University of Piraeus

Master Thesis

Recommender Systems Comparison

Vasileios Simeonidis

supervised by
Dr. Dimosthenis Kyriazis

August 15, 2017

Contents

I Master Thesis **1**

1 Intro **1**

2 Related Work **2**
2.1 RecBench: Benchmarks for Evaluating Performance of Recommender System Architectures [1] 2
2.2 Recommender Systems Evaluation: A 3D Benchmark [2] 2
2.3 RiVal: A New Benchmarking Toolkit for Recommender Systems [3] 3

3 Collaborative filtering **4**
3.1 Content based 4
3.2 Latent Factors 5

4 Experiment **7**
4.1 Infrastructure 7
4.1.1 Apache Spark 7
4.2 Dataset 11
4.3 Implementation and assumptions 11

5 Results **12**

6 Conclusion **21**

II Appendices **23**

A Code used **23**
A.1 Item Based Collaborative Filtering 23
A.2 Latent Factors 28
A.3 Infrastructure code 30

This page was intentionally left blank.

Part I
Master Thesis

1 Intro

The last decade Internet has been flooded with information. Information that no one can filter to find what he needs, raw data, videos, music or products. Large retail sites like amazon, developed recommenders systems in order to offer products to their users. The need although is not limited only in the retail area. Web sites like youtube or vimeo needs to recommend to each user of their, videos that may like to watch next. Facebook is another example of an application utilizing lots of data and offering recommendations on what you may want to read or who may be a friend of yours. Most of the times, a recommender system is not the core business functionality of an application. It is though a very useful feature that gives a clear advantage on any business area needed.

The way a recommender system has been built is very dependent on the business case which will be served. Even a specific case of recommendation, similar to one already existing, might need a different recommender system.

So far recommender systems has been very interesting area of study. Netflix in 2009 declared a challenge, which can be found here [1]. Its reward was one million dollars for the task of improving the accuracy of predictions. The prize was granted to BellKors Pragmatic Chaos team for their algorithm. You can come across this challenge on lots of papers published every year.

With such a wide study of recommender systems, it is reasonable to start wondering "How are we going to compare recommender systems?". As we will discuss bellow, there are several papers suggesting ways of comparison. The majority of those papers are using the dataset given in the challenge above.

In this thesis the first system to compare is a content-based recommendation system that provide predictions based on movies genre attributes. The second system is the based on the Alternating Least Squares (ALS) implementation of Apache Spark's MLlib.

The comparison metrics used are the Mean Absolute Error (MAE), the Root Mean Squared Error and the ration between them (MAE/RMSE). Last but not least is the execution of time metric, measuring the training and estimation time.

[1] http://www.netflixprize.com/

2 Related Work

In this thesis's section we will list numerous different approaches made in order to compare recommender systems.

2.1 RecBench: Benchmarks for Evaluating Performance of Recommender System Architectures [1]

University of Minnesota, published in 2011 a paper stating a comparison between a recommender framework and a DBMS-based recommender. In that paper they used the Movie Lens dataset 100k, from the Netflix Challenge. The benchmark had five areas of comparison. Those areas were initialization, pure recommendation, filtered recommendation, blended recommendation, item recommendation and item update.

The initialization task was about the preparation needed for the system to go live. The next area was pure recommendation. By pure recommendation the author mean the home page recommendation, meaning the items that are going to be in the home page. Moving forward, we find the filtered recommendation. This recommendation is constrained by variables specific to the item, like movie genre etc. Another area of this evaluation contains the blended recommendation. Those recommendations are based on free text provided by the use in order to search. Item prediction is another are of the evaluation, in this prediction the user is navigated to the items page and the system is trying to predict the user's rating on the item. Last but not least, the paper examines the case of a new item being added to the system and how this is going to be incorporated to it.

As a result, of those experiments the paper conclude that "hand-build recommenders exhibit superior performance in model building and pure recommendation tasks, while DBMS-based recommenders are superior to more complex recommendations such as providing filtered recommendations and blending text-search with recommendation prediction issues.

2.2 Recommender Systems Evaluation: A 3D Benchmark [2]

In this paper the authors recognize the need for a common benchmark formula for recommender systems. This need lead them to propose one. They named it the 3D recommendation evaluation because they evaluate a system in three axis. These axis are business models, user requirements, and technical constrains. In business model axis they state that a recommender system must be evaluated on how well it serves the business case it is used for. In their paper they give the example of a video on demand service and evaluate it versus the pay per view business model and pay per subscription.

In the user requirements axis, the evaluate the system based on what need it covers for the users. Is it, for example, going to reduce search time or decision making time.

Last but not least is the technical constrains axis. In this axis the system is being evaluated based on data or hardware constrains, scalability and robustness.

2.3 RiVal: A New Benchmarking Toolkit for Recommender Systems [3]

RiVal, is an open source tool kit implemented in Java programming language. Rival is available via maven repositories. It is used in order to measure the evaluate recommender systems. Its evaluation is based on three points. Those point are data spiting, item recommendation, candidate item generation and performance measurement.

3 Collaborative filtering

The need of having recommender systems lies between the need of obtaining recommendations from trustworthy sources and the availability of a large amount of user data.

Like on any demand and supply system, on the one hand lies the demand of accurate and trustworthy recommendations. On the other hand are the tons of user data that can serve this demand.

Over the years have been developed techniques that can utilize those data, in order to provide good recommendations. Those techniques are highly dependent on the volume of data they use.

The more the data the more accurate the recommendation. But its system's training phase is largely impacted by the volume mentioned before. Thus, any algorithm or system is will provide good recommendations as long as it is trained with the right dataset.

Those algorithms were initially based only on statistical models that were available at the time. Whereas the data were growing rapidly, and the sample started to approach the population.

3.1 Content based

The most common and easy to interpret way of recommendation is collaborative filtering. In this area of algorithms you are trying to utilize data from other users in order to come to a recommendation. Those data might be attributes that characterizes the item of interest.

In case of users those attributes may be their age or occupation, whereas for a product might be its color, prize or weight in kilograms. In order to put this to a mathematical expression we could write the following.

$$w = R^{-1}M^T \tag{1}$$

Raw data though are not always clear of normalized. Due to this fact we would consider to normalize the approach we used above. If we were about to add a normalization factor to that expression we will end up with the one below.

$$w = (\lambda I + R^T R)^{-1} R^T M \tag{2}$$

Those kind of algorithms are easier to interpret. They also can handle well a cold-start problem. But they are computational heavy, meaning that the scaling of them is limits.

3.2 Latent Factors

Another approach to recommender algorithms is the latent factors group. This group of algorithms does not take into consideration the meta-data we have on a any user or item.

In that area we are trying to determine relationships between a user and an item based only on the rating. Those relationships may not be the age or the color.

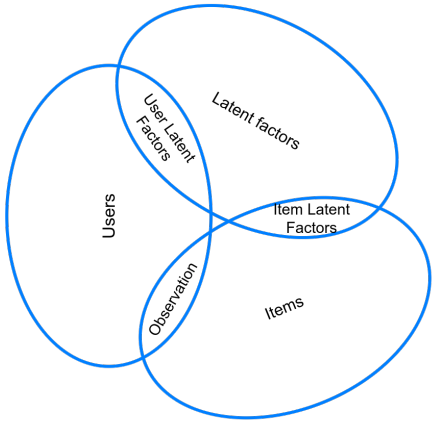

Figure 1: **LatentFactors**

Alternating least squares (ALS) algorithm belongs to the group above. In this case we assign initially random values of rating between user and items. Then it takes the error between the actual value and the one assigned to it.

Then the algorithm runs again using as input the errors and tries to minimize them. Below we can see a how this algorithm is defined.

Algorithm 1 ALS for Matrix Completion

1: Initialize X,Y
2: **repeat**
3: **for** `u=1...n` **do**
4: $x_u = (\sum_{r_{ui}} y_i y_i^T + \lambda I_k)^{-1} \sum_{r_{ui}} r_{ui} y_i, \in r_{u*}$
5: **for** `i=1...m` **do**
6: $y_i = (\sum_{r_{ui}} x_u x_u^T + \lambda I_k)^{-1} \sum_{r_{ui}} r_{ui} x_u, \in r_{*i}$
7: **until** convergence

As we can see above, this algorithm has a λ parameter used for normalization during the process. We can see the deference below were we have both the expression with and without normalization factor.

ALS is a very efficient recommender algorithm. Due to its nature it can be easily parallelized reducing the execution time needed [4]. It also requires no meta-data about any user or item. Although, ALS suffers from the cold start problem.

$$\min_{X,Y} \sum_{r_{ui} observed} (r_{ui} - x_u^T y_i)^2 \tag{3}$$

$$\min_{X,Y} \sum_{r_{ui} observed} (r_{ui} - x_u^T y_i)^2 + \lambda (\sum_u ||x_u||^2 + \sum_i ||y_i||^2) \tag{4}$$

Moving forward this thesis, we are going to discuss how those two algorithms were implemented and validate the results they gave.

4 Experiment

4.1 Infrastructure

As the experiment's infrastructure we will describe the frameworks used during the implementation. The framework that has been used to implement the item-based algorithm was Apache Spark. The ALS algorithm was used from the Apache Spark's MLLib library. The framework is common to infrastructure because Apache Spark can orchestrate the work on multiple machine as well as in one. So the framework is as close as we can get to the infrastructure.

4.1.1 Apache Spark

The last decade, analyzing big data is at its peek. Lots of data are produced on daily basis. This means that the need of extracting information from them is raised. Lots of frameworks has been used in order to manage and analyze this amount of data. One of the analysis reasons is the need for accurate item recommendations to users. Those items could be movies (e.g. Netflix), music (e.g. Spotify) or products in general(e.g. Amazon). The one of the most popular frameworks that could enable this in a distributed way was Apache's Hadoop MapReduce.

Apache Hadoop has discrete jobs of processing data. The most common jobs are map and reduce but it has two more jobs, combine and partition. Hadoop has a master node and N worker nodes. The first is responsible to distribute the work, and the second for the work to be done. Each worker usually is called after the job is executing. Hence we have the mapper, the reducer, the partitioner and the combiner. In order to put this to a schema, you can see the figure 2 below.

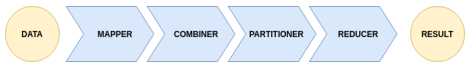

Figure 2: **Hadoop Jobs Order**

Hadoop map reduce, is a distributed map-reduce system, this means that it has a mechanism to distribute work on nodes and a common interface for handling data. In Hadoop's case this was able to happen due to Apache Hadoop Yarn and the Hadoop Distributed File System or as commonly used HDFS. When a job was scheduled, data were loaded by the HDFS to a worker, then the worker was done, he was putting the result back to the HDFS.

As mentioned in [5], "The term MapReduce actually refers to two separate and distinct tasks that Hadoop programs perform. The first is the map job, which takes a set of data and converts it into another set of data, where individual

elements are broken down into tuples (key/value pairs). The reduce job takes the output from a map as input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce job is always performed after the map job."

So Hadoop has two basic processes, Map which is responsible for turning the data into key value pairs, and Reduce which takes those pairs and turns them into valuable data.

If we would like to see where in the DIKW (Data Information Knowledge Wisdom) stack. The Map process would with data and the reduce will end up with information. Of course this in not always the case, lots of algorithms require lots of cycles in order to complete.

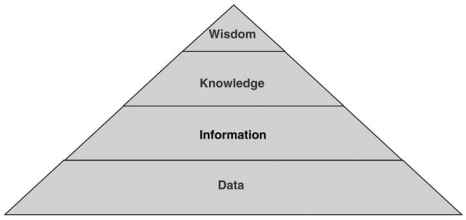

Figure 3: **Data Information Knowledge Wisdom Pyramid [6]**

But lets make a step back and take a look at Hadoop's architecture. As it is described in its official website [7], and shown in the figure 4 Hadoop uses Hadoop yarn in order to coordinate which process will run on which machine. Also it uses, its file system, the HDFS in order to have a common reference for the files over the network. Last but not least, Hadoop ecosystem is supported by the Hadoop Commons library.

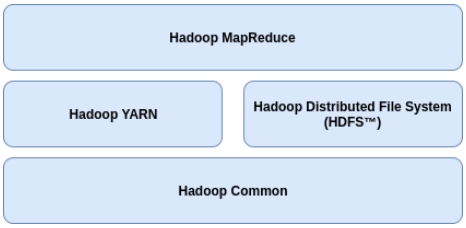

Figure 4: **Hadoup Software Stack**

In 2009, University of California, Berkley, proposed a new framework for cluster computing in their paper, Spark: Cluster Computing with Working Sets [8]. They wanted to tackle two major Hadoop issues.

The first was the iterative jobs. Each Hadoop job reads from the disk to load data. This means that having iterative jobs, on any given algorithm, you were going to get a large time penalty on reading and of course writing to the disk.

The second issue was the interactive analytics. Each Hadoop SQL interface was running as a seperete job, and as we mentioned previously we have a big impact on execution time.

In order to break the acyclic nature of Hadoop, they introduced the Spark's major abstraction, the RDDs. The name RDD stands for Resilient Distributed Datasets. Those datasets are a read-only collection of objects distributed across machines. If a machine fail, the lost part of the RDD can be recalculated. This notion is called lineage.

Spark is implemented in Scala. Scala is a high-level statically typed programming language. At the time that paper was published it was believed that Spark was the only system available in a general purpose programming language to make clusters process large amount of data. As it was mentioned in [8] "We believe that Spark is the first system to allow an efficient, general purpose programming language to be used interactively to process large datasets on clusters"

Back to RDDs, an RDD can be created with four different operations as it is described in [8]. The first operation is loading data from any shared file system. That file system could be HDFS or even an Amazon S3. The second way to create a RDD is by parallelizing any Scala collection. Spark will slice the collection into pieces and distribute it among the nodes. The third way is via transforming an RDD to another one. Because RDDs are immutable, any transformation operation on a RDD, filter, map, flatmap, will generate a new RDD. The last but not least method is by changing an RDDs persistence using save or cache operations.

Spark also give us the power to do a lot of different distributed operations. Some of them was mentioned before, but we also have operations that would return data to the driver program like collect or reduce.

Another important feature of Sparks spine are the shared variables. Spark at its first appearance introduced two of them. The first shared variable is the broadcasted variables. Those variable, RDDs or not, are variables that are commonly used in an algorithm, like a look-up table. By broadcasting a variable, each node gets a copy of the variable in order to access it quickly. The second shared variable that was introduced in that paper was the Accumulators. Those variable live on the spark context, but they can only be increased by any worker and be read from the driver program only. That paper concludes that

Spark can outperform Hadoop in some machine learning algorithms and more specific on logistic regression.

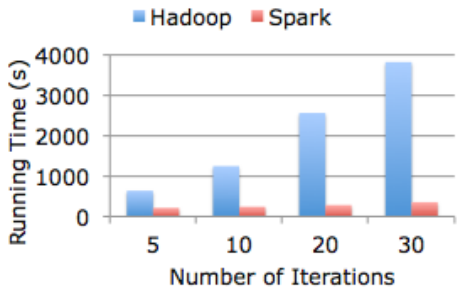

Figure 5: **Logistic regression, Hadoop vs Spark [9]**

Coming back to today, Spark's current architecture is depicted below in fig. 6. Spark nowadays has an sql interface in order to search in RDDs with in a query language. Also Spark support a streaming api to make available real-time analytics. Most of the core machine learning algorithms like ALS, which I used in order to complete this thesis, are inside the Spark's MLlib component. Finally spark has the component GraphX that is used for handling graphs and graph computation.

Apache Spark was by design meant to work within Hadoop ecosystem, and most importantly with the HDFS. Apache Spark does not has a file system by itself. You can load data from almost any database, cloud-based or not, even from a local file system. But most will agree that Hadoop and Spark work together just perfect.

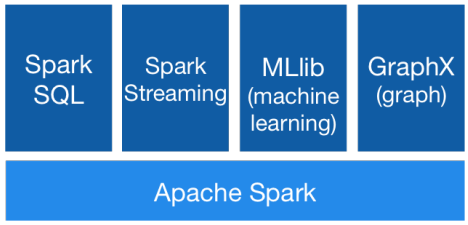

Figure 6: **Apache spark stack [10]**

To conclude, Spark has dominated the big data field the last years, Amazon and other cloud providers gives you the option to deploy an Apache Spark cluster on their infrastructure. Also large companies like Google and Intel are actively

contributing to projects like Apache Spark On Kubernetes which can be found at its Github repository [2]

4.2 Dataset

Selecting a clear and reliable dataset is very important for every experiment in the field. Due to that need several papers, like [1], are using the Movie lens dataset. This dataset contains users, movies and ratings. Each user may rated more than movie and each movie can be rated by more than one user. The dataset split to multiple subsets of 80000 training sets and respective 20000 testSet. It also provides scripts that can be used to create more sets.

In order to better visualize the above dataset, we created an entity relationship(ER) diagram below.

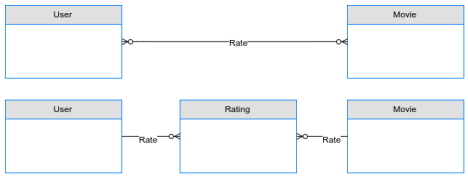

Figure 7: **MovieLens ER diagram [11]**

4.3 Implementation and assumptions

During the implementation I had to make some assumptions and choices. The first of choices was the framework and the programming language that the implementation would take place. The framework that has been chosen, as you may had already figured out, is apache spark due to its trend and the high scalability it offers. The language of choice was scala, due to its functional nature.

[2]https://github.com/apache-spark-on-k8s/spark

5 Results

In this chapter we will examine the results taken from the experiment described in the previous chapter. In order to measure the performance of each recommender system we used three different metrics. Those metrics are the Mean Absolute Error (MAE), the Root Mean Square Error (RMSE), and the execution time of each system.

The Mean Absolute Error is defined as the sum of each error's absolute value divided by the number of them. Let's take a look at error's definition. It is common in statistics to symbolize the error with e_i. The i index shows which measurement's error is. An error can be calculated using the following formula $e_i = \widehat{y_i} - y_i$, referring to $\widehat{y_i}$ as the estimated value for the i-th item whereas to y_i as the actual value of the i-th item. For example if we estimated that the $\widehat{y_i} = 5$ and its actual value is $y_i = 5.2$ and its error could be the following.

$$e_i = \widehat{y_i} - y_i => e_i = 5 - 5.2 => e_i = -0.2 \tag{5}$$

Mean absolute error is easier to interpret than other statistical metrics like RMSE. We will examine RMSE later in this chapter. If we want to express mathematically the MAE we could write the following.

$$MAE = \frac{\sum_{i=1}^{n} |e_i|}{n} = \frac{\sum_{i=1}^{n} |\widehat{y_i} - y_i|}{n} = \frac{\sum_{i=1}^{n} \sqrt{(\widehat{y_i} - y_i)^2}}{n} \tag{6}$$

During the experiment we took the MAE metric for each system. The two following tables can show the results in detail.

Table 1: **Content Based Algorithm Results vs Mean Absolute Error**

Training Dataset — Testing Dataset	**Mean Absolute Error**
u1.base — u1.test	1.6467431428
u2.base — u2.test	1.6055222167
u3.base — u3.test	1.6089259075
u4.base — u4.test	1.6259192043
u5.base — u5.test	1.6284658627
ua.base — ua.test	1.642536458
ub.base — ub.test	1.6357196576

Table 2: **Latent Factors Algorithm Results vs Mean Absolute Error**

Training Dataset — Testing Dataset	**Mean Absolute Error**
u1.base — u1.test	1.1818684937
u2.base — u2.test	1.1800652808
u3.base — u3.test	1.1783366748
u4.base — u4.test	1.1730543877
u5.base — u5.test	1.1686585292
ua.base — ua.test	1.2008035301
ub.base — ub.test	1.2134460078

As we can see, comparing the two systems on this metric we realize that the latent factors system outperformed the content based one. That is clearly depicted on the graph below.

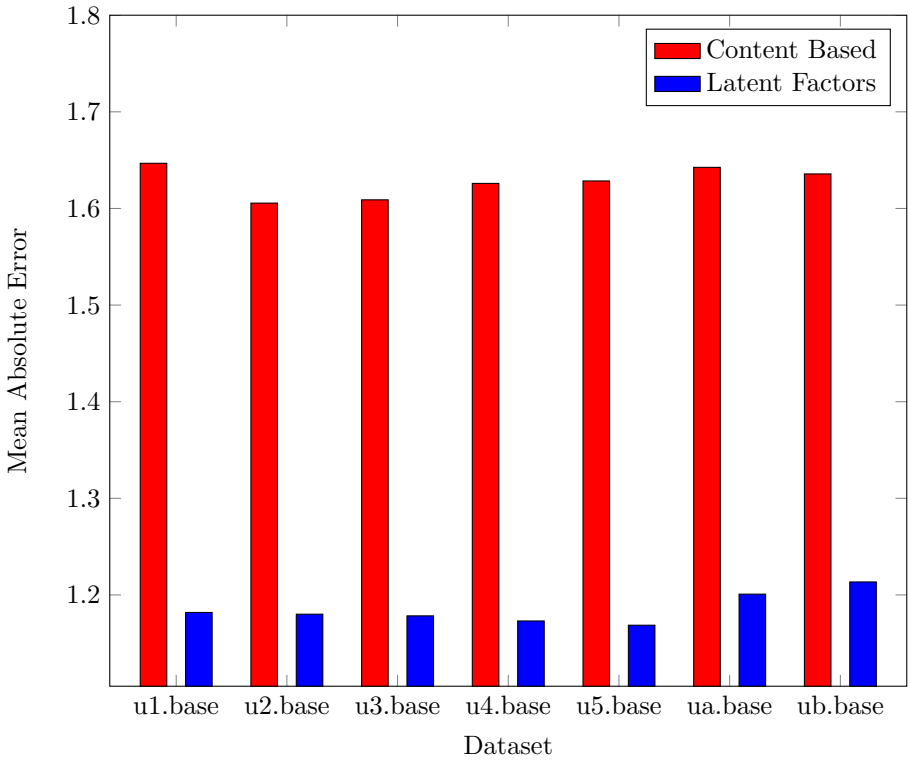

Figure 8: **Latent Factors vs Content Based on Mean Absolute Error**

In bibliography the most common statistical metric for recommender systems is the root mean square error(RMSE). This metric is considered to give a greater estimation on error magnitude due to the fact that it uses the squared value of each error. In order to better understand this metric we can have a look at its mathematical type below.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} e_i^2}{n}} = \sqrt{\frac{\sum_{i=1}^{n} (\widehat{y_i} - y_i)^2}{n}} \tag{7}$$

Each error is measured as before but now due to the structure of that metric the larger the error it is, the greatest the impact it has on RMSE.

The results we got during the experiment on that metric was that latent factors system outperformed the content base again. You can see the details of the result on the two following tables.

Table 3: **Content Based Algorithm Results vs Root Mean Square Error**

Training Dataset — Testing Dataset	**Root Mean Square Error**
u1.base — u1.test	2.1040777758
u2.base — u2.test	2.0594484822
u3.base — u3.test	2.0914574229
u4.base — u4.test	2.0862531343
u5.base — u5.test	2.0990150946
ua.base — ua.test	2.1224499666
ub.base — ub.test	2.0994553018

Table 4: **Latent Factors Algorithm Results vs Root Mean Square Error**

Training Dataset — Testing Dataset	**Root Mean Square Error**
u1.base — u1.test	1.3793448223
u2.base — u2.test	1.3784149025
u3.base — u3.test	1.3754645935
u4.base — u4.test	1.3706400993
u5.base — u5.test	1.3668078009
ua.base — ua.test	1.3968580906
ub.base — ub.test	1.4119780481

The results above can be shown clearly on the graph following, depicting the performance of latent factors and content based systems, on each dataset, on RMSE metric.

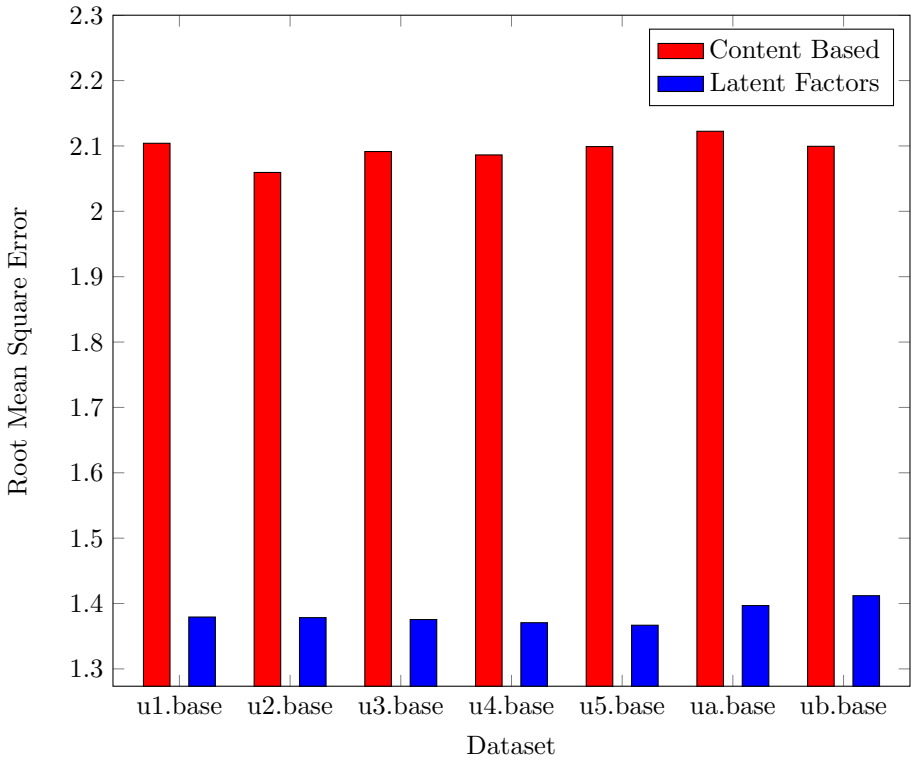

Figure 9: **Latent Factors vs Content Based on Root Mean Squeare Error**

Even if RMSE is considered a better metric when large errors are undesired, it is useful to check those systems on the $\frac{MAE}{RMSE}$ metric too. It is easily proven that $MAE \leq RMSE$. Those two are equal when the magnitude of all errors is the same. Examining this ratio we can see if the errors magnitude have close values.

Table 5: **Content Based Algorithm Results vs MAE over RMSE**

Training Dataset — Testing Dataset	**MAE \ RMSE**
u1.base — u1.test	0.7826436654
u2.base — u2.test	0.7795884338
u3.base — u3.test	0.769284562
u4.base — u4.test	0.779348957
u5.base — u5.test	0.775823798
ua.base — ua.test	0.7738870097
ub.base — ub.test	0.7791162099

Table 6: **Latent Factors Algorithm Results vs MAE over RMSE**

Training Dataset — Testing Dataset	**MAE \ RMSE**
u1.base — u1.test	0.8568332404
u2.base — u2.test	0.856103107
u3.base — u3.test	0.8566826659
u4.base — u4.test	0.8558442062
u5.base — u5.test	0.8550276992
ua.base — ua.test	0.8596460429
ub.base — ub.test	0.8593943861

As it was expected and this area of examination, latent factors system has ten percent (10%) less error spikes than the content based on every dataset. The graph below depicts the results.

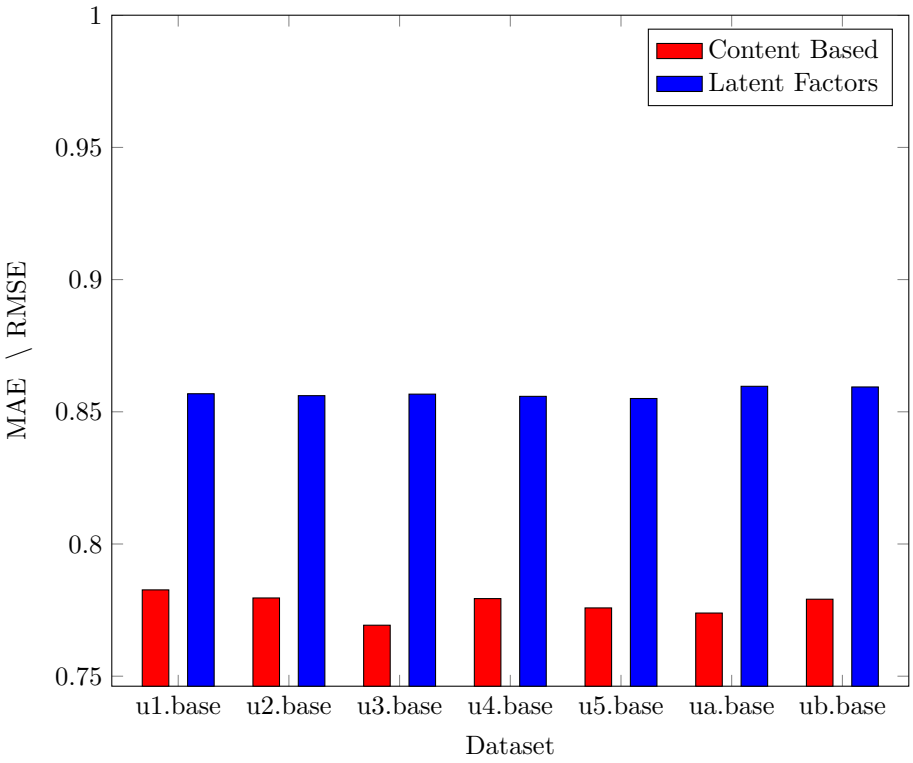

Figure 10: **Latent Factors vs Content Based on MAE over RMSE**

The last metric we took in order to compare those two systems was the execution time. On execution time metric we included the time need to train the system against a dataset and the time needed to calculate the metrics. We extracted the methods used on metrics calculation in order to be commonly used and impact the each execution time result on the same level. The results can be found in the tables below and on the graph that visualize them, also below.

Table 7: **Content Based Algorithm Results vs Execution Time**

Training Dataset — Testing Dataset	**Execution Time (ms)**
u1.base — u1.test	30514
u2.base — u2.test	27714
u3.base — u3.test	27164
u4.base — u4.test	26687
u5.base — u5.test	27124
ua.base — ua.test	26640
ub.base — ub.test	26861

Table 8: **Latent Factors Algorithm Results vs Execution Time**

Training Dataset — Testing Dataset	**Execution time (ms)**
u1.base — u1.test	10195
u2.base — u2.test	6517
u3.base — u3.test	5377
u4.base — u4.test	5433
u5.base — u5.test	5217
ua.base — ua.test	5214
ub.base — ub.test	5083

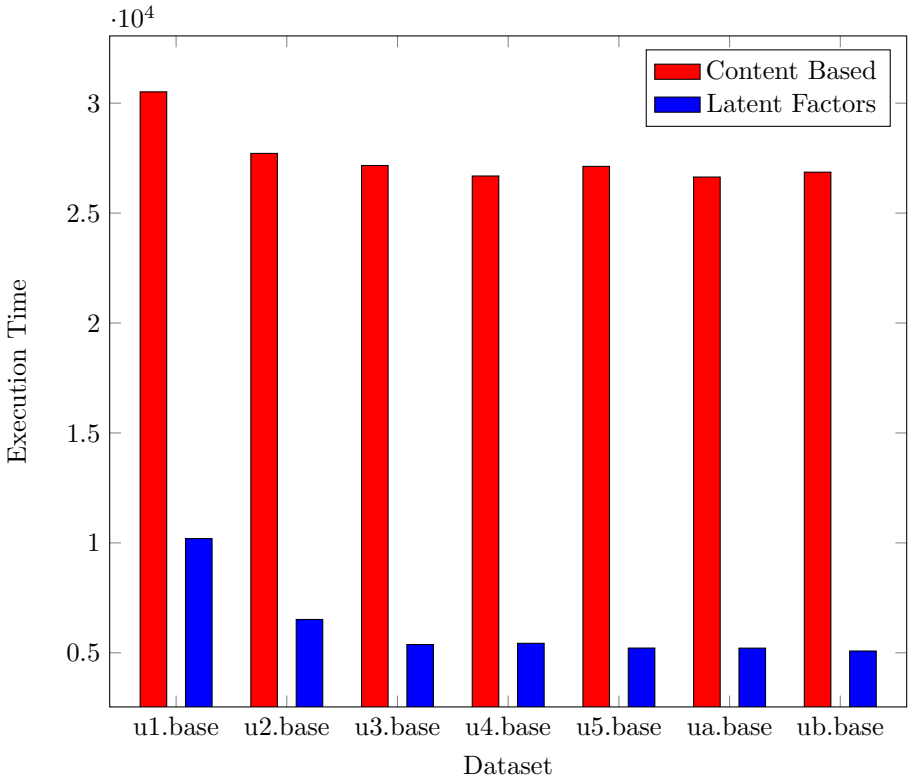

Figure 11: **Latent Factors vs Content Based on Execution Time**

Comparing those systems in the execution time metric we can also see that latent factors system outperformed the content base again by taking one third of the time on its worst case.

6 Conclusion

Recommender systems have a short but intense history. It started from simple statistical models and nowadays it is a great field of study. Recommender systems are now used widely in the online market. Every day you are using them without even noticing it. While you are browsing videos or the web, getting a message from a friend or even listening to music a recommender system might serve you at the time.

In this last chapter of this thesis, we will summarize the experiment and the result we got. This thesis is the attempt of the author to compare two recommender algorithms. Those algorithms were the classic content based and the alternative least squares. The first one is in the area of collaborative filtering while the other is in the latent factors area.

Those two algorithms were implemented or used in apache spark. The first, the content based algorithm was implemented, the second one the alternating least squares was used via apache spark's MLlib library.

Then those systems were put to test. As metrics were used the mean absolute error(MAE), the root mean square error (RMSE), the ratio between them (MAE/RMSE) and the execution time. Execution time is composed by two parts, the training time and the time taken to make the metrics. Because the metrics are common, on the same platform and they were using the same code, we can assume that the execution time difference has the training part and the prediction part for every rate in the test dataset.

During the results examination, as it was shown in the previous chapter, we found that the ALS system outperformed the content based on every metric we used. It shown low error metrics, MAE and RMSE, while execution time was low. The ratio MAE/RMSE was high. The last showed us that ALS has less data spikes comparing to the content based.

Recommender systems will be around for quite along time, it is important to know how to compare them. Even more important is to identify which recommender algorithm to use on each business case.

This thesis was a very important milestone for the author. This milestone couldn't be true without the help of those mentioned in the acknowledgment page.

Vasileios Simeonidis,
August 15, 2017

References

[1] J. J. Levandoski, M. D. Ekstrand, M. J. Ludwig, A. Eldawy, M. F. Mokbel, and J. T. Riedl, "Recbench: benchmarks for evaluating performance of recommender system architectures," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, 2011.

[2] A. Said, D. Tikk, K. Stumpf, Y. Shi, M. Larson, and P. Cremonesi, "Recommender systems evaluation: A 3d benchmark.," in *RUE@ RecSys*, pp. 21–23, 2012.

[3] A. Said and A. Bellogín, "Rival: a toolkit to foster reproducibility in recommender system evaluation," in *Proceedings of the 8th ACM Conference on Recommender systems*, pp. 371–372, ACM, 2014.

[4] B. H. Haoming Li, M. Lublin, and Y. Perez, "Cme 323: Distributed algorithms and optimization, spring 2015." `http://stanford.edu/~rezab/dao`, 2015. Lecture 14, 5/13/2015.

[5] "IBM what is map-reduce." `https://www.ibm.com/analytics/us/en/technology/hadoop/mapreduce/`. Accessed: 2017-06-24.

[6] J. Rowley, "The wisdom hierarchy: representations of the dikw hierarchy jennifer rowley," *Journal of Information Science*, pp. 163–180, 2007.

[7] "Apache Hadoop." `http://hadoop.apache.org/`. Accessed: 2017-06-24.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[9] "Hadoop Vs Spark." `https://amplab.cs.berkeley.edu/projects/spark-lightning-fast-cluster-computing/`. Accessed: 2017-06-24.

[10] "Apache Spark lightning-fast cluster computing." `https://spark.apache.org/`. Accessed: 2017-05-21.

[11] "MovieLens grouplens." `https://grouplens.org/datasets/movielens/`. Accessed: 2017-05-22.

Part II
Appendices

In this part you will find the code used for this master thesis.

A Code used

Below is the code used in order to produce the results. It has been written for Apache Spark [10] using scala. The library breeze has been used for matrix processing.

A.1 Item Based Collaborative Filtering

```
import java.util

import breeze.linalg.{Axis, DenseMatrix, pinv}
import breeze.optimize.linear.PowerMethod.BDM
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.{
    ↪ CoordinateMatrix, MatrixEntry}
import org.apache.spark.mllib.recommendation.Rating
import org.apache.spark.rdd.RDD




object ContentBased {

  val sparkContext: SparkContext = Infrastructure.sparkContext

  def main(args: Array[String]) {

    val bestNormalizationFactor = Infrastructure.
        ↪ normalizationFactorsList.map { v =>
      val sum = Infrastructure.dataSetList.map(dataSet =>
          ↪ getMetricsForDataset(v, dataSet._1, dataSet._2)).map(
          ↪ u => u._5).sum
      val mean = sum/Infrastructure.dataSetList.size
      (v, mean)
    }.maxBy(v=> v._2)

    Infrastructure.dataSetList
```

```
    .map(dataSet => getMetricsForDataset(bestNormalizationFactor
        ↪ ._2, dataSet._1, dataSet._2))
    .foreach(metric => println(metric))
  println("training␣set", "testing␣set", "MSE", "RMSE", "MAE", "
      ↪ Execution␣Time")

}


private def getMetricsForDataset(normalizationFactor: Double,
    ↪ trainingSet: String, testingSet: String) = {
  val startingTime = System.currentTimeMillis()

  val itemsMatrixEntries: RDD[MatrixEntry] =
      ↪ generateItemMatrixEntries

  val itemMatrix: Matrix = new CoordinateMatrix(
      ↪ itemsMatrixEntries).toBlockMatrix().toLocalMatrix()
  val itemMatrixBreeze = toBreeze(itemMatrix).copy

  val ratings = sparkContext.textFile(trainingSet)
    .map(_.split("\t") match {
      case Array(user, item, rate, timestamp) => Rating(user.
          ↪ toInt, item.toInt, rate.toDouble)
    }).cache()

  val usersRatings = ratings.groupBy(r => r.user)
    .map(v => (v._1, generateUserMatrix(v._2)))

  val refinedMatrices = usersRatings
    .map(v => (v._1, getRefinedMatrices(v._2, itemMatrixBreeze))
        ↪ )

  val userWeights = refinedMatrices.map(v => Pair(v._1,
      ↪ generateWeight(v, normalizationFactor)))
  val testRatings = sparkContext.textFile(testingSet)
    .map(_.split("\t") match {
      case Array(user, item, rate, timestamp) => Rating(user.
          ↪ toInt, item.toInt, rate.toDouble)
    }).cache()

  // remove rating from dataset
  val usersProducts = testRatings.map {
    case Rating(user, product, rate) => (user, product)
  }
```

```
  // predict
  val b = sparkContext.broadcast(userWeights.collect())

  val predictions = usersProducts.map(v =>
    ((v._1, v._2),
     predict(
       b.value.apply(v._1 - 1)._2,
       getRow(itemMatrixBreeze, v._2 - 1))
    ))

  val ratesAndPredictions = testRatings.map {
    case Rating(user, product, rate) => ((user, product), rate)
  }.join(predictions)

  ///// Metrics ////

  // calculate MSE (Mean Square Error)
  val MSE = Metrics.getMSE(ratesAndPredictions)

  // calculate RMSE (Root Mean Square Error)
  val RMSE = Math.sqrt(MSE)

  // calculate MAE (Mean Absolute Error)
  val MAE = Metrics.getMAE(ratesAndPredictions)

  val endingTime = System.currentTimeMillis()

  val executionTime = endingTime - startingTime

  (trainingSet, testingSet, MSE, RMSE, MAE, executionTime,
      ↪ normalizationFactor)
}

private def predict(weight: DenseMatrix[Double], item:
    ↪ DenseMatrix[Double]): Double = {
  val result = item.t * weight
  if (result.data.length > 1) {
    println("something␣went␣wrong␣on␣prediction")
    0
  }
  else result.data.apply(0)

}

private def generateWeight(v: (Int, (DenseMatrix[Double],
    ↪ DenseMatrix[Double])), normalizationFactor: Double):
```

```
    ↪ DenseMatrix[Double] = {
  calculateWeightsWithNormalizationFactor(v._2._2, v._2._1,
      ↪ normalizationFactor)
}

private def calculateWeightsWithNormalizationFactor(
    ↪ ratingMatrix :DenseMatrix[Double], itemMatrix:
    ↪ DenseMatrix[Double], normalizationFactor: Double):
    ↪ DenseMatrix[Double] = {
  val lambdaIdentity = DenseMatrix.eye[Double](ratingMatrix.cols
      ↪ ) :* normalizationFactor
  pinv(
    lambdaIdentity
      +
      (ratingMatrix.t * ratingMatrix)
  ) * (ratingMatrix.t * itemMatrix)
}


private def calculateWeightsWithoutNormalizationFactor(
    ↪ ratingMatrix :DenseMatrix[Double], itemsMatrix:
    ↪ DenseMatrix[Double]): DenseMatrix[Double] = {
  pinv(ratingMatrix) * itemsMatrix
}

def getRefinedMatrices(userMatrix: DenseMatrix[Double],
    ↪ itemMatrix:DenseMatrix[Double]): (DenseMatrix[Double],
    ↪ DenseMatrix[Double]) = {
  var sequence = Seq[Int]()
  userMatrix.foreachKey { v =>
    if (userMatrix(v._1,v._2) == 0) {
      sequence = sequence :+ v._1
    }
  }
  val localItemMatrix = itemMatrix.delete(sequence, Axis._0)
  val localUserMatrix = userMatrix.delete(sequence, Axis._0)
  (localUserMatrix, localItemMatrix)
}


def getRow(matrix: DenseMatrix[Double], row: Int): DenseMatrix[
    ↪ Double] = {
  val numberOfColumns = matrix.cols
  val array = new Array[Double](numberOfColumns)
  for (i <- 0 until numberOfColumns){
    array(i)=matrix(row,i)
```

```
  }
  new DenseMatrix(numberOfColumns ,1, array)
}

def generateUserMatrix(userRatings: Iterable[Rating]):
    ↪ DenseMatrix[Double] = {

  val numberOfItems = Infrastructure.items.count().toInt
  val array = new Array[Double](numberOfItems)
  util.Arrays.fill(array, 0)
  userRatings.foreach(r => array(r.product - 1) = r.rating)
  new DenseMatrix(numberOfItems ,1, array)
}

private def toBreeze(matrix: Matrix): DenseMatrix[Double] = {
  val breezeMatrix = new BDM(matrix.numRows, matrix.numCols,
      ↪ matrix.toArray)
  if (!matrix.isTransposed) {
    breezeMatrix
  } else {
    breezeMatrix.t
  }
}

private def generateItemMatrixEntries: RDD[MatrixEntry] = {
  Infrastructure.items.flatMap(a => Array(
  MatrixEntry(a(0).toLong - 1, 0, a(4).toInt),
  MatrixEntry(a(0).toLong - 1, 1, a(5).toInt),
  MatrixEntry(a(0).toLong - 1, 2, a(6).toInt),
  MatrixEntry(a(0).toLong - 1, 3, a(7).toInt),
  MatrixEntry(a(0).toLong - 1, 4, a(8).toInt),
  MatrixEntry(a(0).toLong - 1, 5, a(9).toInt),
  MatrixEntry(a(0).toLong - 1, 6, a(10).toInt),
  MatrixEntry(a(0).toLong - 1, 7, a(11).toInt),
  MatrixEntry(a(0).toLong - 1, 8, a(12).toInt),
  MatrixEntry(a(0).toLong - 1, 9, a(13).toInt),
  MatrixEntry(a(0).toLong - 1, 10, a(14).toInt),
  MatrixEntry(a(0).toLong - 1, 11, a(15).toInt),
  MatrixEntry(a(0).toLong - 1, 12, a(16).toInt),
  MatrixEntry(a(0).toLong - 1, 13, a(17).toInt),
  MatrixEntry(a(0).toLong - 1, 14, a(18).toInt),
  MatrixEntry(a(0).toLong - 1, 15, a(19).toInt),
  MatrixEntry(a(0).toLong - 1, 16, a(20).toInt),
  MatrixEntry(a(0).toLong - 1, 17, a(21).toInt),
  MatrixEntry(a(0).toLong - 1, 18, a(22).toInt))
  )
```

```
  }
}
```

A.2 Latent Factors

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.recommendation.{ALS, Rating}
import org.apache.spark.rdd.RDD

object LatentFactors {

  val sparkContext: SparkContext = Infrastructure.sparkContext

  def main(args: Array[String]) {

    val bestNormalizationFactor = Infrastructure.
        ↪ normalizationFactorsList.map { v =>
      val sum = Infrastructure.dataSetList.map(dataSet =>
          ↪ getMetricsForDataset(dataSet._1, dataSet._2, v)).map(
          ↪ u => u._5).sum
      val mean = sum/Infrastructure.dataSetList.size
      (v, mean)
    }.maxBy(v=> v._2)

    Infrastructure.dataSetList
      .map(dataSet => getMetricsForDataset(dataSet._1, dataSet._2,
          ↪  bestNormalizationFactor._2))
      .foreach(metric => println(metric))
    println("training␣set", "testing␣set", "MSE", "RMSE", "MAE", "
        ↪ Execution␣Time")

  }

  private def getMetricsForDataset(trainingSet:String, testingSet
      ↪ :String, normalizationFactor: Double) = {

    val startingTime = System.currentTimeMillis()

    val ratings = sparkContext.textFile(trainingSet).map(_.split("
        ↪ \t") match { case Array(user, item, rate, timestamp) =>
      Rating(user.toInt, item.toInt, rate.toDouble)
    }).cache()

    //// Build the recommendation model using ALS
    val rank = 15 // 10 - 20
    val numIterations = 75 // 50 - 100
```

```
    val model = ALS.train(ratings, rank, numIterations,
        ↪ normalizationFactor)

    //import test dataset
    val testRatings = sparkContext.textFile(testingSet).map(_.
        ↪ split("\t") match { case Array(user, item, rate,
        ↪ timestamp) =>
      Rating(user.toInt, item.toInt, rate.toDouble)
    }).cache()

    // remove rating from dataset
    val usersProducts = testRatings.map {
      case Rating(user, product, rate) => (user, product)
    }

    // predict the rating
    val predictions = model.predict(usersProducts).map {
      case Rating(user, product, rate) => ((user, product), rate)
    }

    // join rdd to get the rating and the prediction value for
        ↪ each combination
    val ratesAndPredictions: RDD[((Int, Int), (Double, Double))] =
        ↪  testRatings.map {
      case Rating(user, product, rate) => ((user, product), rate)
    }.join(predictions)

    ///// Metrics ////

    // calculate MSE (Mean Square Error)
    val MSE = Metrics.getMSE(ratesAndPredictions)

    // calculate RMSE (Root Mean Square Error)
    val RMSE = Math.sqrt(MSE)

    // calculate MAE (Mean Absolute Error)
    val MAE = Metrics.getMAE(ratesAndPredictions)

    val endingTime = System.currentTimeMillis()

    val executionTime = endingTime - startingTime

    (trainingSet, testingSet, MSE, RMSE, MAE, executionTime)
  }
}
```

A.3 Infrastructure code

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.RDD

import scala.collection.immutable

object Infrastructure {
  val sparkConfiguration: SparkConf = new SparkConf()
    .setMaster("local[*]")
    .setAppName("RecommenderSystemsComparison")
  val sparkContext: SparkContext = {
    val sc = new SparkContext(sparkConfiguration)
    sc.setCheckpointDir("checkpoint/") // set checkpoint dir to
        ↪ avoid stack overflow
    sc
  }


  //import data to rdds
  val users: RDD[Array[String]] = sparkContext.textFile("ml-100k/
      ↪ u.user").map(u => u.trim.split("\\|")).cache()
  val genres: RDD[Array[String]] = sparkContext.textFile("ml-100k
      ↪ /u.genre").map(u => u.trim.split("\\|")).cache()
  val items: RDD[Array[String]] = sparkContext.textFile("ml-100k/
      ↪ u.item").map(u => u.trim.replace("||", "|").split("\\|"))
      ↪ .cache()
  val occupations: RDD[String] = sparkContext.textFile("ml-100k/u
      ↪ .occupation").cache()
  val dataSetList = List(
    ("ml-100k/u1.base", "ml-100k/u1.test"),
    ("ml-100k/u2.base", "ml-100k/u2.test"),
    ("ml-100k/u3.base", "ml-100k/u3.test"),
    ("ml-100k/u4.base", "ml-100k/u4.test"),
    ("ml-100k/u5.base", "ml-100k/u5.test"),
    ("ml-100k/ua.base", "ml-100k/ua.test"),
    ("ml-100k/ub.base", "ml-100k/ub.test")
  )

  val normalizationFactorsList: immutable.Seq[Double] = List
      ↪ (0.01,0.03,0.06,0.09,0.12,0.15,0.18,1)
}

import org.apache.spark.rdd.RDD

object Metrics {
```

```
  def getMSE (ratesAndPredictions: RDD[((Int, Int), (Double,
      ↪ Double))] ): Double = {
    ratesAndPredictions.map { case ((user, product), (r1, r2)) =>
      val err = r1 - r2
      err * err
    }.mean()
  }

  def getMAE (ratesAndPredictions: RDD[((Int, Int), (Double,
      ↪ Double))] ): Double = {
    ratesAndPredictions.map { case ((user, product), (r1, r2)) =>
      val err = r1 - r2
      Math.abs(err)
    }.mean()
  }
}
```

List of Tables

1 **Content Based Algorithm Results vs Mean Absolute Error** 13
2 **Latent Factors Algorithm Results vs Mean Absolute Error** 13
3 **Content Based Algorithm Results vs Root Mean Square Error** 15
4 **Latent Factors Algorithm Results vs Root Mean Square Error** 15
5 **Content Based Algorithm Results vs MAE over RMSE** . 17
6 **Latent Factors Algorithm Results vs MAE over RMSE** . 17
7 **Content Based Algorithm Results vs Execution Time** . . 19
8 **Latent Factors Algorithm Results vs Execution Time** . . 19

List of Figures

1 **LatentFactors** 5
2 **Hadoop Jobs Order** 7
3 **Data Information Knowledge Wisdom Pyramid [6]** 8
4 **Hadoup Software Stack** 8
5 **Logistic regression, Hadoop vs Spark [9]** 10
6 **Apache spark stack [10]** 10
7 **MovieLens ER diagram [11]** 11
8 **Latent Factors vs Content Based on Mean Absolute Error** 14
9 **Latent Factors vs Content Based on Root Mean Squeare Error** 16
10 **Latent Factors vs Content Based on MAE over RMSE** . 18
11 **Latent Factors vs Content Based on Execution Time** . . 20