iWisps - IBM Apache Spark Competition

Exploring Yelp dataset and building recommendation models

In [ ]:
```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.{DataFrame, Row, SQLContext}
import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationMo
del, Rating}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types.{IntegerType, StringType, StructField,
 StructType}

val sqlContext = new SQLContext(sc)
import sqlContext.implicits._

val csv = "com.databricks.spark.csv"
val csvOptions = Map("delimiter" -> "|", "header" -> "true", "inferSchem
a" -> "true")

val listSeparator = ";"
```

In [ ]:
```
//removed private credentials required to connect to the object store
```

In [ ]:
```
val businessFile = fs + "yelp_academic_dataset_business.json"
val userFile = fs + "yelp_academic_dataset_user.json"
val reviewFile = fs + "yelp_academic_dataset_review.json"

val modelFileName = fs + "simpleCfModel"
val usersFileName = fs + "users"
val businessesFileName = fs + "businesses"
val perBusiness = fs + "perBusiness"
val perUser = fs + "perUser"
val perCategory = fs + "perCategory"
```

For this demo only limit to city Edinburgh

In [ ]:
```
val scotBusinessDf = sqlContext.read.format("json").load(businessFile).w
here("city = 'Edinburgh'")
```

In [ ]:
```
scotBusinessDf.show(10)
```

In [ ]:
```
val reviewDf = sqlContext.read.format("json").load(reviewFile)
```

In [ ]:
```
reviewDf.show(10)
```

In [ ]:
```
val userDf = sqlContext.read.format("json").load(userFile)
```

In [ ]:
```
userDf.show(10)
```

In [ ]:
```
val scotReviewDf = scotBusinessDf.select("business_id").
    join(reviewDf.select("business_id", "user_id", "stars"), Seq("busine
ss_id"))
```

In [ ]:
```
def getUserWithIdDf(sqlContext: SQLContext, scotReviewDf: DataFrame, use
rDf: DataFrame): DataFrame = {
    val scotUserWithId = scotReviewDf.select("user_id").distinct.rdd.zip
WithIndex.map(x => (x._2.toInt, x._1.getString(0)))
    val scotUserWithIdDf =
sqlContext.createDataFrame(scotUserWithId.map(x => Row.fromTuple(x)), St
ructType(
      Seq(StructField("numeric_user_id", IntegerType, false), StructFiel
d("user_id", StringType, false))))
      .join(userDf, Seq("user_id"))
    //choose fields of interest
      .select("numeric_user_id","user_id", "name")

    scotUserWithIdDf.write.format(csv).options(csvOptions).save(usersFil
eName)
    scotUserWithIdDf
}
```

In [ ]:
```
val scotUserWithIdDf = getUserWithIdDf(sqlContext, scotReviewDf, userDf)
//val scotUserWithIdDf = sqlContext.read.format(csv).options(csvOption
s).load(usersFileName)
```

In [ ]:

```
def getBusinessWithIdDf(sqlContext: SQLContext, scotReviewDf: DataFrame,
 scotBusinessDf: DataFrame): DataFrame = {
    val scotBusinessWithId =
scotReviewDf.select("business_id").distinct.rdd.zipWithIndex.map(x =>
      (x._2.toInt, x._1.getString(0)))
    val scotBusinessWithIdDf = sqlContext.createDataFrame(scotBusinessWi
thId.map(x => Row.fromTuple(x)), StructType(
      Seq(StructField("numeric_business_id", IntegerType, false), Struct
Field("business_id", StringType, false))))
      .join(scotBusinessDf.where("open = true"), Seq("business_id"))
      //choose fields of interest
      .select("numeric_business_id", "business_id", "hours", "categorie
s", "name", "latitude", "longitude", "stars", "full_address")

    val removeLineBreakers = udf((s: String) => {
      s.replaceAll("\n", ", ")
    })

    val turnToString = udf((seq: Seq[String]) => {
      seq.mkString(listSeparator)
    })

    val cleaned = scotBusinessWithIdDf.withColumn("address", removeLineB
reakers(scotBusinessWithIdDf("full_address")))
      .drop("full_address").withColumnRenamed("address", "full_address")
      .withColumn("clean_categories",
turnToString(scotBusinessWithIdDf("categories")))
      .drop("categories").withColumnRenamed("clean_categories", "categor
ies")

    cleaned.write.format(csv).options(csvOptions).save(businessesFileNam
e)
    //cleaned
    sqlContext.read.format(csv).options(csvOptions).load(businessesFileN
ame)
}
```

In [ ]:
```
val scotBusinessWithIdDf = getBusinessWithIdDf(sqlContext, scotReviewDf,
 scotBusinessDf)
//val scotBusinessWithIdDf = sqlContext.read.format(csv).options(csvOpti
ons).load(businessesFileName)
```

In [ ]:
```
def bestBusinessesPerCategory(sqlContext: SQLContext, businessDf: DataFr
ame) = {

 val sorted = businessDf.select("numeric_business_id", "stars", "categor
ies").rdd.flatMap(r =>
   r.getString(2).split(listSeparator).map(c => (c, (r.getInt(0), r.getD
ouble(1)))))
   .groupByKey()
   .mapValues(v => v.toList.sortWith((x,y) => x._2 > y._2).map(x =>
x._1).mkString(","))
   .zipWithIndex.map(x => x._2 + "|" + x._1._1 + "|" + x._1._2)

 sorted.coalesce(1).saveAsTextFile(perCategory)
}
```

In [ ]:
```
bestBusinessesPerCategory(sqlContext, scotBusinessWithIdDf)
```

In [ ]:
```
sc.textFile(perCategory).take(10).foreach(println)
```

Split the data and train and test the models with various parameters

In [ ]:
```
case class CFParams(rank: Int, lambda: Double, numIter: Int)
```

In [ ]:
```
def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: L
ong) = {
  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)
}
```

In [ ]:

```
//The best model was trained with rank = 12 and lambda = 0.1, and numIte
r = 20, and its RMSE on the test set is 1.0992.
def train(ratings: RDD[Rating]): CFParams = {

  val splits = ratings.randomSplit(Array(0.6,0.2,0.2), 42L)
  val training = splits(0).cache
  val validation = splits(1).cache
  val test = splits(2).cache
  val numValidation = validation.count
  val numTest = test.count

  val ranks = List(8, 10, 12)
  val lambdas = List(0.1, 1.0, 10.0)
  val numIters = List(5, 10, 20)
  var bestModel: Option[MatrixFactorizationModel] = None
  var bestValidationRmse = Double.MaxValue
  var bestRank = 0
  var bestLambda = -1.0
  var bestNumIter = -1
  val blocks = -1
  val seed = 42L

  for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
    val model = ALS.train(training, rank, numIter, lambda, blocks, seed)
    val validationRmse = computeRmse(model, validation, numValidation)
    println("RMSE (validation) = " + validationRmse + " for the model tr
ained 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 l
ambda = " + bestLambda
    + ", and numIter = " + bestNumIter + ", and its RMSE on the test set
 is " + testRmse + ".")

  CFParams(bestRank, bestLambda, bestNumIter)
}
```

Build the model for best parameters and export the data so it can be used outside of Spark

In [ ]:
```
def buildBestModel(ratings: RDD[Rating], params: CFParams): MatrixFactor
izationModel = {
  val blocks = -1
  val seed = 42L

  ALS.train(ratings, params.rank, params.numIter, params.lambda, blocks,
 seed)
}
```

In [ ]:
```
def export(sc: SparkContext, modelFileName: String): Unit = {
  val sqlContext = new SQLContext(sc)
  import sqlContext.implicits._

  val modelCf = MatrixFactorizationModel.load(sc, modelFileName)

  val users = modelCf.userFeatures.map(x => x._1)
  val businesses = modelCf.productFeatures.map(x => x._1)

//only keep the best recommendations - with more than 4.5 stars
  val allOptionsDf =
modelCf.predict(users.cartesian(businesses)).filter(r => r.rating >
4.5).toDF.cache

  val perBusinessRDD = allOptionsDf.rdd.map(r => (r.getInt(1),
r.getInt(0))).groupByKey()
      .map(x => x._1 + "|" + x._2.mkString(",")).coalesce(1).saveAsTextF
ile(perBusiness)

  val perUserRDD = allOptionsDf.rdd.map(r => (r.getInt(0),
r.getInt(1))).groupByKey()
      .map(x => x._1 + "|" + x._2.mkString(",")).coalesce(1).saveAsTextF
ile(perUser)
}
```

In [ ]:
```
val ratings = scotReviewDf.join(scotUserWithIdDf.select("numeric_user_i
d", "user_id"), Seq("user_id"))
  .join(scotBusinessWithIdDf.select("numeric_business_id",
"business_id"), Seq("business_id"))
  .select("numeric_user_id", "numeric_business_id", "stars").rdd.map(r =
>
  Rating(r.getInt(0), r.getInt(1), r.getLong(2).toDouble)).cache()

//choose the best model
//val bestParams = train(ratings)
val bestParams = CFParams(12, 0.1, 20)

//train on all data with the best model
buildBestModel(ratings, bestParams).save(sc, modelFileName)

//export to be used outside of Spark
export(sc, modelFileName)
```

In [ ]: