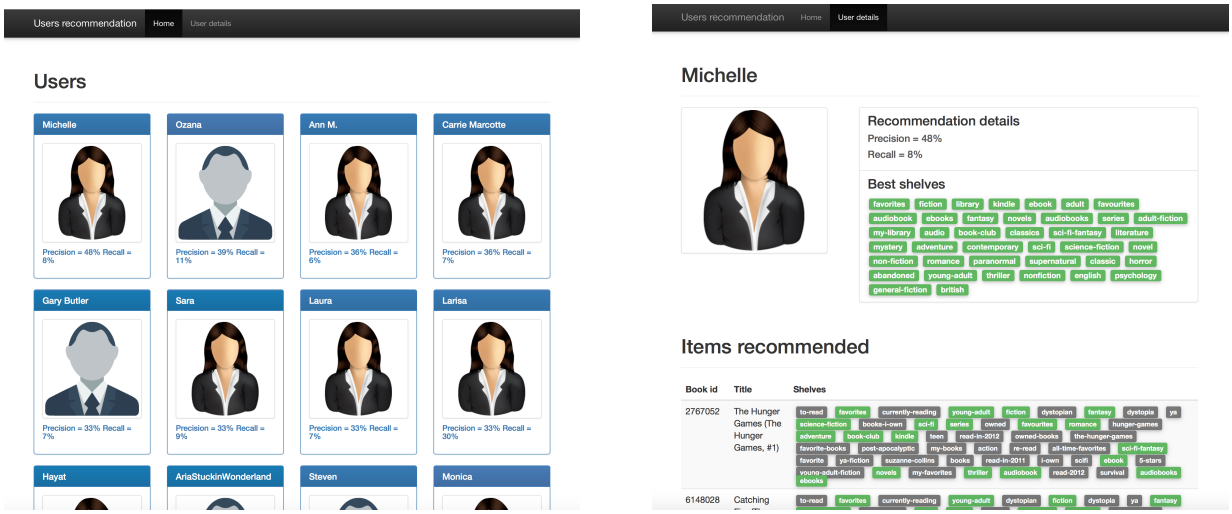
4.5.1.1 Items similarity colculation

To calculating the items similarity, we have to compare every pair item-item and calculate their Jaccard similarity. To doing this, we used the cartesian() method and we observed an exponential increase in the memory usage. One of the possible solutions would be to select only the useful items in the dataset and calculate the similarity on these sub-set of items. This will reduce the memory and the computation time considerably. In addition,

we can calculate the similarity of only the current items at every iteration to made several little calculation instead of a big computation at the beginning. These enhancements have not been implemented yet because of the dead line defined for this project.

4.6 Web UI

To better visualize the results of the recommendation it has been implemented an application that shows the top N items recommended. This application has been developed using Django and runs on the same machine as Cassandra. We can see a list of 477 users in the first page in the figure 4.6a. For each user we can see the details of the recommendation by clicking on its image. In the page showed in figure 4.6b we can see the list of recommended items as well as the best shelves of the users. The best shelves are retrieved by calculating the most frequent ones in the train-set. With this information, we can see if the shelves of the items recommended reflect the real best shelves of the user. In the test chapter in section 5.4.3 we explain more in details the information showed in this page.



(a) Home (b) Recommendation detail

Figure 4.6: Web UI of the recommended items for 477 users

4.7 Conclusion

In this chapter, we cover the implementation phase of our project. In particular, we explain how we configured Spark and Cassandra cluster and the principals step of our two algorithms: FWUM and Trust Walkers. For the configuration of Spark we also discuss some problem related to providing the right dependencies to the Spark cluster without incurring in duplicate dependencies and errors. We also talked about the problems encountered during the implementation of the FWUM and the Trust Walkers algorithms. In particular, we have observed an huge consumption in term of memory. We solve this problem by avoiding to accumulate results on list but instead spread the results as a key value RDD to take advantage of the distribution of this structure.

Several optimizations are also be made, especially in term of memory, database size, computation time, and scalability. Concerning the memory and the database size we had a massive disk consumption by the string representing the categories of the items. As we had around 50 Million of strings, we had a massive memory consumption on disk and in memory. We resolve this by representing each category with a hash value representing the

category with a very little duplication rate (more information in section 4.4.6.2). With this method, we reduce the disk space consumption by a factor 100 which was reflected in good improvements also in memory during the analysis with Spark. We also optimize the computation time when performing the test with the FWUM algorithm. In the first version, we see an exponential increase in the execution time when augmenting the number of users. This was because the matrix multiplication implemented in the Mllib library don't take into account when a matrix is very sparse, as in our case. This was resolved after implementing it ourselves (more information on the section 4.4.6).

Tests

To evaluate the FWUM algorithm, we used the Goodreads and the Epinions datasets. We took 75% of the ratings made by users as a train set and we hide the remaining 25% for the evaluation phase. We expect that the items recommended by the FWUM algorithm are most likely to be the same as the items contained in the test set. In particular, in section 5, we discuss the results of the FWUM algorithm by using two datasets. The Epinions.com dataset and the Goodreads dataset[1].

5.1 Metrics and visualization of the results

For measuring the quality of the recommendation we used the metrics suggested in [19] and also used in [1, cp. 5]. Specifically, we used the *Precision* and the *Recall*. These measures are calculated by comparing the test set against the recommended ones.

To calculate the precision the recall we build the confusion matrix (table 5.1) by counting the True Positive (TP), False Positive (FP), False Negative (FN),True Negative (TN) of the recommendation.

	Recommended	**Not recommended**
Preferred	True-Positive (TP)	False-Negative(FN)
Not preferred	False-Positive (FP)	True-Negative(TN)

Table 5.1: Confusion matrix of the recommendation results [19]

The evaluation metric are then calculated with the following formulas:

$Precision = \frac{tp}{tp+fp}$

$Recall = \frac{tp}{tp+fn}$

The *precision* measure is calculated by the percentage of the correct recommended items among the total number of the recommend ones. This metric is useful to knowing if the relevant results appears often in the recommendation. If the precision is high this means that most of the recommended items are relevant and the user don't need to search a lot to find one. This measure is very useful but does not tell anything about to total number of relevant items. If, for example, we have 1000 relevant items in the dataset and we recommend to the user only 2, the precision has a value of 100% but actually the recommendation quality is poor. For this reason we need the recall metrics that take into account the total number of relevant items. More specifically it measure the percentage of

[9]Epinions.com dataset, `http://liris.cnrs.fr/red/`

[1]Goodreads.com dataset, `http://goodreads.com`

the relevant items among the total number of the relevant ones contained in the dataset. In our precedent example we would have a recall value of 0.002.

In the context of this project, we give more importance at *Precision* because users expect that most of the recommended items are relevant items. They also require that the set of items recommended is relatively small because they do not have time to look at a big set of items.

Since the results are calculated for every user, it is necessary to group them for readability. Thus, we calculate the average, and we group users by counting how many user recommendations have the results in a specified interval. For example, we count the number of users in which their recommendation have a precision between 5% and 10%.

It is important to note that this evaluation method is very approximative as the real pertinence of the results have to be evaluated by the users themselves. However, it gives us a measure to comparing different algorithms, and it's useful to give us a general idea of the RS performance.

5.2 Parameters optimization

To optimize the results of the recommendation we have to choose the best values for each of the algorithm parameters. To adjust the parameters we made several experiments by changing their values and measuring the resulting *Precision* and *Recall*. With these measures, we can discover the value of the parameters that give the best results.

Thus, we have performed recommendation with the same dataset by changing the *Number of similar users* (k), the *Number of recomended items* (N) and the *Percentage of shelves* (S).

In figure 5.1 we can see that at the beginning the precision increase by augmenting k. After that k is near 40 the precision tend to be stable around the 9%. We choose k=40 as there is no significant amelioration in the quality of the recommendation. By choosing a small value of k we can also improve the duration of the computation.

In figure 5.2 we perform the same test, but we fix k to 40, and we change N. We can see that by increasing the N we have a linear increment of the Recall and a decrease of the precision. These results are also observed in [1, cp. 5.3] and can be easily explained. In fact, if the set of recommended items is big, we have more chance to find relevant items so the *Recall* measure is greater. On the contrary, the percentage of relevant items among the whole set of the recommended ones (called *Precision*) is smaller because there are a lot of not relevant items.

For the context of this project, we give more importance at *Precision* as explained before. Thus, we choose N to 20 as an acceptable compromise to have a good *Precision* and *Recall* by maintaining a limited number of items.

In figure 5.3 we evaluate the results for different values of S. The value of S represent the percentage of the most rated shelves (tag categories) that below to an item. We can see from the graph showed in figure 5.3 that if we take all the shelves, also these with few ratings, the recommendation has insufficient precision and recall. On the other hand, we have the best results by taking the 10% of the most rated shelves. This shows that some

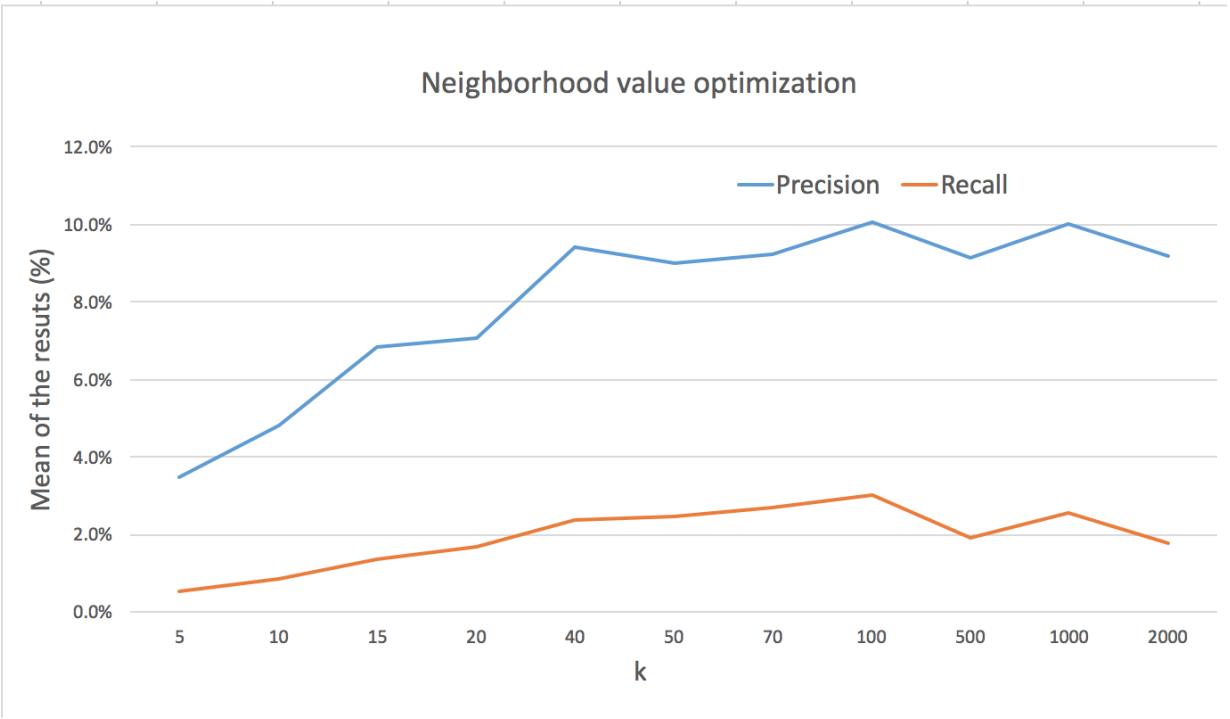


Figure 5.1: Performance measurement by changing the size of the neighborhood

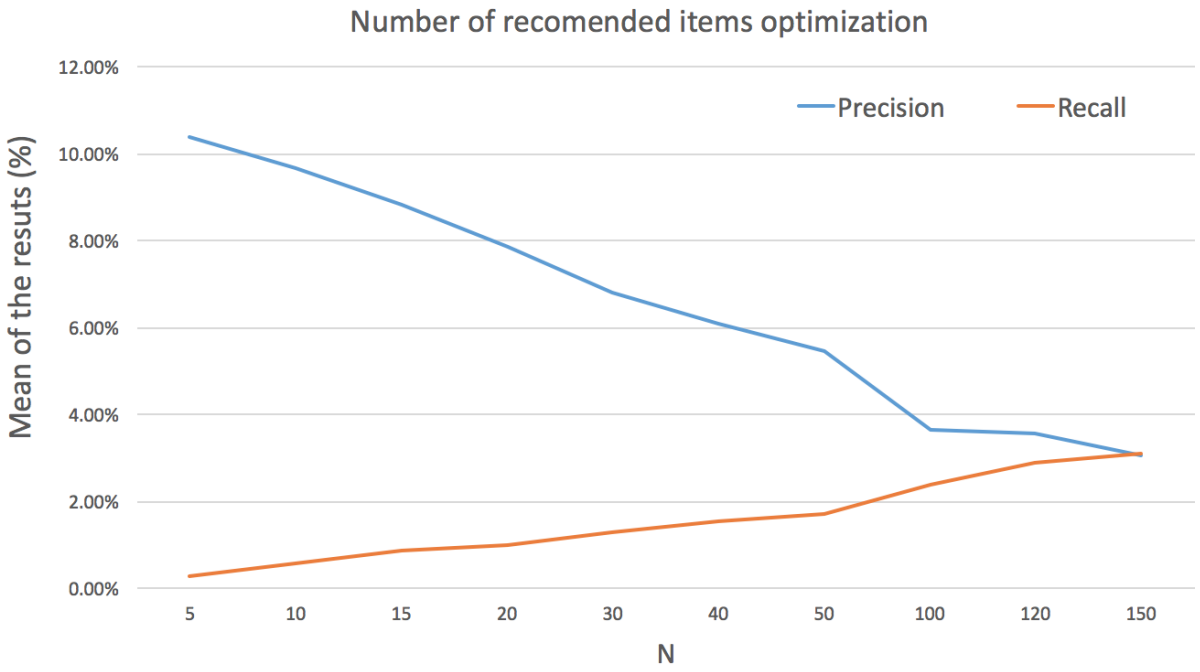


Figure 5.2: Performance measurement by changing the number of the recommended items

shelves are not very characteristic of the items. By choosing only the most voted ones, the recommendation quality increase. With these considerations, we choose the S=10%.

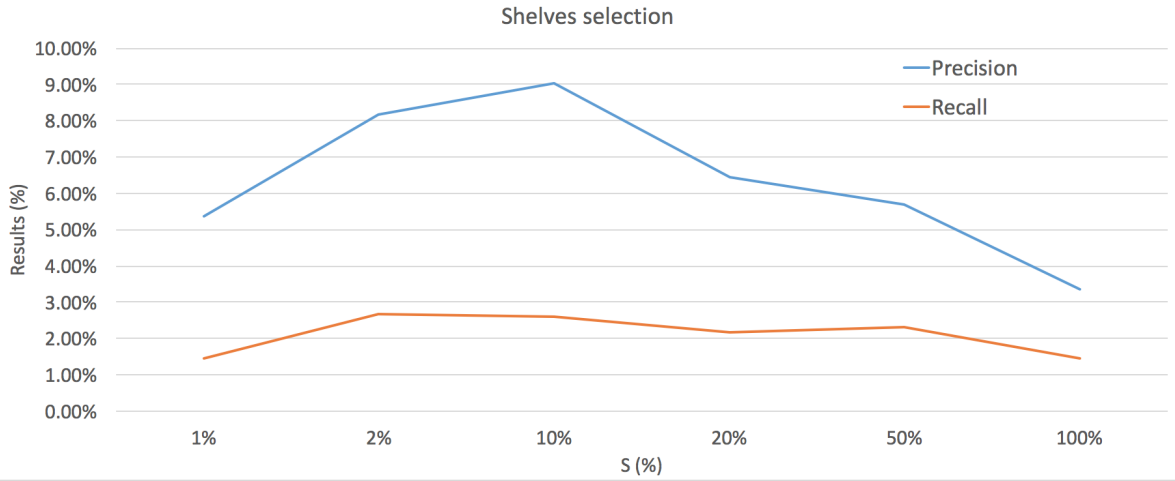


Figure 5.3: Performance measurement by changing the percentage of shelves

5.3 Scalability test

We performed a scalability test for the HRS by computing the recommendation to 477 up to 5276 users. We can see for the graph in Figure 5.4 that the performance of the algorithm is almost linear for up to 3000 users. We observe, then, an increment in the duration time when performing the recommendation for 5276 users. The difference between the measured time and the linear reference for 5276 users is around 8.6 minutes. We can explain this non-linearity by regarding the computation of the similarity between the users. In fact, the complexity of this operation is in $O(n^2)$. For a small number of users this value is negligible but for a big number of users we can note the difference.

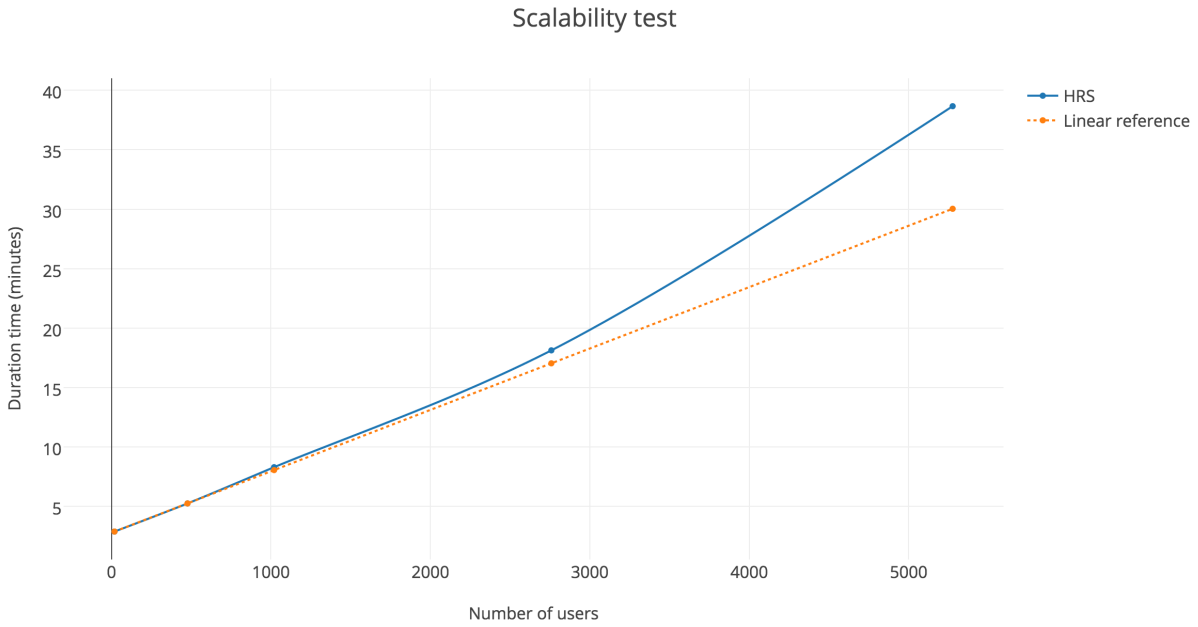


Figure 5.4: Measure of the duration time of the HRS by changing the number of users

5.4 Test with Goodreads dataset

5.4.1 Test description

The test on the Goodreads dataset has been made using 5276 users, ∼1 Million reviews and ∼200'000 items from the Goodreads datasets. To performing the analysis, ti has been setting up a Spark Cluster using the EC2 service of Amazon AWS as described in section 4.2 In table 5.2 are listed the algorithm parameters as well as the configuration of the Spark cluster. We use the r3.xlarge EC2, which provides up to 30 GiB in memory and 13 EC2 Compute Unit (ECU). These instances are one of the most optimized in term of cost per GiB of memory.

Algorithm parameters		**Cluster parameters**		**Versions**	
Number of similar users (k)	40	Region	eu-west-1	Spark version	1.5.2
Number of recomended items(N)	20	Avalaibility zoneone	eu-west-1a	Scala version	2.10
Minimal users similarity	0.2	Number of slaves	10	Cassandra version	1.5.0
Shelves votes percentage	10%	Instance-type	r3.xlarge	CSD version	1.5.0

Table 5.2: Algorithm parameters

5.4.2 Obtained results

From the results obtained, we calculate the average of the most useful measures. We can see these values on table 5.3.

In the graph showed in figure 5.5 we can see in the X axis the percentage of users where their recommendation has a precision and recall in a particular interval. In blue is represented the precision and the red the recall.

For example, we can see that the 10.7% of users (the fourth group of columns from left) have a Precision between 10 and 15% and only the 4.4% have precision between 35-40%.

From the table 5.4 we can see in details the value of the percentage of users in given interval of measures. From this table, we calculated that the 45.4% of users have a precision greater than 10% and the 7% of users have a Precision bigger that 35%.

In the figure 5.6 and in figure 5.7 are showed an example of recommendation of Michele and Megan. In this image, we can see the "best shelves" of their profiles and the list of recommended items. In green are represented the shelves of the recommended items that match the best shelves of the user profile.

Table 5.3: Mean of all the recommendations

Time of the computation	38 minutes
Precision mean	12,83%
Recall mean	3, 36%

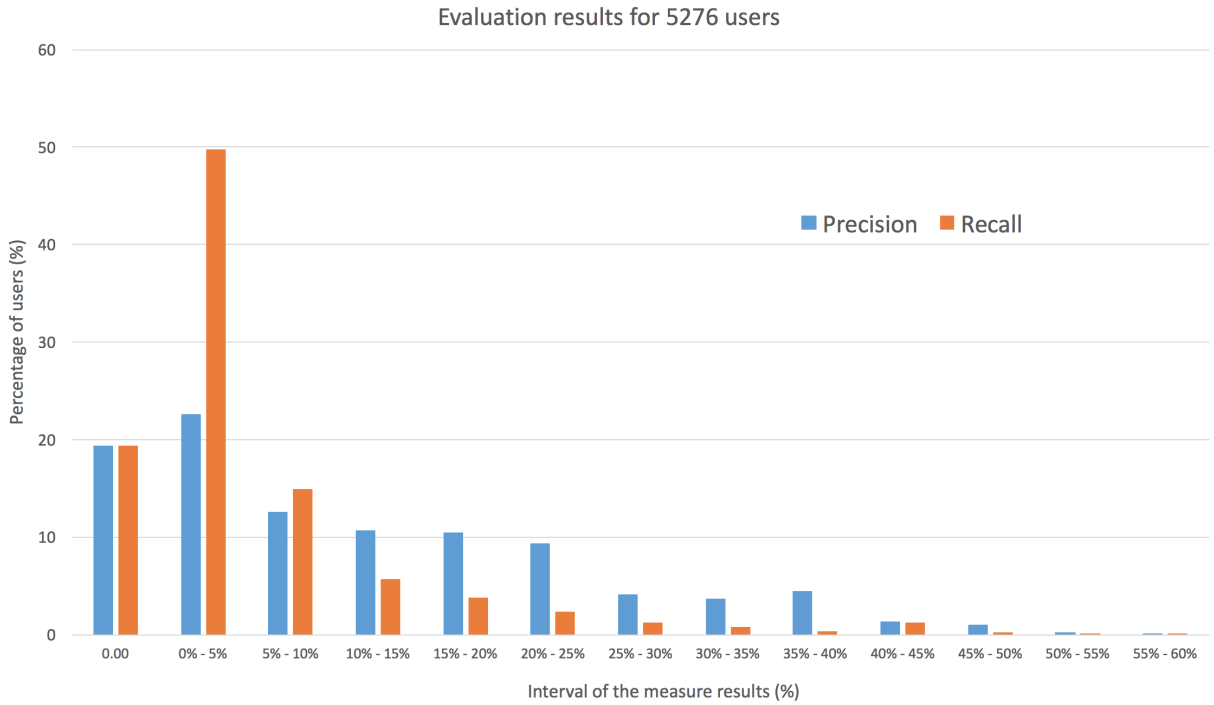


Figure 5.5: Graph of the percentage of users having the precision and the recall in a given interval.

Table 5.4: Percentage of users where the results is in a given interval of percentage. We can see the results for the precision and the recall metrics.

Interval of results(%)	*Perc. users percision(%)*	*Perc. users recall(%)*
0.00	19.32114883	19.32114883
0% - 5%	22.6054017	49.75223033
5% - 10%	12.53263708	14.9281501
10% - 15%	10.70496084	5.72434448
15% - 20%	10.44386423	3.82667624
20% - 25%	9.399477807	2.383389817
25% - 30%	4.177545692	1.261097606
30% - 35%	3.65535248	0.83234456
35% - 40%	4.438642298	0.33333223
40% - 45%	1.305483029	1.261096606
45% - 50%	1.044386423	0.2003446
50% - 55%	0.261096606	0.103432345
55% - 60%	0.110003	0.07241243
60% - 65%	0	0
65% - 70%	0	0
70% - 75%	0	0
75% - 80%	0	0
80% - 85%	0	0
85% - 90%	0	0
90% - 95%	0	0
95% - 100%	0	0

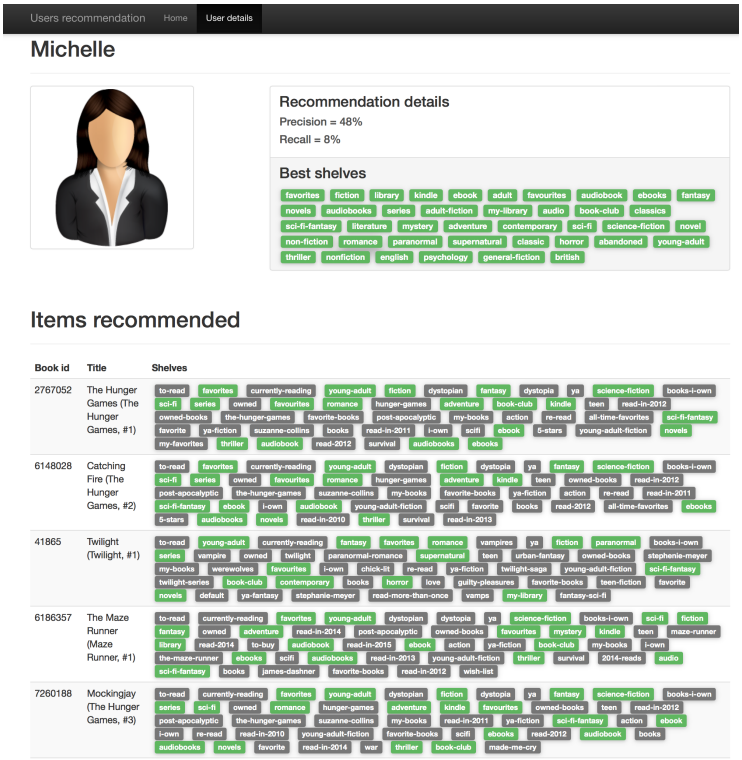


Figure 5.6: Exemple of the recommendation to Michelle

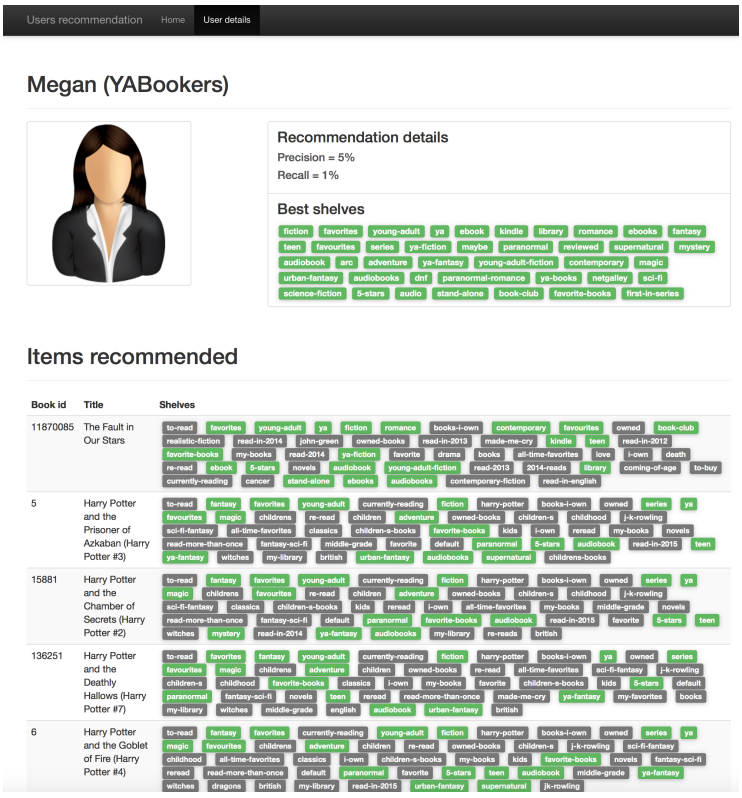


Figure 5.7: Exemple of the recommendation to Megan

5.4.3 Discussion of the results

The results of this test reveal that for almost the half of the users (∼45.2%) we can predict a good recommendation with a precision between 10% to 60% . For the remaining ones (almost 19%) the recommendation does not reflect their real ratings of the test items. Is important to note that this test is made by calculating in what percentage the previously rated items by the users are equals to the recommended ones (precision measure). In other words, we make the assumption that the recommended books are relevant only if they are exactly the same as the previously rated one. That means that if we have recommended some very similar books to the ones contained in the test set, they are categorized as totally irrelevant. For example, if we recommend "Harry Potter and the Chamber of Secrets" and there is "Harry Potter and the Philosopher's Stone" in the test set, the two ids don't match. Therefore, this book is categorized as irrelevant even if it is very similar the one rated in the test set. With these considerations, we do not know if a precision of 5% actually provide, relevant items or not, we can only say that the recommendation quality seems to be lower than other users that have a precision of 40-50%. That been said, we can compare these results with another presented in several articles that use the same metrics to estimating the quality of their recommendation. In [1] they perform several tests using the same algorithm but using a different dataset. Even if we can't compare it in an exact way, we can see how our results differs from them. In [1] they obtain their best results with a precision around the 60% that is similar to our maximal value of precision that is between 55% and 60%.
In figure 5.6 we showed a recommendation made to the "Michelle" that have a precision of 48%. We also calculated the shelves (or categories) that best describes its profile by counting the feature frequency in its ratings. In the "Items recommended" section we can see the recommended items details including their shelves. From this visualization, we can see very quickly the shelves that match with the user best shelves (in green). From the example of Michelle, we can see that a lots of shelves, related to the items recommended, matches with its preferred ones. This mean that the books recommended are very similar to the previously rated ones as they have the same categories. We can also see from the title that all these books are very similar to each other, so they are all potentially relevant (Hunger Games 1,2,3). The same thing can be view on the recommendation of "Megan" in figure 5.7 even if she have a less percentage of precision.
In conclusion, we can say that this technique performs quite similarly to the results observed in the reference article [1] and, even if a percentage of users have limited performance in term of precision, we can see that the recommendation is nevertheless coherent. To enhance the recommendation quality we can increase the number of users in the dataset. In fact, the more are the users in the dataset more high are the probability to find similar users and to recommend more interesting items.

5.5 Test on Epinions dataset

5.5.1 Test description

In this test has been made using 1000 users, 200' 000 reviews and 124'546 items from Epinions.com. For the computation ti has been set-up a Spark Cluster using the EC2 service of Amazon AWS [2].

The cluster configuration has the following parameter:

Table 5.5: Test parameters

Algorithm parameters		**Cluster parameters**		**Versions**	
Number of similar users	20	Region	eu-west-1	Spark version	1.5.2
Number of recomended items	100	Avalaibility zoneone	eu-west-1a	Scala version	2.10
Minimal users similarity	0.2	Number of slaves	5	-	-
Minimal shelves votes	10	Instance-type	m4.large	-	-

5.5.2 Obtained results

Average of the evaluation measures on users recommendation:

- Time of the computation = ∼16 minutes
- Precision mean = 4.5%
- Recall = 4.9%

Table 5.6 shows the percentage of users where their results have a percentage in a given interval.

Results value (%)	**Perc. of users precision(%)**	**Perc. of users recall(%)**
0.0 - 5	97.14255091103966	97.14255091103966
5 - 10	2.752449088960343	2.7002679528403001
10 - 20	0.15	0.05
15 - 20	0.0	0.0
20 - 30	0.0	0.10718113612004287
30 - 40	0.0	0.0
50 - 60	0.0	0.0

Table 5.6: Table of the percentage of users in given interval of measures

[2] Amazon AWS, `https://aws.amazon.com/it/`

5.5.3 Discussion of the results

As we can see from the averages measures and the table 5.6, the results from the Epinions.com dataset are less positive than the results of the Goodreads dataset discussed in section 5.4.3. We can see that only the 2% of users have a precision and recall between 5% and 10%.

Our hypothesis for this difference in the results of the two datasets is the quality of the categories of items. The Goodreads dataset provides at least 50 features per items instead of only one feature per items in the Epinions dataset. Also, in Goodreads.com the categories are very detailed and accurate as they are created directly by the community. The less quality of this information can influence the quality of the recommendation negatively since the FWUM algorithm uses the features as the principal factor to describing the users profile. Also, the dataset of Epinions.com provides all types of products that make it more challenging to perform a good recommendation. This hypothesis is not already proven in this test. To prove it, we have to carry out the test with more type of datasets in order to provides more reliable results.

5.6 Conclusion

In this chapter we spoke about the several tests made on the Goodreads dataset and the Epinions dataset using the FWUM algorithm. Firstly, we describe the metrics used to evaluating the performance of the RS. In particular, we use the precision and the recall. Then, we perform several measures to optimize every parameter of the algorithm (section 5.2). Then, we perform the tests by using the best parameter values. The results of the tests are showed in two way. In the first way, we calculate the precision and the recall for every user, and we summarize these measure in a graph and by calculating the global mean. The results shows that the average precision of all the recommendations is 12.83%. With the graph generated we were able to observe that the 45% of the total users ($\sim$2300 users) have a precision bigger that 10% and a maximal value of 60%. We can also see that the 19% users have a precision near to zero. This means that for this part of users there are no similar users in our dataset, and the recommendation can't be very accurate for this reason. To increase the recommendation quality, one possibility is to augmenting the number of users in the dataset to increase the probability to find similar users and recommend more accurate items. Another possibility is to cluster similar items to reducing the sparsity of the utility matrix and decreasing the effect of the cold start problem. We also compare the results with the reference article [1] where their best results have a precision near the 60%. Even if they use another type of dataset we can say that the results are quite similar as our best results is observed at 55%-60% of precision. However, we don't know their average results.

By testing our dataset of the Epinions dataset we obtain more limited results. The hypothesis for explain this result is that compared the Goodreads dataset, it have only a few categories (500) on the items and are very general. In opposite, the Goodreads dataset has more than 27000 unique categories, and they are more precise as introduced and confirmed by the community.

We also observed that the value of the precision is not a perfect information to measuring

the quality of the recommendation. In fact, by comparing two users with a significant difference in term of precision, we were able to see that the recommended items are coherent to the best categories rated previously by the user.

In conclusion, the FWUM algorithm performs very good for at least the 45% of the users. The recommendation quality for the remaining users can be enhanced by augmenting the number of users in the dataset.

Conclusion

The objectives of this thesis were to analyze the state of the art of several techniques used for constructing a Recommender System. Then, experiment one of the recommendation approaches for testing their performances with some standard metrics. An objective was also to use some common big data technology for computing the analysis by taking advantage of the power of more machines in parallel.

In the beginning, we choose two datasets that provide information about the user ratings on products and, preferably, with some data concerning the social relation between them. In chapter 2.2 we analyze some of them, and we finally choose to retrieve our dataset from Goodreads.com and Epinions.com.

The Goodreads dataset provides information about books; the Epinions dataset [9] about the purchases of generic products.

In the state of the art we explored some of the most common recommend systems techniques such as CB and CF as well as the Hybrid approach. In addition, we study severals techniques used for constructing a SRS by using social network information.

We found that the Hybrid approach described in [1] is the one that better take advantages of the information contained in our dataset and performs excellent results compared to CF and CB. For this reason, we choose to implement it and test his performance.

We also wanted to implement a second algorithm that takes advantages of the social relations between the users. In particular, we decided to use the "TrustWalkers" algorithm described in [2].

To choose the right and the more appropriate technology for implementing our algorithms, we made a short overview of them. In particular, we made a comparison between Spark and Hadoop by regarding at some key points such as speed and ease of use. Our analysis shows (chapter 2.4) that Spark provide a better speed thanks to their optimized use of the memory [1] and it's more appropriate to implement a complex algorithm thanks to the RDD abstraction and their lambda function that provides lots of built-in optimized features.

We decided then to use a distributed database because we didn't know, in the beginning, the final size of the dataset. Also, was an excellent opportunity to explore a quite new technology that is increasingly used by a lot of companies in these times. We choose Cassandra as distributed database after we made a shot overview of the principal characteristics of the other ones, such as MongoDB, HBase, Apache Hive and Amazon Dynamo (more information on section 2.3). Since most of these databases provide more than enough features for our use case, the choice has been made by regarding three principals aspects: the facility to set-up, the tools and APIs available and the quality of the documentation. We found that Cassandra is easy to install and to set up and provides an excellent integration with

[9] Epinions.com dataset, `http://liris.cnrs.fr/red/`

[1] Spark Official, `http://spark.apache.org`

Spark. We also found that their documentation is very clear, and have a great community on StackOverflow.com.

The implementation phase (chapter 4) has gone through several steps: (1) the implementation of the Python API for downloading the Goodreads dataset (2) the configuration of Spark and Cassandra clusters and (3) the implementation of the Hybrid Recommender System and the TrustWalkers algorithms.

For every algorithm, it has been taken a considerable amount of time for optimizing every aspect of the algorithm including memory consumption and computational time. In fact, we were able to reduce the Goodreads dataset from 25GB down to 250MB (factor 100) without loose relevant information. We're also able to reduce the time duration from 20 minutes to 5.2 minutes, for a small dataset, and from 3h to 8.3 minutes for a larger dataset (see section 4.4.6). From the scalability test performed in section 5.3 we were able to observe that the final version of the Hybrid algorithm has a complexity linear for up to 3000 users and we observe a an increment on the duration time for 5200 users. This is due by the computation of the User-User similarity matrix that is performed in $O(n^2)$. For a small number of users this time is negligible but for a bigger number of them we can note the difference.

The functioning of our Recommender System is pretty straightforward. Firstly we have downloaded the dataset from Goodreads.com into the Cassandra database. Spark then read this database, containing the users ratings information and analyze it with several EC2 [2] on-demand instances in the cluster. The results of the computation are then re-saved into the Cassandra database or on S3 for further analysis on the results data (see section 3.1).

In the final part, we performed several tests of the FWUM algorithm using both the datasets and by using some of the most common metrics for evaluating the quality of a Recommender System such as Precision and Recall.

The results showed that for the 45% of the users the recommendation has a precision between 10% and 60% by using the Goodreads dataset. We observed, instead, more limited performance when tested using the Epinions.com dataset.

Our hypothesis for explaining this difference is that the Goodreads dataset provides more and better categories on books (over than 27'000 detailed categories) compared to Epinions.com dataset which contains only a total of 500 generic types. This hypothesis has not been proven yet in the context of this project.

Concerning the Trust Walkers algorithm, the test has been made only with a dataset build by hand for tasting their functioning. In future will be possible to optimize it and merge it with the Hybrid algorithm for increase the recommendation quality (see section 3.4).

By regarding the initial purpose, we can see that the main aims are widely achieved. The research of the RS techniques has been made, and a Recommender System has been implemented and tested. A little web application for demonstrating the results has been also established. The last objective, the one concerning the RecSys challenge 2016 [3], will be an excellent continuation of this thesis as it begins just after the deadline for this project. Maybe there is a possibility of doing this on a stage in the HE-ARC school.

[2] Amazon EC2,`https://aws.amazon.com/fr/ec2/`

[3] `http://2016.recsyschallenge.com`

6.0.1 Issues and difficulties

During the whole project we observe the following main issues:

- Dataset selection: We had some difficulties to find a public dataset that provides social information about the users (see section 3.1.1). In fact, most of the dataset available don't publish this information also for privacy purpose.
- Database downloading: We had some issues to retrieve and download the Goodreads dataset from the Goodreads API [4]. In fact, we observe a lot of "exceptional cases" that took a lot of time to treat. In this case, we have underestimated the difficulty of this task (more info in section 4.3).
- Compilation: Some problems have also been found in the configuration of the dependencies. In fact, even if we need some dependency at the compiling time in the client we don't need to send them to the Spark master as they are already present in the standard Spark library. A duplication of these dependencies causes a crash at runtime. To avoiding this issue we found the assembly plugin[5] for Scala that allows to specify an "included" option for use some dependencies at the compiling time but not in the final compiled jar (more info in section 4.4.7.3).
- Memory consumption: In the implementation phase we observe a heavy use of memory. The first problem was the utilization of the *aggregateByKey()* operation as we collected too many elements in a local list. This gives an out of memory error. We solve this by changing the *aggregateByKey()* to the *reduce()* function instead. The second problem was the size of the data. In fact, we had a massive memory consumption caused by the strings representing the categories of the books. As we had around 25 Million of strings, we had a huge memory consumption on disk and in memory. We resolve this by replacing each category with a hash value representing the category. By doing this we observe a very little duplication rate (more information in section 4.4.7.2) but remaining negligible . With this method, we reduce the memory consumption by a factor 100.
- Scalability: In the tests performed with the first version of the FWUM algorithm we see an exponential increase in the execution time when augmenting the number of users. The cause was that we didn't take into account that the matrix multiplication implemented in the Mllib library don't take into account when a matrix is very sparse, as in our case. This was resolved after implementing it ourselves with the basic functions of Spark (more information on section 4.4.6).

[4] Gooreads API,
[5] SBT assembly, `https://github.com/sbt/sbt-assembly`

6.0.2 Future enanchements

There are lots of possible entrenchments on this project, and the principals are the following:

- The results obtained in the test chapter have been compared with the results of the reference paper [1]. A better evaluation would be to compare our algorithm with others articles results by using some standard datasets. This will give more comparable results.
- To improve the performance of the FWUM algorithm is recommended to retrieve more users in the Goodreads dataset. In fact, with a higher number of users, there is more probability of finding similar users. Another possibility is to cluster the similar items to reducing the sparsity of the utility matrix and decreasing the effect of the cold start problem.
- An optimization of the TrustWalkers algorithm will also be very interesting to discover their results on the Epinions and Goodreads dataset. Actually, it has some problems related to the time complexity of the algorithm (see section 4.5.1).
- Then, a combination with the Hybrid algorithm and the Trust Walkers algorithm will be very interesting for try to improve the recommendation quality. A possibility would be to run the Trust Walkers algorithm on the results of the FWUM Algorithm and filter only the best results. Or merge the two results and weighting the contribution of the Trust Walkers and the FWUM algorithm by the users themselves.
- An improvement in the test strategy will also be interesting. In our case, we divide our dataset in train and test set. A more precise evaluation can be to use the "leave-one-out" method where only the data of one user will be hidden and the recommendation of every user will be evaluated singularly using the whole dataset. Then an average of every user results give the final results of the RS.

6.0.3 Personal conclusion

This project is the result of many years of study and works in my Bachelor and Master. I used notions in Information Retrieval, Data Management, Web Mining and Big Data Analytics. In this project, I learned a new domain related to the world of the Recommender Systems. It's a vast domain, and it was very interesting to discover some of the techniques used. I also learned how to use Spark for distributing the computation on several machines, and I learned some concept of Scala. I found Scala and Spark a very powerful combination for a cloud framework.

I also had the possibility to work with Cassandra database, and I found it very interesting and powerful distributed database. In general, I'm very glad of the work that was done and the possibility of working in a domain that I'm very interested in. Was a challenging project for lots of aspects but I'm satisfied with the obtained results. Of course, a lot of work can still be done for improving the results of this project.

I want to thank all my professors, Gorbel Hatem and Punceva Magdalena, as well as my expert, David Jacot, for their support and expertise during the whole project.

Declaration of honor

I declare that the content of this thesis is completely my own work. No parts of this assignment were taken from other people's work without giving them credit. All references have been clearly cited. [1]

Fribourg, 4.02.2016

Simone Cogno

[1] Declaration of honor, http://www.ku.de/fileadmin/160115/Master-__Seminar-_ _Bachelorarbeiten/Formale_Richlinien_LFB_English.pdf

Acronyms

CB Content-based ii, 2, 6–9, 15, 54

CF Collaborative filtering ii, 2, 6–9, 15, 54

EC2 Elastic Compute Cloud 28, 47, 51, 74, 77

ECU EC2 Compute Unit 47, 76

FF Feature Frequency 23, 34

FN False Negative 43

FP False Positive 43

FWUM Featured-Weighted User Model ii, iv, v, 8, 24, 27, 28, 33, 38, 39, 41, 43, 52, 55, 57

HRS Hybrid Recommender Systems 3, 46, 64

IF Information Retrieval 23

IUF Inverse User Frequency 7, 23, 34

PCC Pearson Correlation Coefficient 9, 25, 26

RDD Resilient Distributed Dataset 15, 33

RS Recommender Systems ii, iv, 2–5, 7, 10, 11, 13, 44

RW Random Walk 25, 26, 39, 40

RWs Random Walks 9, 25

SRS Social Recommender System iv, 8, 54

TF Term Frequency 23

TFIDF Term Frequency Inverse Document Frequency 7, 23

TN True Negative 43

TP True Positive 43

TW Trust Walkers iv, 16, 27, 28

UDT User Defined Type 20

VSS Vector Space Similarity 9, 25

Bibliography

[1] P. Symeonidis, A. Nanopoulos, and Y. Manolopoulos, "Feature-weighted user model for recommender systems", *Conference: User Modeling 2007 and 11th International Conference*, 2007. DOI: 10.1007/978-3-540-73078-1_13. [Online]. Available: http://www.researchgate.net/publication/221261040_Feature-Weighted_User_Model_for_Recommender_Systems.

[2] M. Jamali and M. Ester, "Trustwalker: A random walk model for combining trust- based and item-based recommendation", *KDD*, 2009.

[3] J. Leskovec, A. Rajaraman, and J. Ullman, *Mining of Massive Datasets*. 2005, ch. 9, pp. 73–105.

[4] P. Lops, M. de Gemmis, and G. Semeraro, *Recommender Systems Handbook, Chapter 3*. 2005, pp. 73–105. DOI: 10.1007/978-0-387-85820-3_3. [Online]. Available: http://google.ch.

[5] J. O'donovan and B. Smyth, "Trust in recommender systems", *In: International Conference on Intelligent user Interfaces*, 167–174, 2005.

[6] W. C.-N. M.L., Hernandez-Alcaraz, R. Valencia-Garcıa, and F. G. Sanchez, "Social knowledge-based recommender system, application to the movies domain", *Expert Systems with Applications*, vol. 39, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417412004952.

[7] H. Ma, D. Zhou, C. Liu, M. R. Lyu, and I. King, "Recommender systems with social regularization", *Journaltitle*, 2011.

[8] A. Olteanu, A.-M. Kermarrec, and K. Aberer, "Comparing the predictive capability of social and interest affinity for recommendations", Lecture Notes in Computer Science, B. B. boualem@cse.unsw.edu.au, A. B. best@cs.bu.edu, Y. M. manolopo@csd.auth.gr, A. V. avakali@csd.auth.gr, and Y. Z. yanchun.zhang@vu.edu.a Eds., pp. 276–292, 2014. DOI: 10.1007/978-3-319-11749-2_22. [Online]. Available: http://link.springer.com/chapter/10.1007%2F978-3-319-11749-2_22.

[9] L. Backstrom and J. Leskovec, "Supervised random walks: Predicting and recom- mending links in social networks", *WSDM*, 2011.

[10] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks", *Journal of the American society for information science and technology*, 2007.

[11] S.-H. Yang, B. Long, N. S. A. Smola, Z. Zheng, and H. Zha, "Like like alike: Joint friendship and interest propagation in social networks", *WWW*, 2011.

[12] F. Fouss, A. Pirotte, J.-M. Renders, and M. Saerens, "Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation", *IEEE Trans. on Knowl. and Data Eng.*, 2007.

[13] S. Milgram, "The small world problem", *Psychology Today*, vol. 2, 1967.

[14] X. Liu and K. Aberer, "Soco: A social network aided context-aware recommender system", *WWW '13 Proceedings of the 22nd international conference on World Wide Web*, pp. 781–902, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2488457.

[15] M. Gupte and T. E. Rad, "Measuring tie strength in implicit social networks", *CoRR*, vol. abs/1112.2774, 2011. [Online]. Available: http://arxiv.org/abs/1112.2774.

[16] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey", *Knowledge-Based Systems*, 2012.

[17] S. V. Sandoval, "Novelty and diversity enhancement and evaluation in recommender systems", Master final work, Universidad Autonoma de Madrid, 2012, ch. 1, p. 1. [Online]. Available: http://www.mavir.net/docs/tfm-vargas-sandoval.pdf.

[18] W. Carrer-Neto, M. H. Alcaraz, R. V. Garcıa, and F. G. Sanchez, "A peer-to-peer recommender system base don spontaneous affinities", *ACM Transactions on Internet Technology*, vol. 9, 1–34, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1462159.1462163.

[19] A. Gunawardana and G. Shani, "A survey of accuracy evaluation metrics of recommendation tasks", *Journal of Machine Learning Research*, vol. 10, pp. 2935–2962, 2009.

[20] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark, Lightning-Fast Big Data Analysis*. O'Reilly Media, 2015, pp. 73–105, ISBN: 978-1-4493-5862-4. [Online]. Available: http://shop.oreilly.com/product/0636920028512.do.

[21] S. Ryza, U. Laserson, S. Owen, and J. Wills, *Advanced Analytics with Spark*. O'Reilly Media, 2015, pp. 73–105, ISBN: 978-1-4919-1276-8. [Online]. Available: http://shop.oreilly.com/product/0636920035091.do?green=BE0C56C4-8FAE-56EC-01DB-060FF498F01D&intcmp=af-mybuy-0636920035091.IP.

[22] J. Cho, K. Kwon, Y. Park, and Q-rater, "A collaborative reputation system based on source credibility theory", *Expert Systems with Applications*, vol. 36, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417408001486.

[23] W. Yuan, D. Guan, Y. Lee, S. Lee, and S. Hur, "Improved trust-aware recommender system using small-worldness of trust networks", *Knowledge Based Systems*, vol. 23, 232–238, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095070511000002X.

List of Figures

3.1 Architecture of the system 18
3.2 First version of the database shema of goodreads dataset 20
3.3 Second version of the database shema of goodreads dataset 21
3.5 Initial matrices[1] 23
3.6 User profile and user similarity matrices after the feature weighting step[1] 24
3.7 Top-N items list generation process 25
3.8 Representation d'un social network or a trust network [2, p. 2] 26

4.1 Spark UI 31
4.2 DAG representation 32
4.3 Events-timeline 32
4.4 General steps for the hybrid algorithm construction 34
4.5 Comparing the performance of the algorithm by using the Block Matrix multiplication and without 38
4.6 Web UI of the recommended items for 477 users 42

5.1 Performance measurement by changing the size of the neighborhood . 47
5.2 Performance measurement by changing the number of the recommended items 47
5.3 Performance measurement by changing the percentage of shelves . . . 48
5.4 Measure of the duration time of the HRS by changing the number of users 48
5.5 Graph of the percentage of users having the precision and the recall in a given interval. 50
5.6 Exemple of the recommendation to Michelle 51
5.7 Exemple of the recommendation to Megan 51

List of Tables

2.1 Pros and cons of the Content-Based RS 5
2.2 Pros and cons of the common CF approach 7
2.3 Comparative table of datasets 12

4.1 Confusion matrix of the recommendation results [19] 37

5.1 Confusion matrix of the recommendation results [19] 45
5.2 Algorithm parameters 49
5.3 Mean of all the recommendations 49
5.4 Percentage of users where the results is in a given interval of percentage. We can see the results for the precision and the recall metrics. . 50
5.5 Test parameters 53
5.6 Table of the percentage of users in given interval of measures 53

CD-ROM Content

In addition to this rapport, a CD-ROM is provided, containing the following documents and folders:

- README.txt ..... This description
- 10_administration ..... Files related to the administration
- 20_meeeting_minutes ..... All weekly and monthly minutes
- 30_RS_articles ..... Related articles
- 40_code_source ..... All useful code source
  - GoodreadsPythonAPI ..... Python API
  - HybridRS ..... Scala code for the Hybrid RS
  - OutilsScripts ..... Useful Spark scripts
  - TrustWalkers ..... Scala code of the Trust Walkers algorithm
  - webui ..... Web App. for views the results
- 50_rapport ..... Rapport and poster
- 60_other_resources ..... All others resources used during this project
  - epinions_dataset ..... Epinions dataset data
  - goodreads_dataset ..... Goodreads dataset data
  - Hybrid_RS_Results ..... Several results of the Hybrid RS

Recommanded reading

To better understand the technique used in this thesis I recommend to read the article "Feature-weighted User Model for Recommender Systems" [1] that are taken as main reference for this thesis.

I also recommend to read the Wikipedia page [1]of the Recommender System that explain very well the basic principles.

To understand the source code of this project as well as some of the problem discussed is recommended to read the book "Learning Spark"[20]. This book has been taken as reference for understanding the basics of Spark. In addition is recommended to read the book "Advanced Analytics with Spark "[21] for see some more advanced use of Spark.

[1]Wikipedia, Recommender Systems, `https://en.wikipedia.org/wiki/Recommender_system`

Source code repository

All the data concerning this project are available on the git repository at the address:

- GitLab HES-SO//FR [1]
- GitHub[2]
- Forge HES-SO//FR[3]

[1]GitLab HES-SO//FR, https://gitlab.forge.hefr.ch/simone.cogno/MT-Products-recommendation-system-using-the-social-media-networks

[2]GitHub, https://github.com/Simone-cogno/MT-Product-Recommender-System$_{i}mproved - with - social - network - information$

[3]Forge HES-SO//FR, https://redmine.forge.hefr.ch/projects/master-thesis-products-recommendation-system-using-the-social-media-networks

Planning

This is the final planning. Each column represents a week starting at date of 15th September 2015 and end at date of 5th February 2016.

Code samples

E.1 Word count with Hadoop

```
public class WordCountInproved extends Configured implements Tool{
	public final static IntWritable ONE = new IntWritable(1);
	private int numReducers;
	private Path inputPath;
	private Path outputPath;

	/**
	* WordCount Constructor.
	*
	* @param args
	*/
	public WordCountInproved(String[] args) {
		if (args.length != 3) {
			System.out.println("Usage: WordCount <num_reducers> <input_path>
				<output_path>");
			System.exit(0);
		}
		numReducers = Integer.parseInt(args[0]);
		inputPath = new Path(args[1]);
		outputPath = new Path(args[2]);

	}

	/**
	* Utility to split a line of text in words.
	*
	* @param text what we want to split
	* @return words in text as an Array of String
	*/
	public static String[] words(String text) {
		StringTokenizer st = new StringTokenizer(text);
		ArrayList<String> result = new ArrayList<String>();
		while (st.hasMoreTokens()){
			String wordString=st.nextToken();
			if(wordString.matches("[a-zA-Z]+"))
			result.add(wordString.toLowerCase());
		}
		return Arrays.copyOf(result.toArray(),result.size(),String[].class);
	}


	/**
	* Simple Mapper class for WordCount
	*
	* Input: (LongWritable id, Text line)
	* Output: (Text word, IntWritable 1)
	*
	* @author fatemeh.borran
	*
	*/
	static class WCMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
		private HashMap<String, Integer> h;
		private IntWritable count;
		private Text text;
		/**
		* The setup before map.
		*/
		@Override
		protected void setup(Context context) throws IOException,
		InterruptedException {
```

```
                super.setup(context);
                h=new HashMap<String, Integer>();
                count=new IntWritable();
                text=new Text();
        }

        /**
        * The map method reads an id as key and a text as value
        * and emits the pair (word,1) using Mapper.context.write()
        *
        */
        @Override
        protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
                for(String word: WordCountInproved.words(value.toString())){
                        if(h.containsKey(word))
                        h.put(word, h.get(word)+1);
                        else {
                                h.put(word, 1);
                        }
                }
        }

        /**
        * The cleanup after map.
        */
        @Override
        protected void cleanup(Context context) throws IOException,
        InterruptedException {

                super.cleanup(context);
                for(String word:h.keySet()){
                        //NO! context.write(new Text(word), new IntWritable(h.get(word)));
                        count.set(h.get(word));
                        text.set(word);
                        context.write(text, count);
                }
        }

}

/**
* Reducer class for WordCount sums results for a given word.
*
* Input: (Text word, IntWritable 1)
* Output: (Text word, IntWritable sum)
*
* @author fatemeh.borran
*
*/
static class WCReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        IntWritable sumInt;
        /**
        * The setup before reduce.
        */
        @Override
        protected void setup(Context context) throws IOException,
        InterruptedException {
                super.setup(context);
                sumInt=new IntWritable();
        }

        /**
        * The reduce method reads an id as key and an iterable collection of 1 as values
        * and emits the pair (word,sum) using Reducer.context.write()
        */
        @Override
        protected void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {

                int sum = 0;

                for (IntWritable value : values)
```

```
                sum += value.get();
                sumInt.set(sum);
                //NO!context.write(key,new IntWritable(sum));
                context.write(key, sumInt);
            }

            /**
            * The cleanup after reduce.
            */
            @Override
            protected void cleanup(Context context) throws IOException,
            InterruptedException {

                super.cleanup(context);
            }
        }

        /**
        * The main method to define the job and run the job.
        */
        @Override
        public int run(String[] args) throws Exception {

            Configuration conf = this.getConf();

            // Create a new Job
            Job job = new Job(conf,"Word Count Inproved");

            // Set job input format to Text:
            // Files are broken into lines.
            // Either linefeed or carriage-return are used to signal end of line.
            // Keys are the position in the file, and values are the line of text.
            job.setInputFormatClass(TextInputFormat.class);

            // Set map class and the map output key and value classes
            job.setMapperClass(WCMapper.class);
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(IntWritable.class);

            // Set reduce class and the reduce output key and value classes
            job.setReducerClass(WCReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

            //Set Combiner class
            job.setCombinerClass(WCReducer.class);

            // Set job output format to Text
            job.setOutputFormatClass(TextOutputFormat.class);

            // Add the input file as job input (from local or HDFS) to the variable inputPath
            FileInputFormat.addInputPath(job, inputPath);

            // Set the output path for the job results (to local or HDFS) to the variable outputPath
            FileOutputFormat.setOutputPath(job, outputPath);

            // Set the number of reducers using variable numReducers
            job.setNumReduceTasks(numReducers);

            // Set the jar class
            job.setJarByClass(WordCountInproved.class);

            // Execute the job
            return job.waitForCompletion(true) ? 0 : 1;
        }

        public static void main(String args[]) throws Exception {
            long time=System.currentTimeMillis();
            int res = ToolRunner.run(new Configuration(), new WordCountInproved(args), args);
            System.out.println("Execution time: "+(System.currentTimeMillis()-time)+" [ms]");
            System.exit(res);
        }
    }
```

Code implemented in the context of the Big Data Analytics course in the HES-SO school in Lausanne, Switzerland. [1]

[1]Cours of Big Data Analytics, Paratical work 1 (Word Count implementation with Hadoop)`http://mse-bda.s3-website-eu-west-1.amazonaws.com/labs/lab1.html`

Installation guide

F.1 Goodreads Python API

To use the Python API used to download the Goodreads datset you need the following python libraries:

```
sudo pip install cassandra-driver --upgrade
sudo pip install coloredlogs
sudo pip install oauth2
pip install requests
sudo pip install beautifulsoup4
sudo pip install -U textblob
sudo python -m textblob.download_corpora #sentiment analysis tool
```

If you have some error installing the cassandra-driver you can try to install the folowing libraries:

```
sudo apt-get install libffi-dev
sudo apt-get install libssl-dev
sudo apt-get install python-dev
```

F.2 Install and configure Cassandra

The official guide for installing Cassandra can found in the article "Installing a Cassandra cluster on Amazon EC2"[1].

By following this instruction we can launch a Cassandra cluster in three steps:

1. Launching the DataStax Community AMI
2. Creating an EC2 security group
3. Creating a key pair
4. Connecting to your DataStax Community EC2 instance
5. Clearing the data for an AMI restart
6. Expanding a Cassandra AMI cluster

As we can see the steps for launching a Cassandra node are similar to those used to launch a normal EC2 instance from AWS. For adding more node (point 6) we can use the OpsCenter fount at *http://INSTANCE_IP:8888* and use a graphical wizard to add and launch a new node, for this part all the configuration are made transparently to the administrator.

[1] Installing a Cassandra cluster on Amazon EC2, 2016, `http://docs.datastax.com/en/cassandra/2.1/cassandra/install/installAMI.html`

When the cluster is ready we can create our tables by initializing first our keyspace by the following CQL command:

```
calsh> CREATE KEYSPACE prs
WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
```

The "Simple Strategy" means that we will use only one datacenter and the "replication_factor=3" means that we have three replicas of every row placed on different nodes. With the Simple Strategy, the replicas are placed on the next node of the cluster clockwise. If we want to use multiple datacenters then, we can use the keyword "NetworkTopologyStrategy" that places the replicas on different datacenter for better fault tolerance (even if an entire region fail).

We can then the create a table by the following CQL command:

```
calsh>CREATE TABLE users (
id uuid PRIMARY KEY,
name varchar,
...
);
```

F.3 Install and configure Spark

Our machine have Java 1.7.x or later installed and Scala 2.10. We recommend to use the same version to make the code build and run without problems. Spark can be downloaded from the official spark page at *http://apache.spark.com* and it's sufficient to unpack the zip file in a folder of your choice. Then, set the environment environment variable to your path:

```
export PATH = $PATH:YOURFOLDER/spark/bin
```

Within the folder you can run a Spark application using the *./spark-submit* or using the console by run the *./spark-shell* program. You can also follow the tutorial at `http://www.tutorialspoint.com/apache_spark/apache_spark_installation.htm`

To configure a Spark cluster and launch an application see the next section.

F.3.1 Spark cluster set-up

To execute a Spark application on a EC2 Cluster we need to do the following steps:

1. Launch a Spark EC2 cluster
2. Connect to the Master driver
3. Build the source code in local and create a jar containing all the necessary dependencies
4. Copy the compiled JAR to the Master of the EC2 cluster

5. Configure the credentials on the Master to connecting on AWS S3
6. Run the spark application
7. Monitor the running application on the Spark UI

F.3.2 Launch a Spark EC2 cluster

To launching an EC2 Cluster we can use the spark-ec2 script contained in the Spark folder *$SPARK_HOME/spark1.5.1/ec2/*. From this folder, we can run the following command.

```
./spark-ec2 \
--key-pair=PRS-Simone \
--identity-file=PRS-Simone.pem \
--region=eu-west-1 \
--zone=eu-west-1a \
--slaves=6 \
--instance-type=m4.xlarge \
launch \
prs-spark-cluster
```

In the code below we have launched a EC2 cluster in Ireland in the availability zone Eu-west-1a. We can see that we have started a total of 6 machines where 1 is the Master driver and the remaining 5 are the Slaves. For our algorithm, we typically use the instance type m4.xlarge that has an equilibrate performance in the computation as well as in memory. This instance has, in fact, a computational performance of 13 ECU and 16 Gib of RAM memory. At this time, it cost $0.239 pro hours. We can then calculate the approximative cost of a job of 2 hours with this configuration we have to multiplicative this cost for every machine for every hour of computation.

F.3.3 Connect to the Master driver

When the EC2 Cluster is started successfully we can connect with SSH on it with the following command:

```
SPARK_HOME/spark1.5.1/ec2/spark-ec2 \
--key-pair=PRS-Simone \
--identity-file=PRS-Simone.pem \
--region=eu-west-1 \
--zone=eu-west-1a login prs-spark-cluster
```

F.3.4 Build and include the dependencies

To executing the application in the Cluster, we have to provide the Master all the necessary dependencies. To doing this, it is recommended to use the SBT compiler for Scala and the plugin ASSEMBLY [2]. This plugin allows creating a big Jar containing all the dependencies described in the build.sbt file in the project directory.

In out build.sbt file we have listen the following dependencies:

[2]SBT assembly, `https://github.com/sbt/sbt-assembly`

```
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.5.1" % "provided"
libraryDependencies += "org.apache.spark" %% "spark-sql" % "1.5.1" % "provided"
libraryDependencies += "org.apache.spark" %% "spark-mllib" % "1.5.1" % "provided"
libraryDependencies += "org.apache.spark" % "spark-graphx_2.10" % "1.5.2" % "provided"
libraryDependencies += "com.datastax.cassandra" % "cassandra-driver-core" % "2.1.9"
libraryDependencies += "com.datastax.spark" % "spark-cassandra-connector_2.10" % "1.5.0-M2"
```

The Spark-core library is the basic library that provides the basic functionality of spark. Then the *Spark-SQL* library allows us to communicate to the Cassandra database and perform some relational operations on it. The *spark-mllib* and the *spark-graphx* are libraries that provides some useful functions to make easier to develop a machine learning application and when we have to manipulate a network structure respectively. The *cassandra-driver-core* is the driver that allows us to read and write data from Cassandra using the Spark.Cassandra Connector listed Abbot in the build file. To starting the building of the Jar file use the following command:

```
sbt clean assembly
```

This generate a jar in the *$PROJECT_DIR/target/scala-2.10/* directory.

We can run then a Spark spplication in local, to testing propose by running the following command:

```
.spark-submit \
--class main.scala.HybridRecomendationWithEvaluationOptimized \
--master local[8] target/scala-2.10/my-project-assembly.jar 2>&1 | tee log.txt
```

F.3.5 Setting up the Master

When we have a running EC2 Cluster we can upload our compiled jar with all the dependencies on the Master for example via SSH.

To accessing to the the S3 service we have to export the AWS credentials in the environment variables .

```
export AWS_ACCESS_KEY_ID=XXXXXXXXX
export AWS_SECRET_ACCESS_KEY=XXXXXXXXXXXXXXXXXXX
```

F.3.6 Run a Spark application

Then we can run a Spark application by using the *./spark-submit* script as showed below:

```
./spark-submit \
--class main.scala.HybridRecomendation \
--master spark://MASTER-IP:7077 \
my-project-assembly.jar 2>&1 | tee log.txt
```

Project specification

MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering

HES-SO//Master- TIC

Products recommendation system improved with social network information

Master thesis Specifications

Auteur
Simone Cogno

In the HE-Arc school

Professors:
Ghorbel Hatem
Punceva Magdalena

Expert:
David Jacot (Swisscom)

Version 1.1

Neuchâtel, HES-SO//Master, February 4, 2016
Master of Science HES-SO in Engineering, Av. de Provence 6 CH-1007 Lausanne

VERSIONS HISTORY

Version	Date	Description
v1.1	29.09.2015	Second draft of project specification
v1.0	23.09.2015	Fist draft of project specification

Table 0.1 – Versions history

CONTENTS

Chapitre 1 - Project description 2

1.0.1 Approach . . . 2

Chapitre 2 - Project objectives 3

2.1 Main objectives . . . 3

2.2 Secondary objectives . . . 3

Chapitre 3 - Tasks to be performed 4

3.1 Project starts . . . 4

3.2 Analysis . . . 4

3.3 Design . . . 4

3.4 Implementation . . . 4

3.5 Tests . . . 4

3.6 Finalization . . . 4

Chapitre 4 - Planning 6

4.1 Gantt diagram . . . 6

Bibliography 8

here to avoid problems with pdfbookmark

1 PROJECT DESCRIPTION

This project is focused on the creation of a the recommendation systems that can recommend specific content adapted to interests and needs of users. In this project we are focused on the recommendation of books. We collect the dataset from Goodreads.com by using their public API [1].
The dataset contains a list of books, users and ratings. A list of friends is also available for describing the relation between the users.

1.0.1 Approach

There are several techniques used for creating a recommendation system. The most common is the "Content-based systems". This technique uses the information about the ratings (and other explicit information) that a user has made on documents or items. The system can then recommend other items that are similar of what the user have rated positively.
This approach can then be improved using a "Social recommender system", that use the information about the users relations to improve the recommendation quality.
In our project we will use these techniques as we have a lot of data about user ratings and the social network of friends.

In fact our dataset is composed of

- List of books with their meta-data (title, author, description, Avg. rating, genres & tags)
- List of users
- List of user rating and reviews
- List of user-user friend relations

Other improvements can also be made using the "Sentiment analysis" technique that use the review's text for determining additional information about the interests of users.

1. Goodreads API, 2015, https://www.goodreads.com/api/

2 PROJECT OBJECTIVES

The aim of this project is to analyze the different techniques used for a content-based recommendation systems and to choose the best solution for our project. After the research, an implementation of the algorithm will be made as well as several tests for demonstrating the correctness and the performance of the system.
As a secondary objective, there is the realization of a standalone application that show the working system. Then we want to make some adaptations of the recommendation algorithm for participating at the RecSys challenge 2015 [1].

The principal objectives of this project are the following:

2.1 Main objectives

- Establish the domain and the dataset that will be used to implement a recommendation system
- Establish the state of the art of the principals techniques for the "Content-based recommendation" as well as for the "Social recommender"
- Choose and study the best technique for our particular subject and dataset
- Make a research of possible technologies/framework for implement the recommendation system
- Realize the architecture and implement the algorithm
- Provide several measures of the performance and the correctness of the system

2.2 Secondary objectives

- Realize a stand-alone application that demonstrate the work done
- Adapt and test the system on the RecSys challenge 2015 [2]

1. RecSys Challenge 2015, http://recsys.acm.org/recsys15/challenge/
2. RecSys Challenge 2015, http://recsys.acm.org/recsys15/challenge/

3 TASKS TO BE PERFORMED

3.1 Project starts

— Access to Git and Forge
— Project specifications
— Planning

3.2 Analysis

— Research of the datasets available for testing a recommendation system
— Establish the state of the art of the principals techniques for the "Content-based recommendation" as well as for the "Social recommender"
— Analysis of technologies / framework
— Analysis of the recommendation technique chosen
— Write the analysis part of the final rapport

3.3 Design

— Design the architecture of the system
— Design the recommendation algorithm to be implemented
— Write the design part of the final rapport

3.4 Implementation

— Implementation of the architecture
— Implementation of the recommendation algorithm

3.5 Tests

— Measure the performance of the algorithm (Recall, Precision, etc.)
— Measure the correctness of the algorithm with a ground-truth dataset

3.6 Finalization

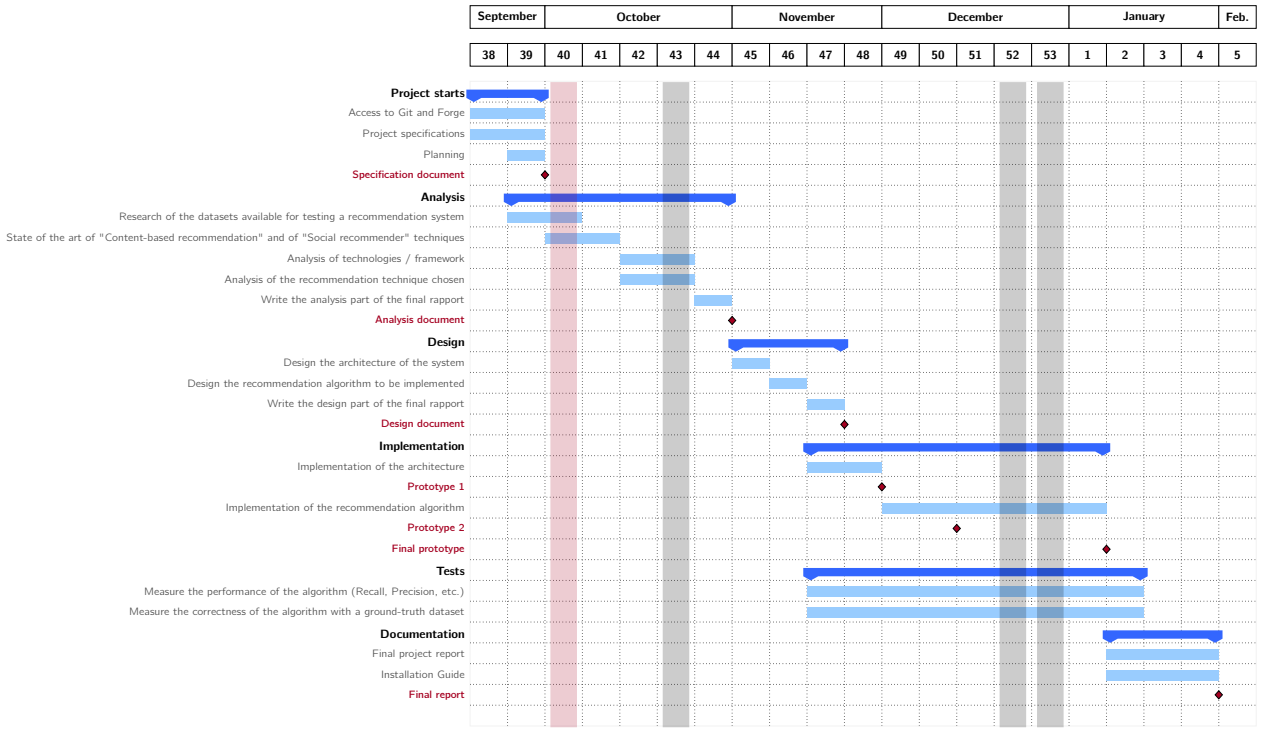
— Write the final project report
— Installation Guide

— User Guide

4 PLANNING

4.1 Gantt diagram

See annexe.


September
October
November
December
January
Feb.
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
1
2
3
4
5
Project starts
Access to Git and Forge
Project specifications
Planning
Specification document
Analysis
Research of the datasets available for testing a recommendation system
State of the art of "Content-based recommendation" and of "Social recommender" techniques
Analysis of technologies / framework
Analysis of the recommendation technique chosen
Write the analysis part of the final rapport
Analysis document
Design
Design the architecture of the system
Design the recommendation algorithm to be implemented
Write the design part of the final rapport
Design document
Implementation
Implementation of the architecture
Prototype 1
Implementation of the recommendation algorithm
Prototype 2
Final prototype
Tests
Measure the performance of the algorithm (Recall, Precision, etc.)
Measure the correctness of the algorithm with a ground-truth dataset
Documentation
Final project report
Installation Guide
Final report

BIBLIOGRAPHY

[1] "Products recommendation system on social media" project statement, Ghorbel Hatem, 2015