Zeppelin

Collaborative Filtering Movie Recommender

FINISHED

Collaborative Filtering Movie Recommender

Grid Search on Matrix Factorization Model

Took 0 seconds (outdated)

```
%dep
z.reset() // clean up previously added artifact and repository

// add maven repository
z.addRepo("nexus-releases").url("http://oss.sonatype.org/service/local/staging/deploy/mav
z.addRepo("sonatype-nexus-snapshots").url("https://oss.sonatype.org/content/repositories/

z.load("org.nd4j:nd4j-x86:0.4-rc3.8")
z.load("org.nd4j:nd4s_2.11:0.4-rc3.8")
z.load("org.deeplearning4j:deeplearning4j-core:0.4-rc3.8")
z.load("org.deeplearning4j:deeplearning4j-ui:0.4-rc3.8")
```

```
res0: org.apache.zeppelin.dep.Dependency = org.apache.zeppelin.dep.Dependency@4dc3ab2c
```

Import Spark and MLlib

```
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}
```

```
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}
```

Import DL4J

```
// import DL4j
import org.nd4j.linalg.factory.Nd4j;
import org.nd4s.Implicits._;
import org.nd4j.linalg.api.ndarray.INDArray;
import org.nd4j.linalg.dataset.DataSet;
import org.nd4j.linalg.dataset.api.DataSetPreProcessor;
import org.nd4j.linalg.factory.Nd4j;
import org.nd4j.linalg.lossfunctions.LossFunctions;
```

```
import org.nd4j.linalg.factory.Nd4j
import org.nd4s.Implicits._
import org.nd4j.linalg.api.ndarray.INDArray
import org.nd4j.linalg.dataset.DataSet
import org.nd4j.linalg.dataset.api.DataSetPreProcessor
import org.nd4j.linalg.factory.Nd4j
import org.nd4j.linalg.lossfunctions.LossFunctions
```

Load Movie Ratings Sample Dataset

```


//val sampleFile  = "/Users/marvinbertin/Desktop/sample/part-0.csv"
val sampleFile =  "/Volumes/EXTRADRIVE/data/netflix_sample.csv"

val data = sc.textFile(new File(sampleFile).toString)
    .mapPartitionsWithIndex { (idx, iter) => if (idx == 0) iter.drop(1) else iter }
    .map { line =>
  val fields = line.split(",")
  // format: (userId, MID, Rating)
  Rating(fields(1).toInt, fields(3).toInt,fields(2).toDouble)
}
```

```
sampleFile: String = /Volumes/EXTRADRIVE/data/netflix_sample.csv
data: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating] = MapPartiti
onsRDD[181] at map at <console>:117
```

```
// Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
data.take(1)
```

```
res255: Array[org.apache.spark.mllib.recommendation.Rating] = Array(Rating(369848,527
```

9,2.0))

Number of Ratings, Users and Movies

```
val numRatings = data.count
val numUsers = data.map(s => s.user).distinct.count
val numMovies = data.map(s => s.product).distinct.count

println("Got " + numRatings + " ratings from "
  + numUsers + " users on " + numMovies + " movies.")
```

```
numRatings: Long = 3749377
numUsers: Long = 17770
numMovies: Long = 17695
Got 3749377 ratings from 17770 users on 17695 movies.
```

Construct Train, Validation and Test Set

```
// 80% train, 10% validation, 10% test
val trainIdx = (numRatings * 0.8).toInt
val validationIdx =(trainIdx + (numRatings - trainIdx) / 2).toInt
numRatings == validationIdx + (numRatings - trainIdx) / 2
```

```
trainIdx: Int = 2999501
validationIdx: Int = 3374439
res264: Boolean = true
```

```
// construct train,validation and test set
val training = data.zipWithIndex.filter(_._2 <= trainIdx).map(_._1)
val validation = data.zipWithIndex.filter(x => x._2 > trainIdx && x._2 <= validationIdx).
val test = data.zipWithIndex.filter(_._2 > validationIdx).map(_._1)
//val test = training.union(validation).sample(false, 0.1, 1)

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

println("Training: " + numTraining + ", validation: " + numValidation + ", test: " + numT
```

```
training: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating] = MapPar
titionsRDD[3110] at map at <console>:120
validation: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating] = MapP
artitionsRDD[3113] at map at <console>:122
test: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating] = Partitionw
iseSampledRDD[3115] at sample at <console>:127
numTraining: Long = 2999502
```

```
numValidation: Long = 374938
numTest: Long = 338133
Total: Long = 3712573
Training: 2999502, validation: 374938, test: 338133, TOTAL: 3712573
```

Root Mean Squared Error Function

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

```
RMSE: (model: org.apache.spark.mllib.recommendation.MatrixFactorizationModel, data: org.a
pache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating], n: Long)Double
```

Train Matrix Factorization Model and Perform Grid Search

```

val ranks = List(8, 12) // matrix factors rank
val lambdas = List(0.1, 10.0) // regularization
val numIters = List(10, 20) // number of interations
var bestModel: Option[MatrixFactorizationModel] = None
var bestValidationRmse = Double.MaxValue
var bestRank = 0
var bestLambda = -1.0
var bestNumIter = -1

// Grid Search
for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
  val model = ALS.train(training, rank, numIter, lambda)
  val validationRmse = RMSE(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
  }
}
```

```
ranks: List[Int] = List(8, 12)
lambdas: List[Double] = List(0.1, 10.0)
```

```
numIters: List[Int] = List(10, 20)
bestModel: Option[org.apache.spark.mllib.recommendation.MatrixFactorizationModel] = None
bestValidationRmse: Double = 1.7976931348623157E308
bestRank: Int = 0
bestLambda: Double = -1.0
bestNumIter: Int = -1
RMSE (validation) = 0.020681789470564583 for the model trained with rank = 8, lambda =
0.1, and numIter = 10.
RMSE (validation) = 0.020780882828699346 for the model trained with rank = 8, lambda =
0.1, and numIter = 20.
RMSE (validation) = 0.08904667865459813 for the model trained with rank = 8, lambda = 1
0.0, and numIter = 10.
RMSE (validation) = 0.08904667865459813 for the model trained with rank = 8, lambda = 1
0.0, and numIter = 20.
RMSE (validation) = 0.020597185852804196 for the model trained with rank = 12, lambda =
0.1, and numIter = 10.
RMSE (validation) = 0.02066793771950654 for the model trained with rank = 12, lambda =
0.1, and numIter = 20.
RMSE (validation) = 0.08904667865459813 for the model trained with rank = 12, lambda = 1
0.0, and numIter = 10.
RMSE (validation) = 0.08904667865459813 for the model trained with rank = 12, lambda = 1
0.0, and numIter = 20.
```

Evaluate Best Model on Test Set

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

```
testRmse: Double = 0.7850467078294023
The best model was trained with rank = 12 and lambda = 0.1, and numIter = 10, and its RMS
E on the test set is 0.7850467078294023.
```

Compare Best Model with Naive Baseline

```
val meanRating = training.union(validation).map(_.rating).mean
val baselineRmse =
  math.sqrt(test.map(x => (meanRating - x.rating) * (meanRating - x.rating)).mean)

val improvement = (baselineRmse - testRmse) / baselineRmse * 100
println("The best model improves the baseline by " + "%1.2f".format(improvement) + "%.")
```

```
meanRating: Double = 3.6047323407736322
baselineRmse: Double = 1.0864294249890856
improvement: Double = 27.74066223056422
```

The best model improves the baseline by 27.74%.