Final Project - James Quacinella

December 20, 2015

1 IS622 Final Project: Book Recommendations

1.0.1 An Extension of Week12's Recommendation Project

1.1 Background

For this final project, I will extend the work done in Week 12 where I built a recommender system for ratings data for books. As a recap on what that project was about, I used data on book ratings from this data set. This data set has 1 millon+ rows of user ratings (from 1 - 10) of books identified by ISBN number.

I used this data to generate a recommendation system using PySpark. The model developed used matrix factorization to allow us to take in a set of ratings for a new user, and predict their ratings for any other book in our data. The conclusion from Week12, however, was that the model generated was not so great with respect to how accurate the matrix factorization reproduced the original data.

One idea to improve its performance was to go through a round of de-duplication: there are books that have different ISBN numbers but are essentially the same book l(two different volumes of same book, for example). De-duplication will be done by some kind of locality sensitive hashing to determine titles that are similar enough to consider merge-worthy. I found a Spark implementation of LSH in python and will use it to efficently find similar books by titles.

My hypothesis is that de-duplicating the data (books in this case) and merging ratings from both books into one would help the accuracy of the model.

The model will be compared to Week12's results to see if the accuracy (based on a train-validation-test set split) has improved. The model will also be compared to see how much better it performs in execution time.

1.2 Data Defintion

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).

Git does not allow large files, so they are not in the git repo. The file 'BX-CSV-Dump.zip' can be unzipped and the data files will be loaded.

1.3 Objective

My hypothesis is that de-duplicating the data (books in this case) and merging ratings from merged books will help the accuracy of the model as measured by the RMSE of the model on a validation data set.

1.4 Methodology

1.4.1 Overview

- First, we will review the original model and how it was built, and associated RMSE values
  - 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
- Next we will perform de-duplication of books, merging the reviews together
- We will then create a second model with the de-duplication in place
- Finally, we will compare performances of these models

NOTE: Going forward, the code that follows was executed on an EC2 server, so I am copying and pasting final code and results. Another notebook has the work in how the merge dictionary is generated, with output from the server

1.4.2 Recapping Week 12 - Load Data into Spark

The code below loads the Book information into a Spark RDD object (this is a listing of tuples of (ISBN, title)):

```
In [ ]: # Configure the necessary Spark environment
        import os
        import sys
        import pickle
        import math

        import itable
        import pandas as pd

        # We are using ALS to factor our user-to-book rating matrix
        from pyspark.mllib.recommendation import ALS

        # Create PySpark context
        from pyspark import  SparkContext, SQLContext, Row
        sc = SparkContext('local[*]', '--executor-memory=8g pyspark')
        # sc.defaultParallelism ## 16

        # Read in file into RDD
        book_filename = "/spark/BX-Books.csv"
        book_raw_data = sc.textFile(book_filename)

        # Used to skip the header
        book_raw_data_header = [ word.replace("\"", "")
                                for word in book_raw_data.take(1)[0].split(';')[1:3] ]

        # Filter out quotes after splitting line by ; delim and keeping only what we need
        book_raw_data =  book_raw_data.map(lambda line: [ word.replace("\"", "")
                                                     for word in line.split(";")[1:3]]) \
                                .filter(lambda line: line != book_raw_data_header) \
                                .map(lambda line: "%s %s" % (line[0], line[1]))

        # Confirm data loaded
        # book_raw_data.take(5)

        # 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
```

Now we will load the Ratings information into a Spark RDD object. Ratings file is just a list of tuples of the form (user_id, isbn, rating of isbn). This is a useful format for ALS:

```
In [ ]: # 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 from the 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 number of ratings loaded
        print "Loaded %d ratings" % ratings_data.count()
```

Now that the ratings data is loaded, lets create a training, validation and testing split:

```
In [ ]: # 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]))
```

Now lets train an ALS models of various rank and pick the best one based on minimizing validation RMSE error:

```
In [ ]: def train_model(training_RDD, validation_for_predict_RDD, validation_RDD):
            # Model parameters
            seed = 5L
            iterations = 10
            # iterations = 20
            regularization_parameter = 0.1

            ranks = range(20, 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

            return best_model

        # Train model
        best_model_pre_merge = train_model(training_RDD, validation_for_predict_RDD, validation_RDD)
```

1.4.3 Finding Titles to Merge

To find the titles to merge together, we need to compare each title to all other titles in the corpus to find which ones are similar (using an appropriate distance metric, like jaccard distance). However, with 200000+ books, this is not feasible. Locality sensitive hashing can help by binning titiles together that are more likely to be similar, to reduce the number of comparisons one has to do. As alluded to above, there is a Spark implementation of LSH in python which I will use to find book titles that are similar enough to merge.

However, even with LSH, I found it necessary to use an Amazon EC2 server to get things done. Using spot instances, I setup a 16 core / 30Gb machine (or whatever was available that had som decent resources) and installed / setup ipython to use Spark. I made sure all data was stored on an EBS volume to make sure that no data was lost if the spot instance died.

After that, I experimented with LSH parameters on a small subset of the data. Even with these resources, I could not process the whole set, but in two sets of 50,000 books at a time. Note: doing it this way does not mean LSH was applied to 100,000 titles, as it was applied to two subsets (i.e. LSH is not applied across the groups, so no matching is done there).

The code in the other ipython notebook is responsible for loading the book titles into a Spark RDD object, creating an appropriate TFIDF representation of all titles, and passing the TFIDF results to the PythonLSH module. This part of the process can take 40m to a few hours depending on the size of the dataset. I found the LSH implementation took up a lot of memory. I tried tweaking the params as best as I could, but I was still limited. See the “Next Steps” section below for more details.

1.4.4 Merge Titles

Using the results from LSH, I tend compare titles that are in each bucket, and construct a merge dictionary that maps an ISBN ‘index’ to an index of a title to merge it with. I serialize this dictionary to save my work as I go, and then load this in another notebook to perform the actual merging when loading ratings. As the rating data is loaded in, the ISBN is converted to an index and the dictionary is consulted to see if this so should be merged (i.e., another index is used). Therefore, all ratings for many titles for the same book will refer to one index.

Here we will use the merge dictionary generated from the first 50,000 books:

```
In [ ]: f = open("/spark/merge_dict_v1_p100m100n50b10c5_xlarge", "rb")
        merge_dict = pickle.load(f)
        f.close()

        # Use spark accumulators to see how many times we merge
        count = sc.accumulator(0)
        missed_count = sc.accumulator(0)

        def merge_ratings(tokens):
            global count
            global missed_count
            # (user_id, isbn_idx, rating)
            orig_isbn_idx = tokens[1]

            if orig_isbn_idx in merge_dict:
                count += 1
                return (tokens[0], merge_dict[orig_isbn_idx], tokens[2])
            else:
                missed_count += 1
                return tokens

        # Read in raw rating data and convert it to proper format
        new_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) \
                           .map(merge_ratings).cache()

        # print "Loaded %d ratings" % new_ratings_data.count()
```

Using the count values, only about 8% of reviews were merged. Not a huge matching but again, only evaluated a portion of the books

1.4.5 Load Data From New Post-Merged Ratings Data

```
In [ ]: # Training, validation and test split
        training_RDD, validation_RDD, test_RDD = new_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]))
```

1.4.6 Train New Model

```
In [ ]: best_model_post_merge = train_model(training_RDD, validation_for_predict_RDD, validation_RDD)
```

1.5 Results & Discussions

Lets compare the pre and post-merge RMSE errors for models of different rank sizes using only the first 50,000 titles and the first 100,000 titles:

```
In [23]: results = [(20, 4.26313112355, 4.25828212238),
                    (22, 4.23259538535, 4.23275657208),
                    (24, 4.21148751659, 4.20943547919),
                    (26, 4.18826859766, 4.18930658745),
                    (28, 4.16448657922, 4.16958226847),
                    (30, 4.15846533055, 4.15713268602),
                    (32, 4.1436364273, 4.1416517817),
                    (34, 4.12220323035, 4.12264843619),
                    (36, 4.11941536673, 4.11715783255),
                    (38, 4.10410432488, 4.10227015671),
                    (40, 4.09841351419, 4.09726798097),
                    (42, 4.09207336934, 4.08903422579),
                    (44, 4.08177248426, 4.07665148098),
                    (46, 4.07279330036, 4.06946764295),
                    (48, 4.07400818709, 4.07036979413)]

           results = map(lambda x: (x[0], x[1], x[2], abs(x[2] - x[1])) , results)

           df = pd.DataFrame.from_records(results)
           df.columns = ["Rank", "Pre-Merge RMSE", "Post-Merge RMSE (50k)", "RMSE Delta (50k)"]
           pt = itable.PrettyTable(df, tstyle=itable.TableStyle(theme="theme1"), center=True)
           pt.set_cell_style(font_weight="bold", rows=[-2])
           pt
```

```
Out[23]: <itable.itable.PrettyTable at 0x7f15cbb4d750>
```

This is not a huge result at all, but the RMSE is generally lower post-merge, which shows that this technique might have promise. The difference is pretty small, so more work would be needed to confirm that viability of this method. The results above were created from only 50,000 books (out of a total of 271,380).

Lets look at the results after using both merge lists (from first 50,000 and the next 50,000). I redefine the merge_ratings() function above to use both merge_dict and merge_dict2, using an appropriate offset for merge dict2 (since its for lines 50,001 - 100,000, idx0 there represents idx50001).

```
In [ ]: f = open("/spark/merge_dict_v1_p100m100n50b10c5_xlarge2", "rb")
        merge_dict2 = pickle.load(f)
        f.close()

        # Use spark accumulators to see how many times we merge
        count = sc.accumulator(0)
        missed_count = sc.accumulator(0)

        def merge_ratings(tokens):
            global count
            global missed_count
            # (user_id, isbn_idx, rating)
            orig_isbn_idx = tokens[1]
        #     orig_isbn_idx = tokens[1] - 50001

            if orig_isbn_idx - 50001 in merge_dict2:
                count += 1
                return (tokens[0], merge_dict2[orig_isbn_idx - 50001], tokens[2])
            elif orig_isbn_idx in merge_dict1:
                count += 1
                return (tokens[0], merge_dict1[orig_isbn_idx], tokens[2])
            else:
                missed_count += 1
                return tokens
```

```
# Read in raw rating data and convert it to proper format
new_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) \
                    .map(merge_ratings).cache()

# print "Loaded %d ratings" % new_ratings_data.count()
```

Running the same methodology above gives me:

```
In [24]: results = [(20, 4.26313112355, 4.25828212238, 4.26208684307),
                    (22, 4.23259538535, 4.23275657208, 4.2175599298),
                    (24, 4.21148751659, 4.20943547919, 4.20655965168),
                    (26, 4.18826859766, 4.18930658745, 4.18325077481),
                    (28, 4.16448657922, 4.16958226847, 4.16371352081),
                    (30, 4.15846533055, 4.15713268602, 4.14492062267),
                    (32, 4.1436364273, 4.1416517817, 4.13621839668),
                    (34, 4.12220323035, 4.12264843619, 4.12741234399),
                    (36, 4.11941536673, 4.11715783255, 4.10674592613),
                    (38, 4.10410432488, 4.10227015671, 4.10010399642),
                    (40, 4.09841351419, 4.09726798097, 4.09361783599),
                    (42, 4.09207336934, 4.08903422579, 4.08362561668),
                    (44, 4.08177248426, 4.07665148098, 4.07829758906),
                    (46, 4.07279330036, 4.06946764295, 4.05993951906),
                    (48, 4.07400818709, 4.07036979413, 4.07202519358)]

         results = map(lambda x: (x[0], x[1], x[3], abs(x[3] - x[1])) , results)

         df = pd.DataFrame.from_records(results)
         df.columns = ["Rank", "Pre-Merge RMSE", "Post-Merge RMSE (50k)", "RMSE Delta (50k)"]
         pt = itable.PrettyTable(df, tstyle=itable.TableStyle(theme="theme1"), center=True)
         pt.set_cell_style(font_weight="bold", rows=[-2])
         pt

Out[24]: <itable.itable.PrettyTable at 0x7f15cbbb7fd0>
```

Same as above. Some marginal gains in error rate, though the delta acheieved with more data (0.0128537813) is 4 times the one gained with just 50k titles. Promising maybe, but one has to compare how much computational cost LSH is, versus how much of a gain this would represent in the real world performance, and it this would be worth all the hassle

1.5.1 Next Steps

The first step would be to review the LSH implementation to make sure its as efficient as possible. I took a brief look at the LSH function in the lsh.py file in the module, and it generally makes sense to me. However, the code I wrote afterwards to process the results may not be the best way of doing things either (though the majority of the time was spent in the LSH routine).

After confirming the effectiveness of the module, I would expand the example to use a cluster of machines. Scaling one machine was difficult, as I had to end up tweaking per-executer memory limits (I ran a few computations that took a few hours to only die of MemoryLimit exceptions; quite frustrating!). Its very time consuming preventing quick iterations of work. When working on this kind of scale, thinking-ahead is paramount. I did not have enough time to get into this, but would have been cool to see if the LSH technique would expand easily into more Spark machines.

I would also tweak the LSH parameters by either 1) a deeper understanding of the space-cpu tradeoff in the parameter space and maybe 2) a meta-parameter search.

```
In [ ]:
```