Master Thesis
Recommender Systems Comparison

Vasileios Symeonidis

27-05-2017

Contents

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]

History, what has been tried so far?

2 Related Work

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

spacing

label

In content based recommender systems we try to recommend 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.

$$w = R^{-1}M^T \quad (1)$$

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

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

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]

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

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

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

4.1.1 Apache Spark

The last decade, analyzing big data is at its peek. Lots of data are produced and the need for getting information from them is raised. The most common technique to do this is map-reduce.

Spark's predecessor, hadoop map reduce, was for a long time at its peak. 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 HDFS (hadoop distributed file system). When a job was scheduled, data were loaded by the hdfs to a worker, then the worker was putting the result back to the hdfs. Map-reduce is a method that is around a lot time for handling large amounts of data. It 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.

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."

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.

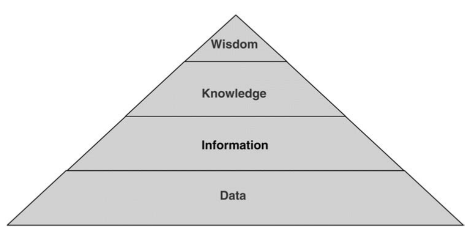

Figure 1: **Data Information Knowledge Wisdom Pyramid [4]**

Hadoop was the core map-reduce framework the last years. As it is described in [5], and shown in the figure 2 hadoop uses hadoop yarn in order to coordinate which process will run on which machine. Also it uses the HDFS (Hadoop Distributed File System) 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.

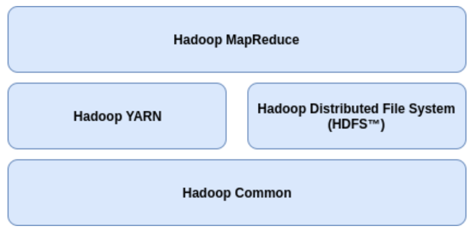

Figure 2: **Hadoop Software Stack**

In 2009 UC Berkley developed spark [6]. Spark's predecessor, hadoop map reduce, was for a long time at its peak. 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 HDFS (hadoop distributed file system). When a job was scheduled, data were loaded by the hdfs to a worker, then the worker was putting the result back to the hdfs.

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.

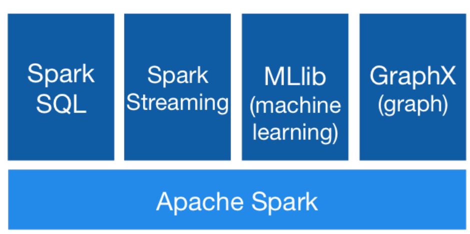

Figure 3: **Apache spark stack** [7]

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.

broadcasting rdd
//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. [8]

4.3 Implementation and assumptions

4.4 Metrics

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.

5 Results

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

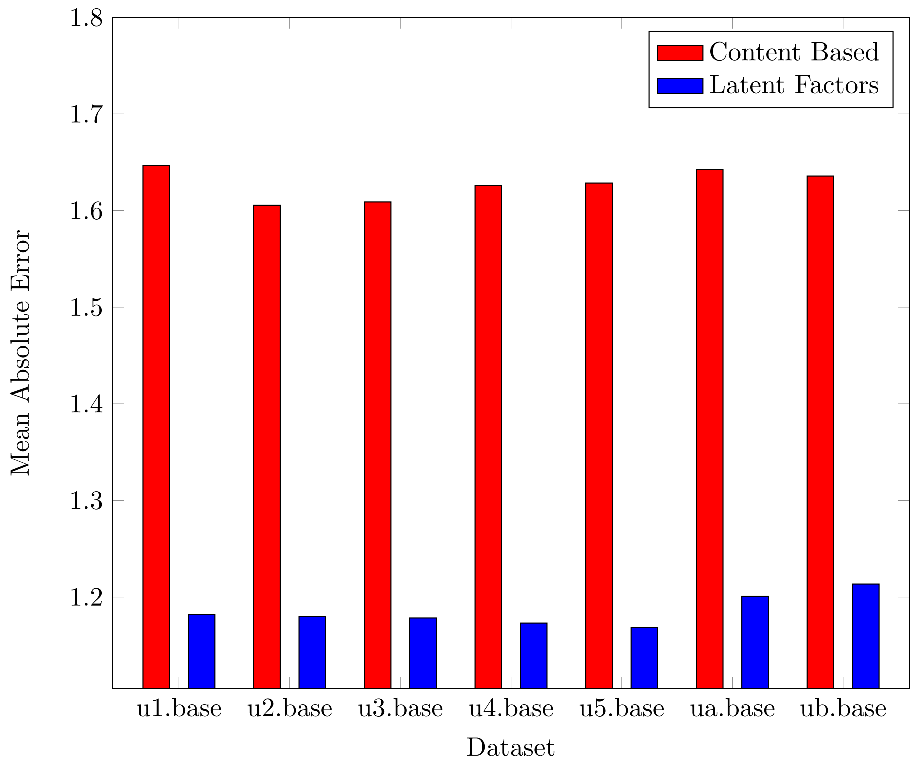

Figure 4: **Latent Factors vs Content Based on Mean Absolute Value**

6 Conclusion

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

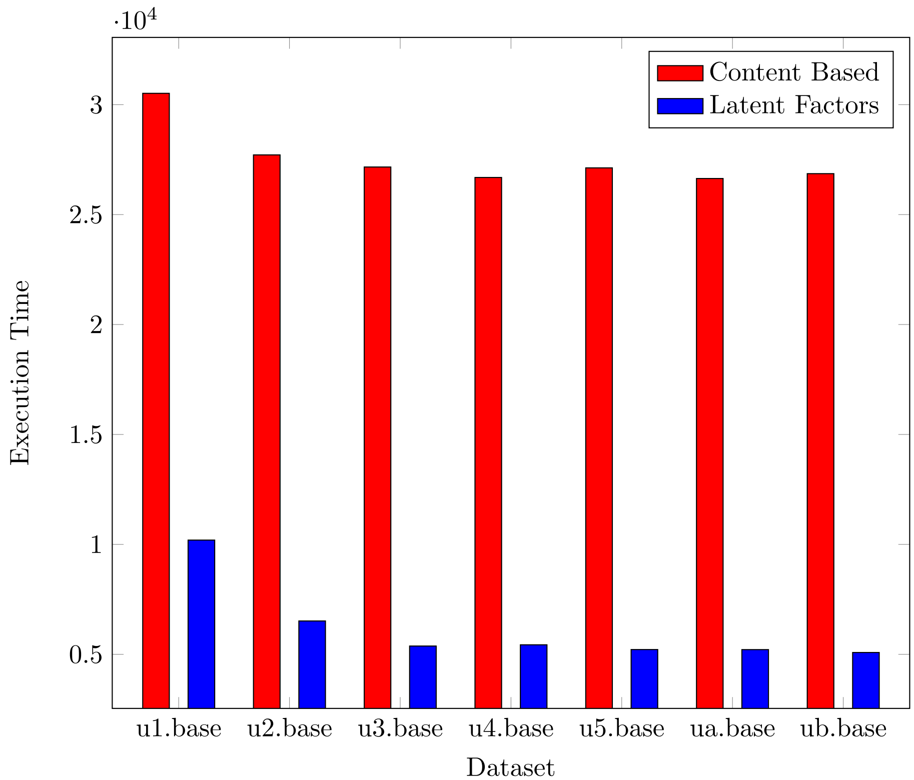

Figure 5: **Latent Factors vs Content Based on Execution Time**

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

A Code used

A.1 User Based Collaborative Filtering

A.2 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.3 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.4 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

List of Figures