Exercise1

July 14, 2018

1 Exercise 1: Recommender System from scratch

Krithika Murugesan - 277537

To build a recommendation engine using matrix factorization, we read the data in an RDD and split the data into test and train using random split function.

```
In [1]: from pyspark import SparkContext,SparkConf
        from pyspark import SQLContext
        import pandas as pd

        sc = SparkContext()
        sqlContext = SQLContext(sc)

        #Read data
        rating = pd.read_csv(r'/home/kritz/Documents/DDL/Ex10/movieLens/ratings.csv')
        ratings = sqlContext.createDataFrame(rating)

        #train test split
        Train,test = ratings.rdd.randomSplit([0.8, 0.2])
```

To intialize matrix, we get the number of users and movies by getting the maximum of available values

```
In [2]: #Size u and size v
        sizeU = Train.map(lambda x: int(x[0])).max() + 1
        sizeV = Train.map(lambda x: int(x[1])).max() + 1

        print(sizeU,sizeV)
```

```
672 163950
```

Next we have to build a matrix with all the userId, movieId and actual ratings, we get the individual values and convert them to a matrix, we use a sparse matrix as the ratings are sparse in nature, a user does not rate all the available movies,a csr matrix is more efficient way

```
In [3]: from scipy.sparse import csr_matrix
```

```
userId = Train.map(lambda x: int(x[0])).collect()
movieId = Train.map(lambda x: int(x[1])).collect()
rate = Train.map(lambda x: int(x[2])).collect()

#Getting matrix UV
UV = csr_matrix((rate,(userId,movieId)), shape=(sizeU,sizeV))
print(UV[1])
```

```
  (0, 31)       2
  (0, 1029)     3
  (0, 1061)     3
  (0, 1129)     2
  (0, 1172)     4
  (0, 1263)     2
  (0, 1287)     2
  (0, 1293)     2
  (0, 1343)     2
  (0, 1371)     2
  (0, 1405)     1
  (0, 1953)     4
  (0, 2105)     4
  (0, 2150)     3
  (0, 2193)     2
  (0, 2455)     2
```

Repeating the same process for test data to get a test UV matrix

```
In [4]: userId = test.map(lambda x: int(x[0])).collect()
        movieId = test.map(lambda x: int(x[1])).collect()
        rate = test.map(lambda x: int(x[2])).collect()

        #Getting matrix UV
        uvTest = csr_matrix((rate,(userId,movieId)), shape=(sizeU,sizeV))
        print(uvTest[1])
```

```
  (0, 1339)     3
  (0, 2294)     2
  (0, 2968)     1
  (0, 3671)     3
```

Next, we have to initialize the U and V matrices to random values to learn the matrix factorization with K latent factors

```
In [ ]: import numpy as np

        #Latent factors
        K = 20
```

```
#Partititons
P = 10

#Initialize matrix
uLatent = np.random.random_sample((sizeU,K))
vLatent = np.random.random_sample((K,sizeV))

#Partition train data
trainParts = Train.repartition(P).persist()
trainParts.collect
```

We use RMSE Loss the function returns the RMSE

```
In [8]: def trainLoss(UV,mat):
            mse = mean_squared_error(UV.todense(),mat)
            return np.sqrt(mse)

        def testLoss(test,mat):
            rmse =mean_squared_error(test.collect(),mat)
            return np.sqrt(mse)
```

The SGD function loops the iterations, it builds a smaller UV matrix for each partition in the data. This is used to update the uLatent nd vLatent matrices seperatly. These matrices are updated using the gradient of the loss function and returned to the main function where its average is taken

```
In [9]: def SGD(partition,uLatent,vLatent,iters,alpha):

            userId,movieId,rating = [],[],[]

            for element in partition:
                userId.append(element[0])
                movieId.append(element[1])
                rating.append(element[2])

            users = np.amax(userId) +1
            movies = np.amax(movieId) +1

            #create UV matrix with userIds and moviesIds present in partition
            partUV = csr_matrix((rating,(userId,movieId)), shape=(users,movies))
                                                                       .todense()

            #non-zero elements
            rows,cols = partUV.nonzero()
            i = 0

            #SGD
            while i < iters:
                #Computing gradient for a random sample
                randomNum = random.randint(0,(rows).size-1)
```

```
        r = rows[randomNum]
        c = cols[randomNum]
        sample = partUV[r,c]

        #compute gradients for U and V
        temp1 = -2*(sample - uLatent[r,:].dot(vLatent[:,c]))*(vLatent[:,c].T)
        uGrad = temp1 + (2*alpha*uLatent[r,:])/(partUV[r,:].nonzero()[0].size)
        temp2 = -2*(sample - uLatent[r,:].dot(vLatent[:,c]))*(uLatent[r,:].T)
        vGrad = temp2 + (2*alpha*vLatent[:,c])/(partUV[:,c].nonzero()[0].size)

        #update the U and V values
        uLatent[r,:] = uLatent[r,:] - alpha*uGrad
        vLatent[:,c] = vLatent[:,c] - alpha*vGrad
        i += 1

    return uLatent,vLatent
```

The training function is where the data is sent to the SGD function, once all the part uLatent and vLatent matrices are calculated, their average is taken to get the final updated matrix. Persist is used to avoid memory errors

```
In [10]: def training(trainParts,alpha,uLatent,vLatent,UV,test):
             #Training
             #Learning rate
             iters,epoch = 20,5
             i = 0
             trainRmse,testRmse = [],[]

             while i < epoch:
                 print(i)
                 updatedUV = trainParts.mapPartitions(lambda ratings_part:
                                 SGD(ratings_part,uLatent,vLatent,iters,alpha)).persist()
                 newUV = updatedUV.zipWithIndex().map(lambda x: (x[1]%2, x[0])
                         .reduceByKey(lambda r, k: r+k).map(lambda q: (q[1]/P)).persist()

                 #update uLatent
                 uLatent = newUV.collect()[0]

                 #update vLatent
                 vLatent = newUV.collect()[1]

                 #prediction
                 prediction = np.matmul(uLatent,vLatent)

                 #TrainLoss
                 TrainLoss= trainLoss(UV,prediction)

                 #Validation loss
```

```
        TestLoss = testLoss(test,prediction)
        trainRmse.append(TrainLoss)
        testRmse.append(TestLoss)
        i = i+1
    return(np.mean(testRmseLoss))
```

The cross-validation function is where the training data is split into 3 folds and all their combinations are trained, only two values of alpha are considered as the computation time is higher and efficiency. The alpha with lowest validation loss is selected.

```
In [15]: def crossValidation(alpha,trainOrig):
             totLoss = []
             #Splitting data into folds
             parts = trainOrig.randomSplit([0.33, 0.33,0.34])
             #Possible train and test combinations
             trainSeq = [[0,1],[0,2],[1,2]]
             testSeq = [2,1,0]
             for i in range(0,3):
                 train = sc.union([parts[trainSeq[i][0]],parts[trainSeq[i][1]]])
                 Test = parts[testSeq[i]]

                 #Size u and size v
                 sizeU = train.map(lambda x: int(x[0])).max() + 1
                 sizeV = train.map(lambda x: int(x[1])).max() + 1

                 from scipy.sparse import csr_matrix

                 userId = train.map(lambda x: int(x[0])).collect()
                 movieId = train.map(lambda x: int(x[1])).collect()
                 rate = train.map(lambda x: int(x[2])).collect()

                 #Getting matrix UV
                 UV = csr_matrix((rate,(userId,movieId)), shape=(sizeU,sizeV))

                 userId1 = Test.map(lambda x: int(x[0])).collect()
                 movieId1 = Test.map(lambda x: int(x[1])).collect()
                 rate1 = Test.map(lambda x: int(x[2])).collect()

                 #Getting matrix UV
                 uvTest = csr_matrix((rate1,(userId1,movieId1)), shape=(sizeU,sizeV))
                 #print(uvTest[1])

                 #Latent factors
                 K = 20
                 #Partititons
                 P = 10

                 #Initialize matrix
```

```
            uLatent = np.random.random_sample((sizeU,K))
            vLatent = np.random.random_sample((K,sizeV))

            #Partition train data
            trainParts = train.repartition(P).persist()
            trainParts.collect

            loss = training(trainParts,alpha,uLatent,vLatent,UV,Test)
            #print("loss",loss)
            totLoss.append(loss)
        return (np.mean(totLoss))
```

```
In [20]: from sklearn.metrics import mean_squared_error
         from scipy.sparse import csr_matrix
         import numpy as np
         import random

         alphaGrid = [0.1,0.2]
         for each,loss in zip(alphaGrid,bongu):
             check = []
             P = 10
             print("Alpha = ", each)
             loss = crossValidation(each,Train)
             print("loss = ",loss)
             check.append(loss)
```

```
Alpha =  0.1
loss =  5.063878816619068
Alpha =  0.2
loss =  5.03456525902943
```

It can be seen that for alpha = 0.2 the loss is less, so this parameter is selected, executing the previous statements the train and test loss are

Train RMSE = 4.4563827615429825 Test RMSE = 3.7691529816539208

Exercise2

July 14, 2018

1 Exercise 2: Recommender System using Apache Spark MLLIB

We have to implement Recommender system using Apache Spark, we first read the data into an RDD, we delete the timestamp column as it is not factored in making suggestions. The ratings column is converted to a double type as the built in function supports only this datatype

In [2]:
```
#Setting up spark
from pyspark import SparkContext
from pyspark.sql import SQLContext
import pandas as pd

sc = SparkContext()
sqlContext = SQLContext(sc)
```

In [4]:
```
from pyspark.sql.functions import col
from pyspark.sql.types import DoubleType

#Reading data
rating = pd.read_csv(r'/home/kritz/Documents/DDL/Ex10/movieLens/ratings.csv')
ratings = sqlContext.createDataFrame(rating)

#Dropping timestamp as it is not necessary
columns_to_drop = ['timestamp']
rate = ratings.drop(*columns_to_drop)
rate.show(5)

#Renaming columns and converting to double type for function to use
rate = rate.select(col("userId").alias("user"),col("movieId")
                                        .alias("item"),col("rating").alias("rating"))
newrate = rate.withColumn("rating", rate["rating"].cast(DoubleType()))
```

```
+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     1|     31|   2.5|
|     1|   1029|   3.0|
|     1|   1061|   3.0|
|     1|   1129|   2.0|
```

```
|     1|   1172|   4.0|
+------+-------+------+
only showing top 5 rows
```

After the data is prepared it is split into train data and test data, with 80% as train and remaining 20% as test

```
In [9]: #train and test split
        train,test = newrate.randomSplit([0.8, 0.2])
```

To make recommendations using matrix factorization method, we use the ALS function, which is Alternating Least Square matrix factorization.It trains a matrix factorization model given an RDD of ratings by users for a subset of products. The ratings matrix is approximated as the product of two lower-rank matrices of a given rank (number of features). To solve for these features, ALS is run iteratively with a configurable level of parallelism

The hyper-parameters used in cross-validation are rank, maximum iterations, regularization parameter and alpha. The RMSE evalutor is used, i.e the cv minimizes the RMSE loss function, the best combination of parameters got from this cross-validation are used to make the predictions using which the RMSE is computed for Train and Test datasets

```
In [12]: from pyspark.ml.recommendation import ALS
         from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
         from pyspark.ml.evaluation import RegressionEvaluator

         #ALS model
         alsImplicit = ALS(implicitPrefs=True)

         #Param grid for cv
         paramMapImplicit = ParamGridBuilder() \
                             .addGrid(alsImplicit.rank, [20.0,100.0])\
                             .addGrid(alsImplicit.maxIter, [10.0, 15.0]) \
                             .addGrid(alsImplicit.regParam, [0.01, 1.0]) \
                             .addGrid(alsImplicit.alpha, [10.0, 40.0]) \
                             .build()

         #RMSE Evaluator
         evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating
                                         ",predictionCol="prediction")

         #CV
         cvEstimator= CrossValidator(estimator=alsImplicit,
                                     estimatorParamMaps=paramMapImplicit, evaluator=evaluator)

         #Fitting CV
         cvModel=cvEstimator.fit(train)

         print(cvModel)
```

```
CrossValidatorModel_4c408ac6fee24df5c0ed
```

```
In [30]: # Evaluate the model on training data
         predictions = cvModel.transform(train)
         rmse = evaluator.evaluate(predictions)
         print("Root Mean Squared Error in Train data = " + str(rmse))

         # Evaluate the model on training data
         prediction = cvModel.transform(test)
         rmse = evaluator.evaluate(prediction)
         print("Root Mean Squared Error in Test data = " + str(rmse))
```

```
Root Mean Squared Error in Train data = 2.8648101624433124
Root Mean Squared Error in Test data = 1.6589547812569874
```

It can be seen that the RMSE for test data is 1.66 which is a little more compared to the 0.98 baseline in "http://www.mymedialite.net". Since we are getting our own hyper paramters with the random split of data we make some deviations are bound to happen. Comparing with the values from the previous implementation these are much lesser RMSE values, as we have a wider combination of hyper-parameters being tested.The mediaLite is still the better model...and the predictions are as follows

```
In [32]: #Prediction
         predsImplicit = cvModel.bestModel.transform(test)
         predsImplicit.show(5)
```

```
+----+----+------+----------+
|user|item|rating|prediction|
+----+----+------+----------+
| 452| 463|   2.0|0.69587874|
|  85| 471|   3.0|0.85212296|
| 588| 471|   3.0|0.73714733|
| 548| 471|   4.0|  1.001869|
| 452| 471|   3.0| 0.3340096|
+----+----+------+----------+
only showing top 5 rows
```