Project Title:

Course Recommender System

Submitted by:

Neha Mahajan
Sayali Dongare
Monica Vaduguru

Instructor:

Prof. Ming Hwa Wang
Santa Clara University

Abstract:

Choosing a course from a large list of courses could prove to be a challenging job for students. To solve this problem in our project we construct a recommender system that is able to recommend courses to students.

Part of what makes the idea novel is the fact that we base our recommendations on the similarities of the students who are asking for the recommendations with the students who have taken up and rated courses earlier rather than basing the recommendations purely on a knowledge base. In the case of new students we will be using a knowledge base to provide our recommendations. We use a hybrid approach combining collaborative filtering and knowledge based techniques to provide our recommendations.

Acknowledgements:

We would like to thank our Professor Ming-Hwa Wang, Ph.D for providing us with the opportunity to choose a topic of our interest. This not only enabled us to pick a topic in a trending area in which we are interested in but also helped us in doing related research providing us with vast knowledge. We would also like to thank him for the guidance that he has provided for us to complete the project.

Table of contents:

List of figures/tables

2. Introduction

2.1 Objective:

The objective of this project is to be able to recommend courses based on the ratings provided by students who have taken the courses in the past. The recommendation is made keeping in mind the similarity between the student who is asking for the recommendation and the other students who have rated the courses along with the content of the course.

2.2 What is the problem?

It is quite a difficult task for a student to be able to take wise decisions about the courses he has to take up specially if he is a new student and has few contacts to refer to. Even if he did have the contacts it would be subjective to a few opinions, which might not ensure the quality of the decision. Moreover this process could result in students choosing courses that might not be totally suited to their liking as it is hard for them to assess the course based on a few opinions. Taking a course without much value to the student would mean monetary wastage as well as wastage of time both of which are very crucial resources to a student.

2.3 Why this is a project related to the class?

To be able to recommend courses based on the ratings provided by other students who have taken the courses earlier accurately would require the analysis and modification of huge amounts of data, depending on the scale of the problem that we undertake. This huge amount of data might otherwise be called as big data which is a collection of data sets that are so big that it is hard to collect, analyze, visualize and process using a single computer. We know that cloud computing is a perfect match for big data since it provides unlimited resources on demand and therefore this project has the scope to be deployed on cloud. Owing to these reasons this project is directly related to the present cloud-computing course.

2.4 Why other approach is no good?

The traditional approach to this problem would be to either base the decision on a word to word referral of courses by peers or to seek the advice of a guidance counselor. Both of these approaches are quite time consuming and might not be absolutely accurate as they are based on individual perspective.

2.5 Why you think your approach is better?

In the approach that we take we analyze a large data set, which includes all the courses offered along with their ratings, which are provided by the students. This approach is known as collaborative filtering technique, which is best suited to make our recommendations in this case as it also includes the students priorities.. We could also include a hybrid approach and include content based filtering (which depends on the content of the course too) which could further improve the accuracy of the recommendation. For new students who have not taken any courses before, including a knowledge based approach to the existing model would further be beneficial .

2.6 Statement of the problem

Design a course recommender system for students by recommending the courses that best fit their talents and program of study for upcoming quarters/semesters.

2.7 Area or scope of investigation

We will create a system that, through a simple interface, which would provide information to students and advisors to improve the choice of courses and consequently improve degree progress and completion. The system will be based on hadoop and in-turn map-reduce paradigm. The major efforts will be paid on developing an algorithm based on a collaborative filtering method.

3. Theoretical Bases and Literature Review

3.1 Definition of the problem

Many students, especially first-generation and non-traditional students, find significant challenges in making well informed decisions about which courses would best help them progress successfully through their degree program. Our system manages to provide a list of recommendations for such students. The recommendations will be based on a knowledge base of different courses, their level of difficulty, specialization tracks, and combinations required (like prerequisites, mandatory courses, and so on). The knowledge base would be

generated based on survey, experience and standard available information at Santa Clara University. This system would be a helpful tool for incoming students by suggesting to them what would be an ideal choice of courses for a term. The student would input his preferences to the system based on the following criteria:

1. The department
2. The concentration track
3. Student's interests
4. Quarter (Fall, Spring etc)
5. Prerequisites/Waivers
6. Skill set
7. Level of difficulty
8. Type of courses (Project course, Classroom course)
9. No of assignments, grading policy etc
10. Course Type (Introductory, Advanced, Research Oriented etc)
11. Student schedules
12. Part-time/Full-time student

3.2 Theoretical background of the problem

Recommender systems have changed the way people find products, information, and even other people. They study patterns of behavior to know what someone will prefer from among a collection of things he has never experienced. The technology behind recommender systems has evolved over the past 20 years into a rich collection of tools that enable the practitioner or researcher to develop effective recommenders.
This same technology is applicable for recommending courses to new students. There are number of proven algorithms to address the problem. The algorithms are based on collaborative filtering, content based filtering, knowledge based filtering and even hybrid approach is used. The base of many methods is information retrieval.

Secondly, the data in system can be processed in parallel using hadoop nodes and hence map reduce. Map Reduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. A typical Map Reduce computation processes many terabytes of data on thousands of machines. The runtime system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing the required inter-machine

communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

3.3 Our Solution to solve this problem

We have defined three milestones to create a efficient and robust course recommender system. In the first milestone, our recommender system will predict recommendations based on user-based collaborative filtering technique. This model will make predictions based on student's rating for different courses. In the second milestone, recommender system will support content collaborative and knowledge based filtering technique, which is a model, based on characteristics of an items like in our case, it will be subjective opinions of students for different courses. Later, modifications will be done to support hybrid recommender system.

3.4 Uniqueness of Solution

Our Solution is different and better than any other recommender systems since there exists no well defined system which can process large amounts of data, which helps us to take the leverage and build a system which can process it. We are building our system in Scala, which is highly used these days for fast big data analytics. Scala is an object-functional programming language, specifically designed to provide concise and elegant solutions. Scala is a blend of "scalable" and "language" specifically designed to meet the growing demands of users.

4. Project Goals.

Our project goal is to build a course recommender system based on different dimensions. As explained earlier, Our initial goal is to build course recommender system based on user-based collaborative filtering technique. Later modifications will be done to support content-based and hybrid-filtering techniques along with running the code parallelly using map reduce. This recommender system will help the students to make the right selection of courses and hence make a better career path.

5. Methodology

5.1 How to generate or collect input data:

In our present project we will be creating a static sample input data which contains few users along with the ratings for the courses that they have taken up in the past in a csv file. In future we could think of implementing a dynamic database for a university as a whole which could add rating as required.

5.2 How to solve the problem:

5.2.1 Algorithm design:
We have designed the architecture for the implementation of this project on scala and have represented this in the diagram below :
Each of the modules along with the functionality are explained below:

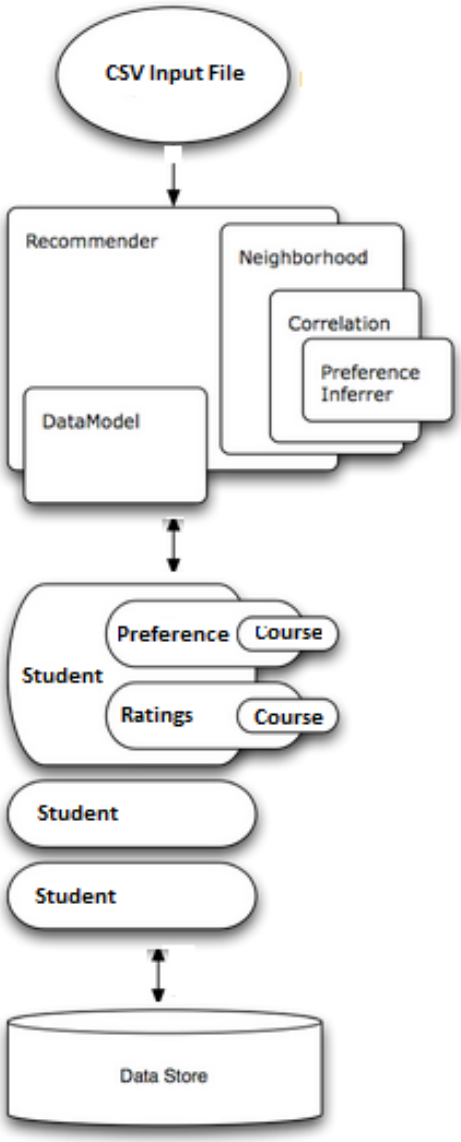

Figure 1: System Architecture

Data store: The input is created in the form of a csv file which is taken as an input to the program. This input file is parsed to be able to determine the student names, ratings and courses that they have rated and stored in the data store in the form of data structures. The input preference for the current student who needs the courses to be recommended is also taken as an input.

Preference inferrer:
This module allows us to understand the kind of courses that the user is looking for.

Correlation module:
This provides some notion of how similar two students are based on their ratings and preferences. We would be using the Pearson or any similar correlation model to implement this.

Neighborhood:
This module defines a notion of a group of students that are most similar to a given student. We will be using the kth-nearest neighbor algorithm.

Recommender:
This module pulls all these components together to recommend courses to students based on collaborative filtering techniques. We will use map reduce (Hadoop) to parallelize the computations in case of using big data.

5.2.2 Language used

Scala:
The language that we have chosen to implement this project is Scala. Scala is an acronym for "Scalable Language" which means that it grows with us. It is an object functional programming and scripting language.

Java :
General-purpose, concurrent, class-based, object-oriented computer programming language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture.

5.2.3 Tools used

The tools used are specified below:

JVM:
Scala runs on the Java platform(Java Virtual Machine) and is compatible with existing Java programs.
Eclipse: We will use Eclipse as our Integrated Development Environment (IDE).

Hadoop:
We will be using Hadoop in order to run our map reduce jobs.

5.3 How to generate output

Once the sequential program is executed and the jobs are run on hadoop, the recommender system provides the student with a list of recommendations as the output.

6. Implementation

6.1 Code:

Our project consists of two parts. The first part is being able to recommend courses to a new user who has no history of taking and rating courses before. For this we have written a code to be run on map reduce to find the 5 top rated courses that can be recommended to the new user which serves as a knowledge base. The second part is being able to recommend courses to a user who has taken a few courses and based on the similarity of the users choices with the other users as per the ratings provided courses are being recommended. This has been implemented in a scala code. We have integrated both the parts to work as one combined recommender system using knowledge based and collaborative filtering techniques.

The entire source code pertaining to the project can be found at the appendix section of this document.

6.2 Design document and flow chart:

We have prepared a flowchart, which would explain the flow of the program execution i.e how the inputs are provided to the program and how the output is expected.
The input file courses.csv has the list of all the courses offered in the format of (Course id , Student name , department)
The input file users.csv has the list of all the students along with their ids and the courses that they have taken in the past. The format is shown here (Student id ,Name , Course name)

The input file Ratings.csv has the list of students and the courses taken along with the ratings that they have given to each course. Format is as shown here(Student id, Course id, Rating)
The top five rated courses file in an intermediate file which is the output of the mapreduce program built to produce the top 5 best courses on the basis of the ratings which is given as an input to the recommender.

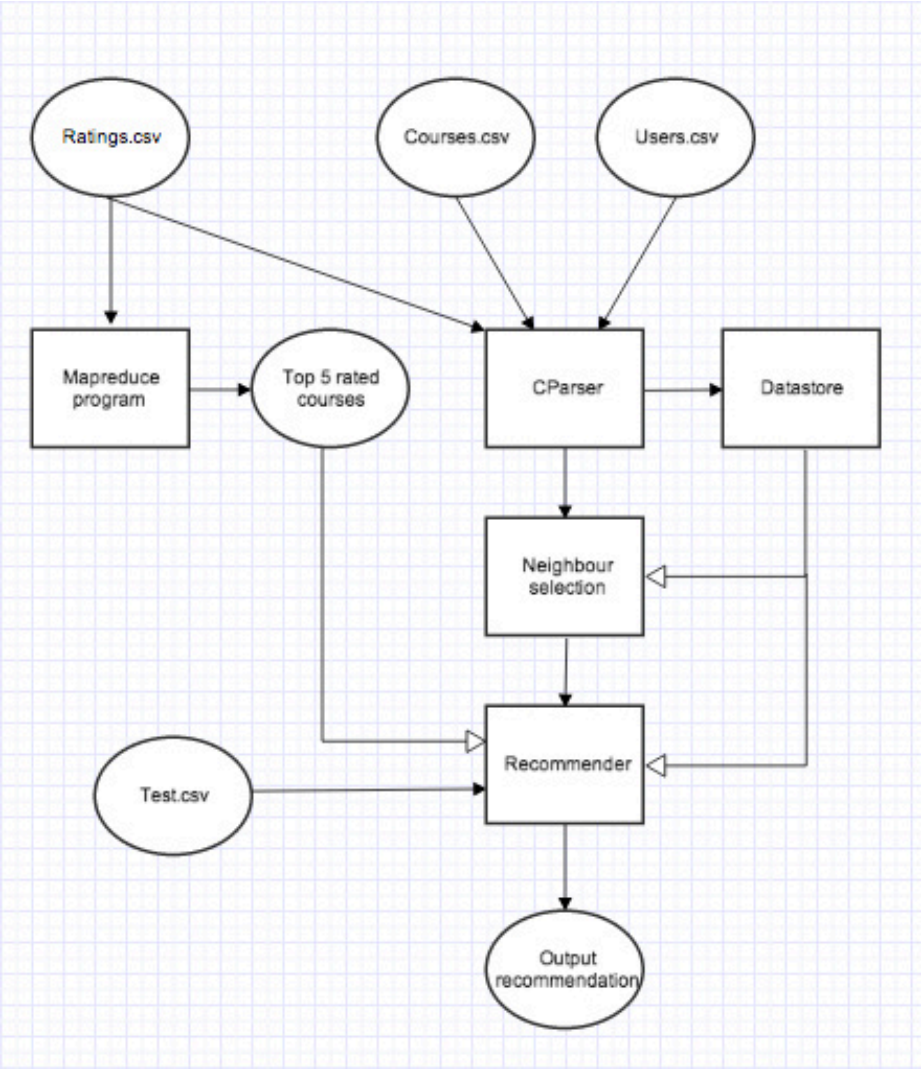

Fig 2:Design flow

The cParser module parses the inputs files into a datastore. The neighbor selection module is designed to use Pearson correlation to determine the top two neighbours for a given user. The recommender module takes the input in the form of Test.csv which contains the student id for whom the recommendation is required and provides recommendations to this user as the output.

7. Data analysis and discussion

7.1 Output generation and analysis:

For a new user the top 5 best-rated courses are recommended. However in order to provide recommendations for a user who has taken courses before first the top two neighbors are chosen based on the highest correlation. Correlation is calculated only if there are at least 2 common courses between the user and the student we are calculating correlation for. Correlation is based on two criteria and:

- Similarity of the ratings that two users provide for the common courses.
- The number of common courses between user and the rest.

After choosing the top 2 neighbors for the user, the user is recommended the courses that the neighbors have taken which he has not. We can explain this with a simple example.
Say there are 4 users A ,B, C, and D who have each taken courses (101,103,105) (101,102,103,) (102,103,104) and (102,105). These courses have been given ratings by these users as shown in the below table:

	101	102	103	104	105
A	7		6		9
B	6	5	7		8
C		6	8	7	
D		5			7

Fig 3 : Ratings and courses

Here when we ask for a recommendation for a new student E the top 5 rated courses are recommended. In this case since there are only five all 5 are recommended in the order of the average rating, first being course 105 in this case. When recommendation is asked for user A, we can see by observation that the pattern in which A chooses courses is similar to B. So by hypothesis student

7.2 Compare output against hypothesis:

We have run the same as inputs for the program that we have designed and asked for a recommendation for a new user E. The output is given as 105,104,103,101,102 which is as expected.
Also we checked for the recommendation for user 101 and the output we got was course 102 as expected since the user B will be a neighbor and course 102 is the only course user B has takes which A has not.

7.3 Abnormal case Explanation
We have encountered two abnormal cases in our implementation.

Case 1: User has provided same rating for all the courses he has taken. In such a case the pearson correlation cannot be calculated as the standard deviation is 0.
Case 2: User has no neighbors as there are no common courses taken or atmost one common course.

For both these cases we recommend the top 5 courses and the user has to pick and choose the courses, which he hasn't already taken from these.

8. Conclusions and recommendations

8.1 Summary and Conclusions:

A recommender system which is able to recommend courses for students would be helpful for students to take wise and well informed decisions while choosing their courses. For our project we have successfully implemented a recommender system for recommending courses given a dataset of ratings provided by various users. This system could provide recommendations for both new users as well as users who have already taken a few courses . We have written a code in scala to implement this recommender system using a hybrid approach which uses collaborative and knowledge based techniques. Given a huge list of courses and users it would be very beneficial to implement the recommender system on a paradigm like mapreduce which would substantially decrease the time taken for the computations by tapping into the parallelism mechanism. To start with we have implemented a mapreduce code to obtain the top 5 best rated courses of the given list which could be used to recommend courses for a new user. This output has been used as a knowledge base for the recommender system that we have built using collaborative filtering making it a hybrid approach.

8.2:Recommendations for future studies:

Furthermore going beyond the scope of the project enhancements can be made to make the implementation interactive and take the user preferences as well. Examples of such preferences could be the user mentioning the department he would like to take the courses for ,checking the complexity of the course, see if he has the prerequisites required for the course recommended or check to see how many other users are taking up the course and the list could go on. Also presently we are using hard coded inputs as the inputs of the program. We can enhance this recommender system to work on a large scale with live data by integrating it on top of existing websites designed for taking user ratings. Finally we could try to improve the performance by making the time taking calculations like finding correlation between each user run in a parallel manner and also in a progressive manner i.e by looking at only new ratings that have been added rather that going through all the calculations each time.

9. Bibliography

1. A Scala Tutorial for Java programmers by Michel Schinz, Phillipp Haller
2. Big Data Processing in Cloud Computing Environments by Changqing Ji, Yu Li, Wenming Qiu, Uchechukwu Awada, Keqiu Li
3. Contextual Recommendation based on Text Mining by Yize Li, Jiazhong Nie, Yi Zhang, BingqingWang
4. Introducing Serendipity in a Content-based Recommender System by Leo Iaquinta, Marco de Gemmis, Pasquale Lops, Giovanni Semeraro, Michele Filannino, Piero Molino
5. Content-based Recommen der Systems: State of the Art and Trends by Pasquale Lops, Marco de Gemmis and Giovanni Semeraro
6. http://www.slideshare.net/xlvector/recommender-system-algorithm-and-architecture-13098396 -
7. http://en.wikipedia.org/wiki/Recommender_system
8. ItemBased Collaborative Filtering Recommendation Algorithms, Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl.
9. An Optimized Item-based Collaborative Filtering Recommendation Algorithm, Jinbo Zhang, Zhiqing Lin, Bo Xiao, Chuang Zhang.
10. Knowledge based recommender system http://wenku.baidu.com/view/496d5b150b4e767f5acfce21.html?from=related
11. Hybrid Recommender Systems: Survey and Experiments,Robin Burke http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.8200&rep=rep1&type=pdf
12. Manning:Mahout in action pdf

10.Appendix

We have provided the source code for the project below:

10.1 Mapreduce Code:

10.1.1 Average.java

```
//Program: Map-reduce job to calculate average of all courses
```

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

// Class average
public class average{

    // Class Map to implement map functionalities
  public static class MapAvg extends MapReduceBase implements
Mapper<LongWritable, Text, Text, FloatWritable> {
      private Text course = new Text();

      // Function map splits the input line
      public void map(LongWritable key, Text value, OutputCollector<Text,
FloatWritable> output, Reporter reporter) throws IOException {
           String line = value.toString();
           String split[]= line.split(",");

           course.set(split[1]);
           Float rat = Float.parseFloat(split[2]);

           // Store the result in output collector
           output.collect(course,new FloatWritable(rat));

    }
  }

     // Class reduce to implement reduce functionalities
  public static class ReduceAvg extends MapReduceBase implements
Reducer<Text, FloatWritable, Text, FloatWritable> {

      // Function reduce calculates the average for each course
    public void reduce(Text key, Iterator<FloatWritable> values,
OutputCollector<Text, FloatWritable> output, Reporter reporter) throws
IOException {
      float sum = 0;
      float count = 0;
      float average = 0;
      while (values.hasNext()) {
        sum += values.next().get();
         count ++;
      }
      average = sum/count;
      // Store the average in output collector
```

```
      output.collect(key, new FloatWritable(average));
    }
  }

    // Function Main
  public static void main(String[] args) throws Exception {

      // Set up the job configurations
    JobConf conf = new JobConf(average.class);
    conf.setJobName("average");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(FloatWritable.class);

    conf.setMapperClass(MapAvg.class);;
    conf.setReducerClass(ReduceAvg.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }
}
```

10.1.2 Sorting.java:

```
//Program: Map-reduce program to sort the ratings in descending order

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
import java.io.*;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
```

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class sorting{

    // Class map includes map functionalities
    public static class Map extends MapReduceBase implements
Mapper<LongWritable, Text, FloatWritable, Text> {

    private Text course = new Text();
    private float minus = -1 * 1;

    //Function map splits the input text into word tokens
    public void map(LongWritable key, Text value,
OutputCollector<FloatWritable, Text> output, Reporter reporter) throws
IOException {
         String line = value.toString();
         String split[]= line.split("\t+");

         course.set(split[0]);
         float rat = Float.parseFloat(split[1])* minus;
         //Store the output in the output collector
         output.collect(new FloatWritable(rat), course);
    }
  }

    // Class reduce includes reduce functionalities
   public static class Reduce extends MapReduceBase implements
Reducer<FloatWritable, Text, Text, FloatWritable> {
       private float minus = -1 * 1;

       //Function reduce collects the output in descending format
       public void reduce(FloatWritable key, Iterator<Text> value,
OutputCollector<Text, FloatWritable> output, Reporter reporter) throws
IOException {
            float var;
            int count = 0;
            while(value.hasNext())
                {
             var = key.get();
             var = var * minus;
             output.collect(value.next(),new FloatWritable(var));
                }
     }
  }

  // Function Main
```

```
public static void main(String[] args) throws Exception {

    // Set the job configurations
  JobConf conf = new JobConf(sorting.class);
  conf.setJobName("sorting");

  conf.setOutputKeyClass(FloatWritable.class);
  conf.setOutputValueClass(Text.class);

  conf.setMapperClass(Map.class);
  conf.setReducerClass(Reduce.class);

  conf.setInputFormat(TextInputFormat.class);
  conf.setOutputFormat(TextOutputFormat.class);

  FileInputFormat.setInputPaths(conf, new Path(args[0]));
  FileOutputFormat.setOutputPath(conf, new Path(args[1]));

  JobClient.runJob(conf);

  // Read the hadoop file into a local file
  Configuration conf1 = new Configuration();
  conf1.addResource(new Path("/opt/hadoop-0.20.0/conf/core-site.xml"));
  conf1.addResource(new Path("/opt/hadoop-0.20.0/conf/hdfs-site.xml"));
  FileSystem fileSystem = FileSystem.get(conf);

  //Set the path name files
  String filename = args[1]+"/part-00000";
  Path path = new Path(filename);

  FSDataInputStream in = fileSystem.open(path);

  BufferedReader br = new BufferedReader(new InputStreamReader(in));

  String strLine;
  int count = 0;
  File output = new File("output");
  PrintWriter printer = new PrintWriter(output);
  byte[] b = new byte[1024];
  int numBytes = 0;

  // Read the top five courses
  while((strLine = br.readLine()) != null && count < 5)
       {
           printer.write(strLine);
           printer.write("\n");
           count ++;
       }

  // Close the files
```

```
    in.close();
    printer.close();

  }
}
```

10.2 Scala code

10.2.1 CourseRecom.scala:

```
object CourseRecom extends App {
        //Defines the flow of the program
val defaultPipe = List (
      ("CParser" -> CParser),
    ("NeighborSelection" -> NeighborSelection),
    ("Recommender" -> Recommender)
   )


      defaultPipe.takeWhile{ case (pipeName, task) =>
      task.execute()
   }

}
```

10.2.2 CParse.scala:

```
//Class to parse the input files into datastore
import io.Source
import java.util.concurrent.atomic.AtomicInteger

case class Course(courseId: String, name: String, Specialization: String)
case class Rating(userId: String, courseId: String, rating: Double)
case class User(userId: String, name: String, course: String)
case class Neighbour(neighborId: String, similarity: Double)

object CParser extends Task {
      def execute() = {
            //println("Inside CParser");
//Imports the input files

            val ratingFileName = "Scala/resources/Ratings.csv"
            val usersFileName = "Scala/resources/users.csv"
            val coursesFileName = "Scala/resources/courses.csv"

            importUsers(usersFileName);
            importCourses(coursesFileName);
```

```
                importRatings(ratingFileName);
                true
        }

        def importUsers(file: String) {
  val lines = Source.fromFile(file, "utf-8").getLines().withFilter(!_.isEmpty)
  lines.map(parseUser).toList.foreach( user => {
    DataStore.users.put(user.userId, user)

    //Console.println("Import user " + user.userId)
  })
}

        def importCourses(file: String) {
                val lineCount = new AtomicInteger(0)

                val lines = Source.fromFile(file, "utf-8").getLines().withFilter(!_.isEmpty)

                // process in parallel each course
                lines.map(parseCourse).foreach { case course =>
                        DataStore.courses.put(course.courseId, course)
                        // TODO


                        //Console.println("finished course - count: " + lineCount.incrementAndGet())
                }
}

        def importRatings(file: String) {
  val count = new AtomicInteger(0)

  // load all ratings to memory to faster parallel processing
  val lines = Source.fromFile(file, "utf-8").getLines().withFilter(!_.isEmpty)

  lines.map(parseRating).foreach{ rating =>
    DataStore.registerRating(rating)

    count.incrementAndGet()

    //Console.println("finished rating - count: " + count.get)
  }

  //DataStore.calcUserAvgRating()
}

        def importTestRatings(file: String) {
```

```
                val lines = Source.fromFile(file, "utf-8").getLines().withFilter(!_.isEmpty)
                        lines.foreach(f => parseRating(f))

        }

        def parseCourse(courseLine: String) = {
                val course = courseLine.split(",")
                        Course(course(0), course(1), course(2))
        }

        def parseRating(ratingLine: String) = {
                val rating = ratingLine.trim.split(',')
                        Rating(rating(0), rating(1), rating(2).toDouble)
        }

        def parseUser(userLine: String) = {
                val user = userLine.split(',')
                        User(user(0), user(1), user(2))
        }

        def parseTestData(userLine: String) = {

        }
}
```

10.2.3 DataStore.scala:

```
import collection.mutable

import collection.immutable.HashMap


object DataStore {

  val users = mutable.HashMap[String, User]()
  val courses = mutable.HashMap[String, Course]()

  val userRatings = mutable.HashMap[String, Map[String, Double]]()
  val courseRatings = mutable.HashMap[String, Map[String, Double]]()
  val userNeighbours = mutable.HashMap[String, List[Neighbour]]()

  /**
  * Only work if called after all inserts happens
  */
  var avgRatingCourse: Map[String, Double] = _

        def registerRating(rating: Rating, setCourse: Boolean = true) {
```

```
  val userRatingMap = userRatings.getOrElse(rating.userId, HashMap[String, Double]())
  userRatings.update(rating.userId, userRatingMap.+( (rating.courseId, rating.rating)  ))

  val courseRatingMap = courseRatings.getOrElse(rating.courseId, HashMap[String, Double]())
  courseRatings.update(rating.courseId, courseRatingMap.+((rating.userId, rating.rating)))
 }

 def calcCourseAvgRating(): List[String] = {
  avgRatingCourse = courseRatings.map { case(course, ratings) =>
   course -> (ratings.map(_._2).sum/ratings.map(_._2).toList.length)
  }.toMap

 //println("Course Average Rating:")
        //avgRatingCourse.foreach(x=> print(x))

        val sortReco = avgRatingCourse.toList.sortBy(m => m._2).takeRight(2)
        val sortRecoList = sortReco.map(m => m._1)

        val topRecommendations = sortRecoList.map(x=> getCourseName(x)).toList

        topRecommendations
}

 def getCourseName(courseId: String): String =  {
   courses.contains(courseId) match {
    case true => {
                val name = courses.get(courseId).get.name
                name
    }
    case _ => null
   }
 }

 def getUserName(userId: String): String =  {
  users.contains(userId) match {
    case true => {
                 users.get(userId).get.name
    }
    case _ => null
   }
  }
}
```

10.2.4 NeighborSelection.scala

```
import scala.math._
```

```
case class CommonRating(userId1: String, userId2: String, candidate:
String, uRating: Double, oRating: Double)

object NeighborSelection extends Task {

  def execute() = {
      //println("Inside NeighborSelection");

      DataStore.users.keySet.foreach(user => {
    val neighbors = getUserNeighbours(user, 2) // numNeighbors)

    DataStore.userNeighbours.put(user, neighbors)
    //printList(neighbors)
      })

      true
  }

  def printList(args: List[_]): Unit = {
    args.foreach(println)
  }
  //Calculating the correlation only if the number of common courses is
greater than or equal to 2
  def pearsonSimilarity(ratingsInCommon: Iterable[CommonRating],
useAvgRating: Boolean = false): Double = {
     if(ratingsInCommon.size < 2) {
      return 0.0
     }

    var user1SumSquare = 0.0d
    var user2SumSquare = 0.0d
    var user1Sum = 0.0d
    var user2Sum = 0.0d
    var sumSquare = 0.0d

    var commonRatingsTotal = 0

    ratingsInCommon.foreach{ case CommonRating(userId1, userId2, _,
myRating, otherRating) =>
      commonRatingsTotal = commonRatingsTotal + 1
      //print("myrating" + myRating + "  otherRating" + otherRating)
      // Sum the squares
      user1SumSquare = user1SumSquare + pow(myRating, 2.0)
```

```
      user2SumSquare = user2SumSquare + pow(otherRating, 2.0)
      user1Sum = user1Sum + myRating
      user2Sum = user2Sum + otherRating

      // Sum the products
      sumSquare = sumSquare + (myRating * otherRating)

    }

    val user1SquareSum = pow(user1Sum, 2.0)
    val user2SquareSum = pow(user2Sum, 2.0)

    // Calculate Pearson Correlation score
    val numerator = (commonRatingsTotal * sumSquare) - (user1Sum *
user2Sum)
    val denominator = (sqrt((commonRatingsTotal * user1SumSquare) -
user1SquareSum) *
        sqrt((commonRatingsTotal * user2SumSquare) - user2SquareSum))
//If denominator is 0 returning 0 else returning the correlation //of
pearson correlation combined with the number of common //courses
    denominator match {
      case 0 => 0
      case _ => {
        val res = (1.0 + (numerator / denominator)) / 2.0
          //val res = (numerator / denominator)
        //println("commonRatingsTotal:" + commonRatingsTotal)
        //println("user2SumSquare" +  user2SumSquare)
        //println("user2SquareSum" +  user2SquareSum)
          //println("\nY: "+ sqrt((commonRatingsTotal * user2SumSquare) -
user2SquareSum))
        //Console.println("N:" + numerator +" D:"+  denominator + " n/d:" +
(numerator / denominator) + " Correlation: " + res)
        res * min(commonRatingsTotal/50.0, 1)
      }
    }
  }

  def getUserNeighbours(user: String, numNeighbors: Int) : List[Neighbour]
= {
    if(!DataStore.userRatings.contains(user)) {
      //return List()
    }
    //Console.println("original user: " + user)
    val myRatings = DataStore.userRatings(user)
```

```
    val users: Iterator[String] = DataStore.users.keysIterator

    // val neighbors = users.filter(user.!=(_))

   // neighbors.foreach(user => {
   //Console.println("user: " + user) })

    val neighbours =
users.filterNot(user.eq).filter(DataStore.userRatings.contains).map {
oUserId =>
      val commonItems = DataStore.userRatings(oUserId).view.filter(m =>
myRatings.contains(m._1))

      val commonRatings = commonItems.map{ case(courseId, oRating) =>
       CommonRating(user, oUserId, oUserId, myRatings.getOrElse(courseId,
0.0), oRating)
      }.toIterable

      Neighbour(oUserId, pearsonSimilarity(commonRatings))
    }.toList

    neighbours.sortBy(n => n.similarity).takeRight(numNeighbors)
  }

}
```

10.2.5 Recommender.scala:

```
import io.Source

import scala.math._

object Recommender extends Task {
 def execute() = {
        //println("Inside Recommender");
        //Takes input as test.csv
        val testFilename = "Scala/resources/test.csv"

        val testRatings = Source.fromFile(testFilename).getLines()
        testRatings.hasNext match {
          case true => {
            val userId = testRatings.next
            DataStore.users.contains(userId) match {
```

```
            case true => {
                                val recommendations = userBasedPrediction(userId)
                                val userName = DataStore.getUserName(userId)

                                println("Recommendations for user " + userName + ":")
                                if(recommendations.length != 0) {
                                        recommendations.foreach(x=>
println(DataStore.getCourseName(x)))
                                }
                                else {
                                        val topReco = topRecommendations()
                                  topReco.foreach(x=> println(DataStore.getCourseName(x)))
                                }
                                }
            case false => {
                                val topReco = topRecommendations()
                                println("Recommendations for New User:")
                                        topReco.foreach(x=> println(DataStore.getCourseName(x)))
                        }
          }
         }
         case false => println("No Input:")
        }

        true
 }

 def printList(args: List[String]): Unit = {
  args.foreach(println)
 }

        def userBasedPrediction(userId: String): List[String] = {

         val myRatings = DataStore.userRatings(userId)
         val stdDev = calculateStdDeviation(userId)

         stdDev match {
          case 0 => {
           val topReco = topRecommendations()
           topReco
          }
          case _ => {
           val neighbours = DataStore.userNeighbours(userId).filter(n => n.similarity > 0).map(n
=> n.neighborId -> n.similarity).toMap
            val recom = neighbours.map { neighbour =>
```

```
            val courses = DataStore.userRatings(neighbour._1).view.filter(m =>
!myRatings.contains(m._1)).map(n => n._1).toList

            courses
            }.toList.flatten.distinct
            recom
          }
         }

        }

        def calculateStdDeviation(userId: String): Double = {
          val myRatings = DataStore.userRatings(userId)
          val ratingsSum = myRatings.map(m => m._2).toList.sum
          val ratingsLength = myRatings.map(m => m._2).toList.length
          val ratingsAvg = ratingsSum/ratingsLength

          val diff = (myRatings.map(m => pow((m._2 - ratingsAvg), 2.0)).toList.sum)/ratingsSum

          sqrt(diff)
        }

        def parseRecommendations(reco: String) = {
          val courseId = reco.trim.split('\t')
          courseId(0)
        }

        def topRecommendations(): List[String] ={
          val filename = "output"
          val lines = Source.fromFile(filename, "utf-8").getLines().withFilter(!_.isEmpty)
          val topReco = lines.map(parseRecommendations).toList
          topReco
        }
}
```

Task.scala:

```
trait Task {
  def execute(): Boolean
}
```

10.3 Inputs for testing:

10.3.1 courses.csv

```
01,Operating system,Systems
```

02,Network Design,Networks
03,Algorithms,Core
04,Cloud Computing,Systems
05,Network Technology,Networks
06,Wireless and Mobile Networks,Networks
07,Computer Architecture,Core
08,Advanced DataBase,Elective
09,Web Search & Information Retrieval,Elective
10,Pattern Recognition,Elective

10.3.2 users.csv
101,alice,Operating system
101,alice,Network Design
101,alice,Wireless and Mobile Networks
101,alice,Algorithms
102,tom,Operating system
102,tom,Network Design
102,tom,Algorithms
102,tom,Cloud Computing
103,John,Operating system
103,John,Cloud Computing
104,Randy,Advanced Database
104,Randy,Computer Architecture
104,Randy,Operating system
105,Mindy,Cloud Computing
105,Mindy,Web Search & Information Retrieval
105,Mindy,Network Design
105,Mindy,Pattern Recognition
105,Mindy,Advanced DataBase
106,Grace,Wireless and Mobile Networks
106,Grace,Operating system
107,Catherine,Cloud Computing
107,Catherine,Web Search & Information Retrieval
107,Catherine,Operating system
108,Raymond,Operating system
108,Raymond,Web Search & Information Retrieval
109,Eva,Advanced DataBase
109,Eva,Operating system
109,Eva,Web Search & Information Retrieval
110,Percy,Cloud Computing
110,Percy,Web Search & Information Retrieval
110,Percy,Pattern Recognition
110,Percy,Operating system
111,Angelina,Wireless and Mobile Networks

111,Angelina,Web Search & Information Retrieval
112,Brad,Cloud Computing
112,Brad,Operating system
112,Brad,Pattern Recognition
112,Brad,Advanced DataBase
112,Brad,Web Search & Information Retrieval
113,Nina,Network Design
113,Nina,Advanced DataBase
113,Nina,Web Search & Information Retrieval
114,Lynette,Network Technology
114,Lynette,Pattern Recognition
114,Lynette,Computer Architecture
114,Lynette,Advanced DataBase
115,Colen,Wireless and Mobile Networks
115,Colen,Web Search & Information Retrieval
116,Pam,Cloud Computing
116,Pam,Computer Architecture
117,Andy,Advanced DataBase
117,Andy,Computer Architecture
117,Andy,Web Search & Information Retrieval
118,Ria,Network Technology
118,Ria,Pattern Recognition
118,Ria,Cloud Computing
119,Morty,Web Search & Information Retrieval
119,Morty,Pattern Recognition
120,Mike,Advanced DataBase
120,Mike,Cloud Computing
120,Mike,Web Search & Information Retrieval
120,Mike,Network Technology
120,Mike,Pattern Recognition

10.3.3 Ratings.csv
101,01,4.0
101,02,5.0
101,06,9.0
101,03,2.0
102,01,4.0
102,02,5.0
102,03,7.0
102,04,5.0
103,01,9.0
103,04,9.0
104,08,6.0
104,07,7.6

```
104,01,4.5
105,04,9.9
105,09,9.9
105,02,9.9
105,10,9.9
105,08,9.9
106,06,8.0
106,01,4.0
107,04,6.0
107,09,3.0
107,01,7.0
108,01,10.0
108,09,9.0
109,08,6.0
109,01,7.0
109,09,8.0
110,04,5.0
110,09,8.0
110,10,7.0
110,01,8.0
111,06,3.0
111,09,2.0
112,04,1.0
112,01,7.0
112,10,8.0
112,08,4.0
112,09,7.0
113,02,9.0
113,08,5.0
114,05,7.0
114,10,9.0
114,07,7.0
114,08,8.6
115,06,5.0
115,09,7.0
116,04,3.0
116,07,7.0
117,08,8.0
117,07,6.0
117,09,5.0
118,05,6.0
118,10,5.0
118,04,6.0
119,09,4.0
```

```
119,10,6.0
120,04,7.0
120,09,8.0
120,05,9.0
120,10,9.0
```

10.3.4 test.csv
115

10.4 Script

```
#!/bin/bash

echo "Loading Recommender System..."

javac -cp /usr/lib/hadoop/*:/usr/lib/hadoop/client-0.20/* -d ./classes
average.java
javac -cp /usr/lib/hadoop/*:/usr/lib/hadoop/client-0.20/* -d ./classes
sorting.java
jar -cvf average.jar -C ./classes/ .
jar -cvf sorting.jar -C ./classes/ .

hadoop fs -mkdir /user/cloudera/cloud/
hadoop fs -mkdir /user/cloudera/cloud/inputAvg0/
hadoop fs -put ./Scala/resources/Ratings.csv
/user/cloudera/cloud/inputAvg0/

hadoop jar average.jar average /user/cloudera/cloud/inputAvg0
/user/cloudera/cloud/outputAvg0

echo "Calculating average completed.."

hadoop fs -mkdir /user/cloudera/cloud/inputSort0

hadoop fs -cp /user/cloudera/cloud/outputAvg0/part-00000
/user/cloudera/cloud/inputSort0/

hadoop jar sorting.jar sorting /user/cloudera/cloud/inputSort0
/user/cloudera/cloud/outputSort0

hadoop fs -rm -r /user/cloudera/cloud/

rm -rf scalaClasses
mkdir scalaClasses
export PATH=/home/cloudera/cloud/scala-2.9.3/bin:$PATH
```

```
scalac -cp scala-2.9.3/lib -d scalaClasses/ Scala/src/*.scala
scala -cp scalaClasses/ CourseRecom

echo "Program ends.."
```