Introduction to Big Data.
Assignment Report.

Spark Wars: Lada Morozova, Danis Alukaev, Denis Schegletov

Introduction.

The goal of the assignment was to finalize the implementation of a movie recommendation system in Scala. Moreover, it was required to use Spark framework and implement it not only on local Hadoop cluster but also on Private Network Hadoop cluster.
To finalize implementation, team implement `parseTitle` and `rmse` functions; post-processing of recommendations; method that allow to load user data directly instead of surveying user. Moreover, the team will run the application with different model ranks, compare and evaluate the differences between baseline and prediction for the model with a higher rank. In addition, low-frequency items will be excluded from the training and recommendation process.

Implementation of functions

parseTitle

The function `parseTitle` is used for splitting film entry and extracting from it the title of the film.
Implementation of **parseTitle** from the **MovieLensALS.scala**:

```
def parseTitle(filmEntry: String) = {
    // Task 0.1. Complete the code
    "\".*\"".r findFirstIn filmEntry match {
      case Some(s) => s.replace("\"", "")
      case None => filmEntry.split(",").map(_.trim).lift(1)
    }
  }
```

rmse

The function `rmse` used for calculating root-mean-square error between ground truth ratings of the film and the ratings predicted by the system. The root-mean-square error is calculated as $\sqrt{\frac{\sum_{i=1}^{N}(x_i-\hat{x}_i)^2}{N}}$.
Implementation of **RMSE** from the **MovieLensALS.scala**:

```
 def rmse(test: RDD[Rating], prediction: scala.collection.Map[Int, Double]) = {
    // Task 0.2. Complete the code
    val rating_pairs = test
      .map(x => (x.rating, prediction.getOrElse(x.product, 0.0)))
    math.sqrt(rating_pairs
      .map(x => (x._1 - x._2) * (x._1 - x._2))
      .reduce(_ + _) / test.count())
  }
```

Post-processing of recommendations

The goal of the recommender system is to present to users 20 movies that can be considered as most suitable for following users based on his/her ratings of other movies. To do so, the prediction based on the user's rating are made, then watched movies are excluded, and 20 movies with the highest ratings are presented to the user. The following code snippet from **MovieLensALS.scala** depicts our solution for the following task:

```
if (doGrading) {
      println("Predictions for user\n")

      // Task 1. Post-processing
      // collect all movie ids in user rating
      val filmIds = sc.textFile(ratingsPath + "/user_rating.tsv")
        .map(line => line.split("\t"))
        .map(x => x(0).toInt)
        .collect()

      // for input data format refer to documentation
      // get movie ids to rank from baseline map
     model.predict(sc.parallelize(baseline.keys.map(x => (0, x)).toSeq))
        // Task 1. Post-processing
        // filter watched movies
        .filter(x => !filmIds.contains(x.product))
        // sort by ratings in descending order
        .sortBy(_.rating, false)
        // take first 20 items
        .take(20)
        // print title and predicted rating
        .foreach(x => println(s"${filmId2Title(x.product)}\t\t${x.rating}"))
    }
```

Data loading

To improve loading of user's ratings data the two methods was introduced: `loadPreferences()` and `predictPreferences()`. First method `loadPreferences()` allows to upload user's rating with use of .tsv file instead of surveying the user film by film. However, the ability to evaluate films one by one has been retained and has been moved to a separate function `predictPreferences()`. The implementation of `loadPreferences()` and `predictPreferences()` from **Grader.scala**

```
  // Task 2. Load movie preferences
  println("Do you want to load your preferences?")
  val answer = scala.io.StdIn.readLine()
  val answerPattern: Regex = "^(y|yes)$".r

  def loadPreferences() = {
    println("Loading from user_rating.tsv...")
    graded = sc.textFile(path + "/user_rating.tsv")
      .map(line => line.split("\t"))
      .map(x => (x(0).toInt, x(1).toDouble))
      .collect()
  }

  def predictPreferences() = {
    while (step < limit) {
      println(s"\nGraded ${step}/${limit}, Viewed ${position}/${films.length}")
      println(s"${grading_message}\n${films(position)._2}")
      var grade = scala.io.StdIn.readLine()

      try {
        var intGrade = grade.toDouble

        if (intGrade < 0.0 || intGrade > 5.0) {
          throw new NumberFormatException()
        }
```

```
        if (intGrade != 0) {
          graded = graded :+ (films(position)._1, intGrade)
          step += 1
          position += 1
        } else {
          position += 1
        }

        if (position == films.length) {
          step = limit + 1
          println("\nNo more movies to grade\n")
        }

        if (step == limit) {
          println("Grade 20 more? y/[n]")
          var ans = scala.io.StdIn.readLine()

          if (ans == "y" || ans == "yes" || ans == "Yes" || ans == "YES") {
            limit += 20
          } else {
            println("\nFinished Grading\n")
          }
        }
      } catch {
        case e: NumberFormatException => println("Try again")
        case e: Exception => println("Unknown Error")
      }
      dumpRatings()
    }
  }
```

Comparison of models

There is no significant difference in baseline error. The least training error was reached for rank of 2.

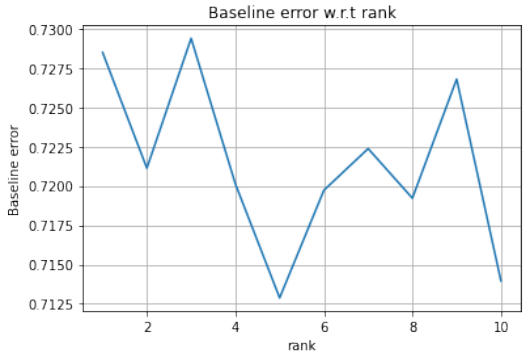

Figure 1

Figure 2

Elimination of low-frequency items

To improve the recommender system the extra filtering was introduced. Movies with a total score of less than 50 were excluded from the learning process, as well as from the recommendation process. The code snippet with elimination process from **MovieLensALS.scala**:

```
// Task 4. Extra Filtering
val films  = ratingsData.map(x => x.product).countByValue().filter(x => x._2 > 50).toArray.map(x=>x._1)
ratingsData = ratingsData.filter(x => films.contains(x.product))
// End of Task 4. Extra Filtering
```

Private Network Hadoop cluster

For the test purposes there was built cluster of two machines with Ubuntu 18.04 on the board. Often distributed computations involve several hosts with approximately equal resources, so it was decided to meet this requirement using Virtual Machines. As a provider of virtualization we have used the Virtualbox. Configuration files and source code is available online.
In Vagrantfile we configured the bridge to public network, so that the DHCP server in our private network assigns the IP address to Virtual Machine independently from the host. The IP address could be determined using hostname utility. However, we faces the difficulty with *InnopolisUniversity* network that does not assign IP addresses to Virtual Machines. Consequently, all the further actions were performed on Wi-Fi of our room.
On master node with IP address 192.168.0.194 we installed Hadoop 3.3.1 and Spark of version 3.2.0. The machine was configured and sample jobs runned. Then, all the settings were passed on the slave node with IP address 192.168.0.118. The former plays the role of Namenode, Secondary Namenode, Resource Manager, Datanode, and Worker, whereas the second - as Datanode and Worker. Figure 3 demonstrates the overall HDFS filesystem in the configured cluster.
The implemented earlier code was compiled using sbt package, and the submitted as a job on private network cluster. The sample output of the program is presented in Figure 4.

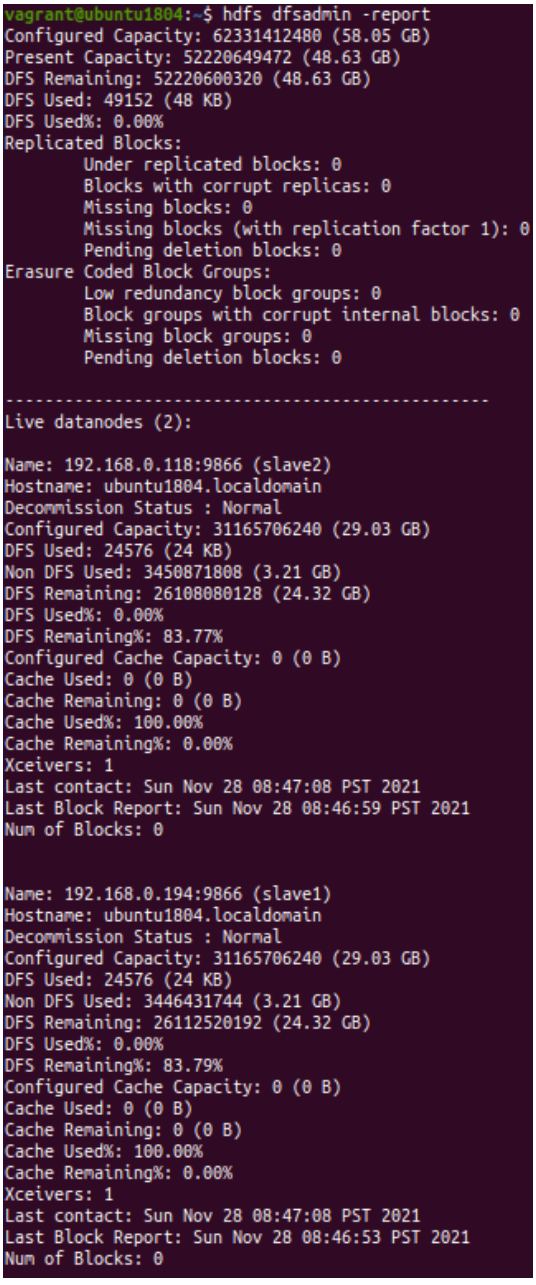

Figure 3: HDFS Filesystem

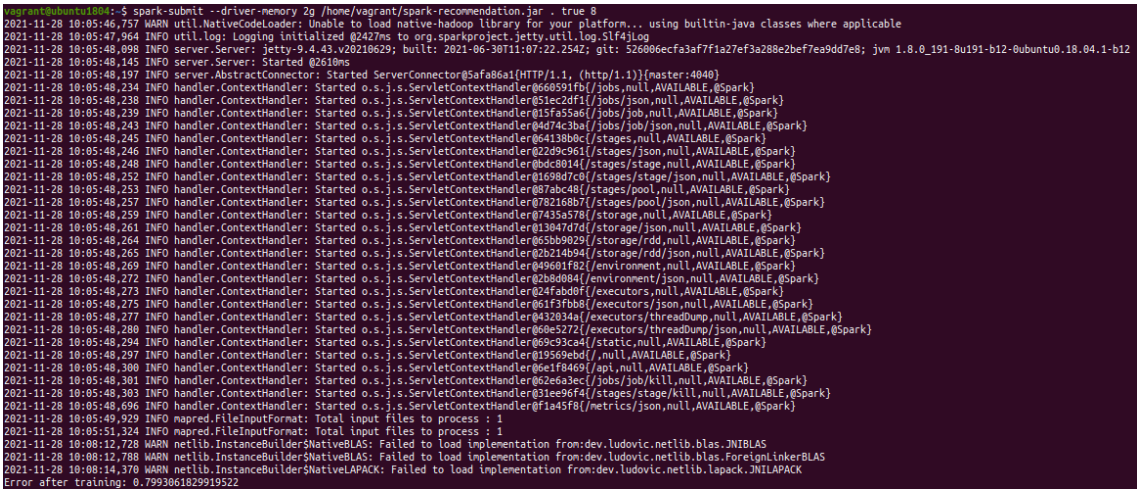

Figure 4: Recommender System run

Contribution of each member

The code was implemented collaboratively using Intellij Idea "Code with me" feature.

Danis Alukaev	Implementation of functions Implementation of data loading
Denis Schegletov	Implementation of post-processing of recommendation Elimination of low-frequency items
Lada Morozova	Composition of report Comparison of models

Conclusion

There was finished provided in specification template. For instance, we successfully added parse title and rmse fucntionality, post-processing of recommendations, data loading, and elimination of low-frequency items. In addition, the final version of the recommendation system was executed in private network Hadoop cluster. All the work was implemented collectively and tasks distributed among team members.