Master Thesis
Recommender Systems Comparison

Vasileios Symeonidis

27-05-2017

Contents

I Master Thesis **1**

1 Intro **1**

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

3 Collaborative filtering **2**
3.1 Content based . . . 2
3.2 Latent Factors . . . 2

4 Our Experiment **4**
4.1 Infrastructure . . . 4
4.1.1 Apache Spark . . . 4
4.2 Dataset . . . 7
4.3 Implementation and assumptions . . . 7
4.4 Metrics . . . 7
4.4.1 Mean Absolute Error . . . 7
4.4.2 Execution Time . . . 7

5 Results **8**

6 Conclusion **9**

II Appendices **12**

A Code used **12**
A.1 Item Based Collaborative Filtering . . . 12
A.2 Latent Factors . . . 17
A.3 Infrastructure code . . . 19

B Metrics **20**
B.1 What is the mean absolute error . . . 20
B.2 Time . . . 20

This page was intentionally left blank.

Part I
Master Thesis

1 Intro

This is the intro suction for this master thesis. Why we need recommendation systems? Retailers can propose the right product to the right target group. User get advertisements the may be interested in.[1]

2 Related Work

Below it is listed the related work to this thesis.

2.1 RecBench: Benchmarks for Evaluating Performance of Recommender System Architectures

2.2 Comparative Recommender System Evaluation: Benchmarking Recommendation Frameworks

2.3 Recommender Systems Evaluation: A 3D Benchmark

2.4 RiVal: A New Benchmarking Toolkit for Recommender Systems

3 Collaborative filtering

Collaborative filtering is the process of filtering items based on others items with similar attributes.

needs more explanation

What is collaborative filtering. [1]

3.1 Content based

In content based recommender systems we try to make recommendations based on features we know. There are two types of content based recommender systems.

On the one hand we have the user based recommendation. This recommendation is done by trying to match users profiles, in order to find which item the user i might like. But in real world we don't have the needed information to make the recommendation to the user.

On the other hand we have the item-product based recommendation, in this case we are trying to find user that might like the given product. This is match easier due to the fact that you know more about a product than a user, and you can classify them easily.

In this case we have a matrix R that contains the rates given by users to items. This matrix most of the times will be low in density, this is because each user does not rate each product. The second matrix we come across is the M. This matrix contains all the movies with the their genres. Each characteristic is binary. For example, the movie with id i is both action and comedy and none of the other genres.

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

In order add an normalization factor to the above equation, we need to get it to the form below.

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

3.2 Latent Factors

Latent factors techniques are used to find attributes that are no clear in the dataset. This means this set of algorithms is trying to find the best metric, which may not be a clear one.

In latent factors recommender systems we follow a similar approach but, in case of ALS(Alternating least squares), we are trying to find metrics that may lead us to the correct recommendation. Those metrics are not distinct, and may change in a number of iterations. Those metrics are inducted from the R matrix as we define it above. This makes this approach more tolerant to missing

values, or wrong quality measures. Thus this metric as will be presented bellow is more efficient on prediction and time. [2]

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

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

ALS explanation. ALS algorithm is based on the latent factors theory.As mentioned before, this means that it is not going to use the attributes given by the dataset for the movies or the users. The algorithm is going to train it self based on the rating set only.

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

4 Our 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. Also 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 the closest you 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 also 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. To put this to a schema, you can see the figure 1 below.

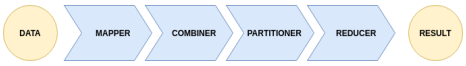

Figure 1: **Hadoop Jobs Order**

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 [3], "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 those processes belong, the map would start with data and the reduce will end up with information.

Of course this in not always the case, like we said above

TODO: remmember to metion above a recommender system would take more than one MapReduce cycles.

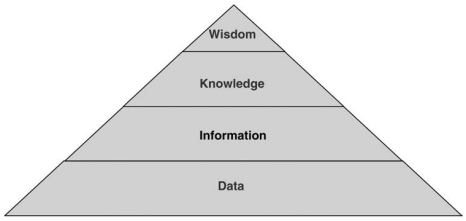

Figure 2: **Data Information Knowledge Wisdom Pyramid [4]**

But lets make a step back and take a look at Hadoop's architecture. As it is described in its official website [5], and shown in the figure 3 Hadoop uses Hadoop yarn in order to coordinate which process will run on which machine. Also it uses, its filesystem, 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.

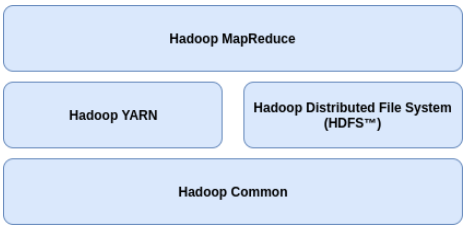

Figure 3: **Hadoup Software Stack**

In 2009 UC Berkley developed spark [6].

TODO: Update citation

Apache spark is the new trend on distributed computation and map-reduce. But first things first, what is map-reduce?

apache mesos -¿ data center operating system, references
But innovation knocked the door and resilient distributed datasets entered the room. In spark world, data are loaded to HDFS as before. Then spark loads them in an RDD, this means that data are now accessible on each machine's memory. Any transformation done to a RDD results a RDD, and so forth. After all the transformations are done, spark can transform the results to a file in HDFS.

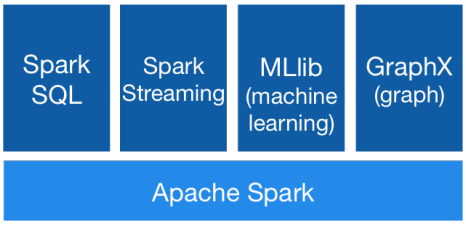

Figure 4: **Apache spark stack [7]**

TODO: Like Hadoop, Spark also follow the structure of master-worker. But unlike Hadoop, Spark has functions that can be distributed. To make things clearer, in spark we can have a map job and the result can be the input to another map job, whereas in Hadoop we couldn't do this.

Apache Spark was by design meant to work within Hadoop ecosystem, and most improtanly with the HDFS. Apache Spark does not has a file system by itself. This means that it has to load files from some source. Initially this source was HDFS but Spark has integration with more sources than HDFS. 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.

Once the data are loaded to spark, they are in a structure called RDD (Resilient Distributed Datasets). Those data sets live on disk and on memory. By this you can see that the speed of processing data on Apache Spark it could be much more faster than its predecessor Hadoop map-reduce.

TODO: cite a benchmark between hadoop map reduce and apache spark

How spark differentiates from its predecessors
Spark lightweight in memory data transformation Resilient Distributed Datasets (RDDs)
mllibs
add spark jira note

Important note: mention als distributed broadcasting implementation.

When a RDD is been broadcasted in Apache Spark, this RDD becomes a local matrix to each machine. This means that you can use it for local calculations. This adds an overload to each machine depending on RDD's size. But it eliminates the network usage. //cite the mastering apache spark book [7]
a Spark cluster to be created on AWS EC2 storage.
New trends on spark https://github.com/apache-spark-on-k8s/spark cite this repository too.

4.2 Dataset

What is the dataset about. This dataset contains users, movies and the rating user made about the movies. This dataset is splited to multiple subsets of 80000 training sets and respective 20000 reviews. This dataset has been used by the related work on ..... [8]

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.

4.4 Metrics

After the implementation I had to make the choice of the metrics I was going to use.

4.4.1 Mean Absolute Error

As metrics are commonly used the MSE, RMSE and MAE. Due to the fact that the author prefers the last one, MAE was used in this experiment.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} \sqrt{(y_i - x_i)^2}}{n} \tag{5}$$

4.4.2 Execution Time

Time is measured in milliseconds. Execution time is always a measure when we are comparing algorithms. Even more if those algorithms execution time is heavily dependent to their complexity and not their resources.

5 Results

Gathering the results of the above experiment, we can see roughly a very big difference between the two algorithms on both metrics, MAE and Execution Time.

Table 1: **Content Based Algorithm Results**

Training Dataset	**Testing Dataset**	**Mean Absolute Error**	**Execution time (ms)**
u1.base	u1.test	1.6467431428213226	30514
u2.base	u2.test	1.6055222166704628	27714
u3.base	u3.test	1.608925907479106	27164
u4.base	u4.test	1.6259192043203685	26687
u5.base	u5.test	1.6284658627202895	27124
ua.base	ua.test	1.6425364580036836	26640
ub.base	ub.test	1.6357196576385744	26861

Table 2: **Latent Factors Algorithm Results**

Training Dataset	**Testing Dataset**	**Mean Absolute Error**	**Execution time (ms)**
u1.base	u1.test	1.1818684937209607	10195
u2.base	u2.test	1.1800652808093945	6517
u3.base	u3.test	1.1783366748334452	5377
u4.base	u4.test	1.1730543877181654	5433
u5.base	u5.test	1.1686585291940668	5217
ua.base	ua.test	1.2008035300836668	5214
ub.base	ub.test	1.2134460078406009	5083

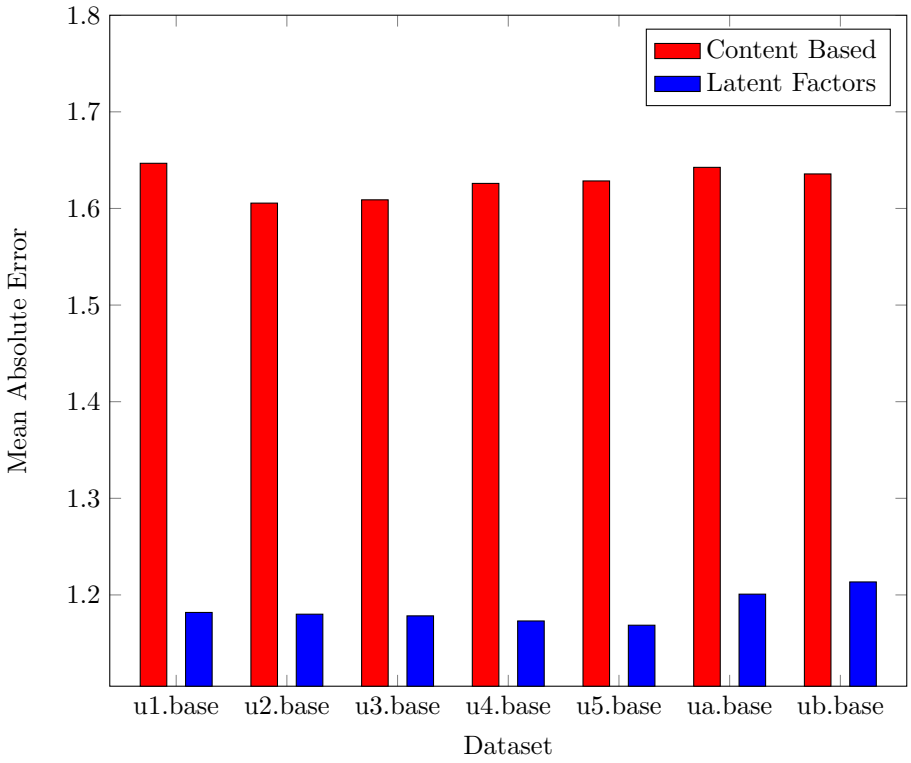

Figure 5: **Latent Factors vs Content Based on Mean Absolute Value**

6 Conclusion

As a conclusion we can see that als is better on both metrics from the content based.

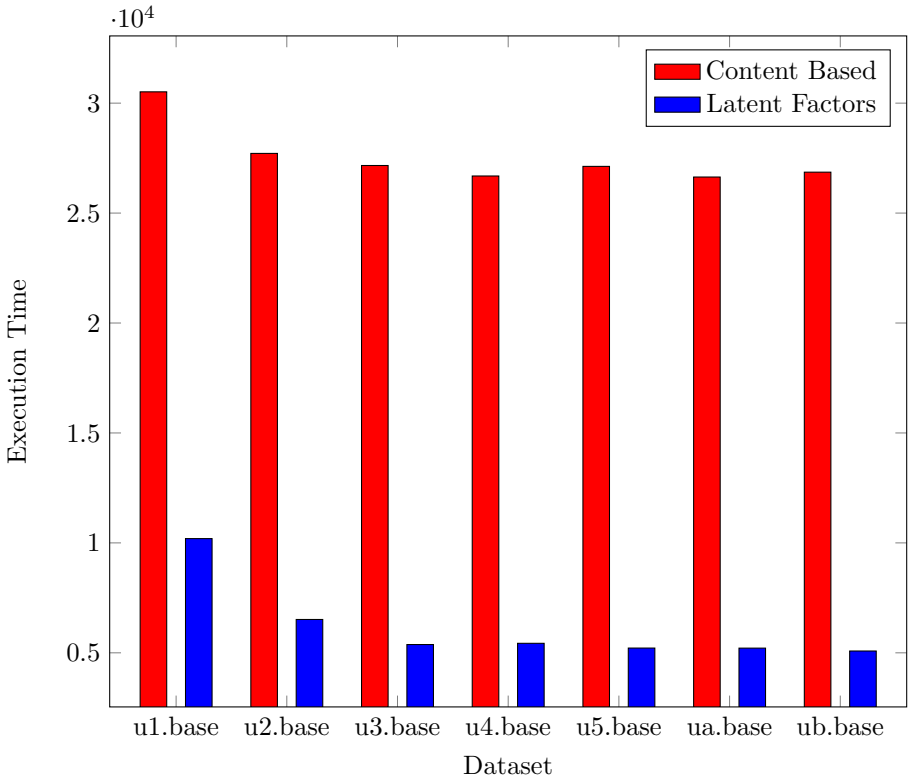

Figure 6: **Latent Factors vs Content Based on Execution Time**

References

[1] P. Melville and V. Sindhwani, "Recommender systems," *Encyclopedia of Machine Learning and Data Mining*, pp. 1056–1066, 2017.

[2] 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.

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

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

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

[6] "Databricks apache spark." `https://databricks.com/spark/about`. Accessed: 2017-06-24.

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

[8] "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 [7] 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()
  }
}
```

B Metrics

B.1 What is the mean absolute error

B.2 Time

List of Tables

1 **Content Based Algorithm Results** 8
2 **Latent Factors Algorithm Results** 8

List of Figures

1 **Hadoop Jobs Order** 4
2 **Data Information Knowledge Wisdom Pyramid [4]** 5
3 **Hadoup Software Stack** 5
4 **Apache spark stack [7]** 6
5 **Latent Factors vs Content Based on Mean Absolute Value** 9
6 **Latent Factors vs Content Based on Execution Time** . . 10