CME 323 HW 2

Charles Zheng

Problem 1.

Code:

```
import java.io.File

import scala.io.Source

import org.apache.log4j.Logger
import org.apache.log4j.Level

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
import org.apache.spark.mllib.recommendation.{ALS, Rating, MatrixFactorizationModel}

object MovieLensALS {

  def main(args: Array[String]) {

    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

    if (args.length != 2) {
      println("Usage: /path/to/spark/bin/spark-submit --driver-memory 2g --class MovieLensALS " +
        "target/scala-*/movielens-als-ssembly-*.jar movieLensHomeDir personalRatingsFile")
      sys.exit(1)
    }

    // set up environment

    val conf = new SparkConf()
      .setAppName("MovieLensALS")
      .set("spark.executor.memory", "2g")
    val sc = new SparkContext(conf)

    // load personal ratings

    val myRatings = loadRatings("/root/spark-training/machine-learning/bin/personalRatings.txt")
    val myRatingsRDD = sc.parallelize(myRatings, 1)

    // load ratings and movie titles

    val movieLensHomeDir = "/root/spark-training/data/movielens/large"

    val ratings = sc.textFile("ratings.dat", 50).map { line =>
  val fields = line.split("::")
  // format: (timestamp % 10, Rating(userId, movieId, rating))
  (fields(3).toLong % 10, Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble))
}

val movies_RDD = sc.textFile("movies.dat", 50).map { line =>
  val fields = line.split("::")
  // format: (movieId, movieName)
  (fields(0).toInt, fields(1))
}

val movies = movies_RDD.collect().toMap

// your code here
val numPartitions = 50
val training = ratings.filter(x => x._1 < 6)
  .values
  .union(myRatingsRDD)
  .repartition(numPartitions)
  .cache()
val validation = ratings.filter(x => x._1 >= 6 && x._1 < 8)
  .values
  .repartition(numPartitions)
  .cache()
val test = ratings.filter(x => x._1 >= 8).values.cache()

val numTraining = training.count()
val numValidation = validation.count()
val numTest = test.count()

println("Training: " + numTraining + ", validation: " + numValidation + ", test: " + numTest)

val ranks = List(8, 12)
val lambdas = List(1.0, 10.0)
val numIters = List(10, 20)
var bestModel: Option[MatrixFactorizationModel] = None
var bestValidationRmse = Double.MaxValue
var bestRank = 0
var bestLambda = -1.0
var bestNumIter = -1
for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
  val model = ALS.train(training, rank, numIter, lambda)
  val validationRmse = computeRmse(model, validation, numValidation)
  println("RMSE (validation) = " + validationRmse + " for the model trained with rank = "
    + rank + ", lambda = " + lambda + ", and numIter = " + numIter + ".")
  if (validationRmse < bestValidationRmse) {
    bestModel = Some(model)
    bestValidationRmse = validationRmse
    bestRank = rank
    bestLambda = lambda
    bestNumIter = numIter
  }
}
```

```
  val testRmse = computeRmse(bestModel.get, test, numTest)

  println("The best model was trained with rank = " + bestRank + " and lambda = " + bestLambda
    + ", and numIter = " + bestNumIter + ", and its RMSE on the test set is " + testRmse + ".")

  val myRatedMovieIds = myRatings.map(_.product).toSet
  val candidates = sc.parallelize(movies.keys.filter(!myRatedMovieIds.contains(_)).toSeq)
  val recommendations = bestModel.get
    .predict(candidates.map((0, _)))
    .collect()
    .sortBy(- _.rating)
    .take(50)

  var i = 1
  println("Movies recommended for you:")
  recommendations.foreach { r =>
    println("%2d".format(i) + ": " + movies(r.product))
    i += 1
  }

  // HW
  val model8 = ALS.train(training, 8, 20, 1.0)
  val model5 = ALS.train(training, 8, 20, 1.0)
  movies.map(_.swap).get("Saving Private Ryan (1998)") // 2028
  movies.map(_.swap).get("Alien (1979)") // 1214
  model8.productFeatures.lookup(2028)
   //Seq[Array[Double]] = WrappedArray(Array(0.6477546219058509, 0.6426229445743988, 0.6461808810337494,
   // 0.6431596287309431, 0.6384334161962797, 0.6415332046689981, 0.6505237147976703, 0.6504696994359823))
  model5.productFeatures.lookup(1214)
   //Seq[Array[Double]] = WrappedArray(Array(0.6373592762840713, 0.6368712379801962, 0.641377009770939,
   // 0.6368617654243005, 0.6435706137762942, 0.6424977510555535, 0.6381539251150117, 0.6366490114006595))

  // clean up
  sc.stop()
}

/** Compute RMSE (Root Mean Squared Error). */
def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: Long): Double = {
  val predictions: RDD[Rating] = model.predict(data.map(x => (x.user, x.product)))
  val predictionsAndRatings = predictions.map(x => ((x.user, x.product), x.rating))
    .join(data.map(x => ((x.user, x.product), x.rating)))
    .values
  math.sqrt(predictionsAndRatings.map(x => (x._1 - x._2) * (x._1 - x._2)).reduce(_ + _) / n)
}

/** Load ratings from file. */
def loadRatings(path: String): Seq[Rating] = {
  val lines = Source.fromFile(path).getLines()
  val ratings = lines.map { line =>
    val fields = line.split("::")
    Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
  }.filter(_.rating > 0.0)
```

```
    if (ratings.isEmpty) {
      sys.error("No ratings provided.")
    } else {
      ratings.toSeq
    }
  }
}
```

1a:

```
model8.productFeatures.lookup(2028)
 //Seq[Array[Double]] = WrappedArray(Array(0.6477546219058509, 0.6426229445743988, 0.6461808810337494,
 // 0.6431596287309431, 0.6384334161962797, 0.6415332046689981, 0.6505237147976703, 0.6504696994359823))
```

1b:

```
model5.productFeatures.lookup(1214)
 //Seq[Array[Double]] = WrappedArray(Array(0.6373592762840713, 0.6368712379801962, 0.641377009770939,
 // 0.6368617654243005, 0.6435706137762942, 0.6424977510555535, 0.6381539251150117, 0.6366490114006595))
```

Problem 2.

Code:

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

val articles: RDD[String] = sc.textFile("vertices.txt", 50)
val links: RDD[String] = sc.textFile("edges.txt", 50)

val vertices = articles.map { line =>
  val fields = line.split('\t')
  (fields(0).toLong, fields(1))
}

val edges = links.map { line =>
  val fields = line.split('\t')
  Edge(fields(0).toLong, fields(1).toLong, 0)
}

val graph = Graph(vertices, edges, "").cache()

val prGraph = graph.pageRank(0.001).cache()

val titleAndPrGraph = graph.outerJoinVertices(prGraph.vertices) {
  (v, title, rank) => (rank.getOrElse(0.0), title)
}

val top10 = titleAndPrGraph.vertices.top(10){
  Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)
}.foreach(t => println(t._2._2 + ": " + t._2._1))
```

```
val titleAndDegree = graph.outerJoinVertices(graph.inDegrees) {
  (v, title, rank) => (rank.getOrElse(0), title)
}

val top10d = titleAndDegree.vertices.top(10){
  Ordering.by((entry: (VertexId, (Int, String))) => entry._2._1)
}.foreach(t => println(t._2._2 + ": " + t._2._1))
```

2a.

```
//University of California, Berkeley: 1321.1117543121866
//Berkeley, California: 664.8841977233967
//Uc berkeley: 162.50132743398052
//Berkeley Software Distribution: 90.47860388486285
//Lawrence Berkeley National Laboratory: 81.9040493964213
//George Berkeley: 81.85226118458093
//Busby Berkeley: 47.87199821801988
//Berkeley Hills: 44.764069795199774
//Xander Berkeley: 30.3240753472881
//Berkeley County, South Carolina: 28.9083364837103
```

2b.

```
//
//University of California, Berkeley: 7387
//Berkeley, California: 3900
//Uc berkeley: 989
//Lawrence Berkeley National Laboratory: 438
//Berkeley Software Distribution: 407
//George Berkeley: 403
//Busby Berkeley: 232
//Berkeley, CA: 197
//Berkeley County, West Virginia: 172
//Xander Berkeley: 166
```

Problem 3.

Each vertex *i* has two variables: its distance to the source, *D* and its previous value for the distance, *D_old*. When *D = D_old* for all vertices, the algorithm terminates.

Single-source shortest path

1. (Initialization). For all vertices *i* other than the source, set *D[i] = Infinity* and *D_old[i] = Infinity*. For the source vertex *s*, set *D[s] = 0* and *D_old[s] = 0*. Send message *M[s] = 1* to all neighbors of *s*.
2. **For** all vertices *i*:
3. Set *D_old[i] = D*.
4. Set *D[i] = min M[j]* for all neighbors *j*
5. **If** *D[i] != D_old[i]*, **then** send *M[i] = D[i] + 1* to all neighbors of *i*
6. **EndFor**