Week 11 & 12 - Recommendation Systems

November 14, 2015

1 Week 11 / 12 - Recommendation Systems

Book recommendation using this data set. Inspiration taken from this blog post

1.1 Setup

```
In [34]: # Create PySpark context
         from pyspark import  SparkContext
         sc = SparkContext('local', 'pyspark')
```

1.2 Data Scription and Methodolgy

The Book-Crossing dataset comprises 3 tables.

- BX-Users: Contains the users. Note that user IDs (`User-ID`) have been anonymized and map to integers.
  - For our purposes, we do not nee this file
- BX-Books - Books are identified by their respective ISBN. Invalid ISBNs have already been removed from the dataset. Moreover, some content-based information is given (`Book-Title`, `Book-Author`, `Year-Of-Publication`, `Publisher`), obtained from Amazon Web Services.
  - We load this manually to convert ISBN numbers to an internal index number to be used in the model
- BX-Book-Ratings - Contains the book rating information. Ratings (`Book-Rating`) are either explicit, expressed on a scale from 1-10 (higher values denoting higher appreciation), or implicit, expressed by 0.
  - This is the main file we load, which consists of more than one mullion rows. However, we filter out books that do not appear in the above BX-Books file (why this happens I am not sure).

The following shows code for single run of creating a model for the full data set with filtering out the implicit ratings. I also rand the code on smaller sets of the data. My methodology is to:

- Create a small snippet of the data, and search for the best rank metaparameter based on validation set RMSE
- Use this meta parameter for the medium sized data (500000 rows) and the full data (1000000+ rows) and find the test set RMSE
- Also did the above for filtering out implitic feedback as well as not filtering it out

I then also present a case of testing out the model on a new user whose only ratings are on the first two Harry Potter books.

1.3 Data Download, Extracttion

Here we download the data, unzip it and then read the data into lists

```
In [ ]: # Download and unzip the file
        !wget http://www2.informatik.uni-freiburg.de/~cziegler/BX/BX-CSV-Dump.zip
        !unzip -o BX-CSV-Dump.zip
```

1.4 Load into Spark RDDs

Load in the book data (not sure if we need this)

```
In [98]: # Read in book data into global dict which maps isbn to title
         isbn_to_idx = { }
         isbn_idx = 0
         book_data = open("BX-Books.csv", "r")
         for line in book_data.readlines():
             # Split the line and filter out quotes
             (isbn, title) = line.split(';')[0:2]
             isbn = isbn.replace("\"", "")
             title = title.replace("\"", "")

             # Store in mapping
             isbn_to_idx[isbn] = isbn_idx
             isbn_idx += 1
```

Load ratings data into Spark:

```
In [193]: # Read in chosen data set (min = 5000 lines, half=50000)

          ratings_filename = "BX-Book-Ratings.csv"
          # ratings_filename = "BX-Book-Ratings.half.csv"
          # ratings_filename = 'BX-Book-Ratings.min.csv'

          ratings_raw_data = sc.textFile(ratings_filename)
          ratings_raw_data_header = ratings_raw_data.take(1)[0] # Used to skip the header


          # Given a row fro mthe file, filter out quotes and create dict of isbn -> index
          def convert_row(tokens):
              (user_id, isbn, rating) = (tokens[0].replace("\"", ""), \
                                         tokens[1].replace("\"", ""), \
                                         tokens[2].replace("\"", ""))

              # Filter out implicit ratings
              if rating == 0:
                  return (None, None, None)

              # Convert isbn to index and ...
              global isbn_to_idx
              if isbn in isbn_to_idx:
                  isbn_idx = isbn_to_idx[isbn]
              else:
                  return (None, None, None)

              # ... use it in ratings row for data frame
```

```
        return (user_id, isbn_idx, rating)

    # Read in raw rating data and convert it to proper format
    ratings_data = ratings_raw_data \
                        .filter(lambda line: line != ratings_raw_data_header) \
                        .map(lambda line: line.split(";")) \
                        .map(lambda tokens: convert_row(tokens)) \
                        .filter(lambda tokens: tokens[0] is not None).cache()

    # Print ratings loaded
    print "Loaded %d ratings" % ratings_data.count()
```

```
Loaded 1031175 ratings
```

```
In [166]: ! wc -l "BX-Book-Ratings.csv"
```

```
1149781 BX-Book-Ratings.csv
```

Looks like there is only one book rated that does not exist in database of books. We skip it.
Now lets create the training, validation and test set:

```
In [194]: # Training, validation and test split
          training_RDD, validation_RDD, test_RDD = ratings_data.randomSplit([70, 15, 15], seed=0L)

          # Trim off the rating to get user-book pairs for the validation set
          validation_for_predict_RDD = validation_RDD.map(lambda x: (x[0], x[1]))

          # Trim off the rating to get user-book pairs for the test set
          test_for_predict_RDD = test_RDD.map(lambda x: (x[0], x[1]))
```

1.5 Create a model

Here we will use MLLib's ALS function to create models for prediction of book ratiings. We do a simple search over different rank values to do some meta-parameter tuning:

```
In [199]: from pyspark.mllib.recommendation import ALS
          import math

          # Model parameters
          seed = 5L
          iterations = 10
          # iterations = 20
          regularization_parameter = 0.1

          # ranks = range(4, 50, 2)
          # ranks = [30] # Found by running on min data
          ranks = [36]   #  Found by running on min data, but with filtering on implicit feeback

          errors = [ ]

          min_error = float('inf')
          best_rank = -1
          best_iteration = -1
          best_model = None

          # Look for best ALS model iterating over diff rank values
```

```
for rank in ranks:
    # Create ALS model
    model = ALS.train(training_RDD, rank, seed=seed, iterations=iterations, \
                      lambda_=regularization_parameter)

    # Come up with predictions for the validation set
    predictions = model.predictAll(validation_for_predict_RDD)\
                       .map(lambda r: ((r[0], r[1]), r[2]))

    # TODO
    rates_and_preds = validation_RDD\
                       .map(lambda r: ((int(r[0]), int(r[1])), float(r[2])))\
                       .join(predictions)

    # Record and update the error
    error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())
    errors.append(error)

    # What is the current RMSE
    print 'For rank %s the RMSE is %s' % (rank, error)

    # Error update
    if error < min_error:
        min_error = error
        best_rank = rank
        best_model = model

# Final output
print 'The best model was trained with rank %s' % best_rank
```

```
For rank 36 the RMSE is 4.14137549189
The best model was trained with rank 36
```

1.6 Test Set

Lets evaluate the best model on the held-out test set and see what the final RMSE of the model is:

```
In [196]: predictions = best_model.predictAll(test_for_predict_RDD)\
                                  .map(lambda r: ((r[0], r[1]), r[2]))

          rates_and_preds = test_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2])))\
                                    .join(predictions)

          error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

          print 'For testing data the RMSE is %s' % (error)
```

```
For testing data the RMSE is 4.16286876454
```

1.7 Results

The test set RMSE values for the mini, half and full data sets (with best rank = 30 and no filtering on implicit feedback):

```
In [198]: from prettytable import PrettyTable
          x = PrettyTable()
```

```
x.field_names = ["Data Set (rank = 30, no filter on implicit ratings)", "RMSE"]
x.add_row(["Mini Data (~5000 rows)", 5.17305247323])
x.add_row(["Half Data (~500,000)", 4.36752088077])
x.add_row(["Full Data (~1,000,000 rows)", 4.16286876454])
print x
```

```
+-----------------------------------------------------+---------------+
| Data Set (rank = 30, no filter on implicit ratings) |      RMSE     |
+-----------------------------------------------------+---------------+
|                Mini Data (~5000 rows)               | 5.17305247323 |
|                 Half Data (~500,000)                | 4.36752088077 |
|            Full Data (~1,000,000 rows)              | 4.16286876454 |
+-----------------------------------------------------+---------------+
```

The RMSE values for the mini, half and full data sets (with best rank = 36 and filtering on implicit feedback), which we can see generally improves the RMSE:

```
In [197]: # Lets record our results but after filtering out implicit feedback
          x = PrettyTable()
          x.field_names = ["Data Set (rank = 36, filter on implicit ratings)", "RMSE"]
          x.add_row(["Mini Data (~5000 rows)", 5.08574829261])
          x.add_row(["Half Data (~500,000)", 4.32793604751])
          x.add_row(["Full Data (~1,000,000 rows)", 4.16286876454])
          print x
```

```
+-----------------------------------------------------+---------------+
| Data Set (rank = 36, filter on implicit ratings) |      RMSE     |
+-----------------------------------------------------+---------------+
|              Mini Data (~5000 rows)                 | 5.08574829261 |
|               Half Data (~500,000)                  | 4.32793604751 |
|           Full Data (~1,000,000 rows)               | 4.16286876454 |
+-----------------------------------------------------+---------------+
```

1.8 Lets Test Some Predicitions

First, lets find some books to rate:

```
In [73]: print books_data.filter(lambda x: x[0]=='0590353403').take(1)
         print books_data.filter(lambda x: x[0]=='0439064872').take(1)
         print books_data.filter(lambda x: x[0]=='0439136350').take(1)
```

```
[(u'0590353403', u"Harry Potter and the Sorcerer's Stone (Book 1)")]
[(u'0439064872', u'Harry Potter and the Chamber of Secrets (Book 2)')]
[(u'0439136350', u'Harry Potter and the Prisoner of Azkaban (Book 3)')]
```

First we need to determine a user ID that is not used:

```
In [74]: !head BX-Users.csv
```

User ID 0 looks fine. Now, we will create two ratings on the first two harry potter books:

```
In [201]: print "Indies for potter books", \
                  isbn_to_idx["0590353403"], \
                  isbn_to_idx["0439064872"], \
                  isbn_to_idx["0439136350"]
```

```
Indies for potter books 2810 3460 3840
```

For user 0, create two ratings for the first two books:

```
In [133]: new_user_id = 0
          new_user_ratings = [
               (new_user_id, 2810, 10),
               (new_user_id, 3460, 10),
              ]
          new_user_ratings_RDD = sc.parallelize(new_user_ratings)
          print 'New user ratings: %s' % new_user_ratings_RDD.take(2)
```

```
New user ratings: [(0, 2810, 10), (0, 3460, 10)]
```

Create a new model based on the training data but with our new user recommendations (via RDD union) and with the best_rank calculated above:

```
In [ ]: training_RDD_with_new_ratings = training_RDD.union(new_user_ratings_RDD)


        new_ratings_model = ALS.train(training_RDD_with_new_ratings, best_rank, seed=seed,
                                      iterations=iterations, lambda_=regularization_parameter)
```

Now, what do we predict as the rating for this user for the third potter book:

```
In [136]: new_ratings_model.predict(new_user_id, 3840)
```

```
Out[136]: 9.075894409195863
```

1.9 Details / Issues

These results aren;t so great. While we get better results using more data, the RMSE are pretty high, considering the scale is from 1 - 10.

One issue that that it would be great to merge books: some books in the data set are essentially the same book, but a different edition, or a different printing, or paperback versus hardcover. They represent the same 'thing' in the world, and having some way of merging them would allow for better recommendations.

1.10 Cleanup

```
In [33]: sc.stop()
```