IS622 Final - Generating Merge with LSH

December 20, 2015

1 Generating Merge Dictionary

This no tebook shows the work I did in generating a dictionary that maps one books index (in the main file) to another book, where the mapping is between equivalent titles with different ISBN numbers (which can be due to different editions, softcover versus hardcover). This I assert will help the recommendation engine, since recommendations for all editions can be merged together (this should maintain the semantics of the rating being about the book, not about a particular form of the book, to a high degree).

First we setup Spark to run on all CPUs (16 in this case, which is printed out)

```
In [54]: sc.stop()
```

```
In [1]: import time
        from pprint import pprint

        import nltk
        stopwords = nltk.corpus.stopwords.words('english')

        # Create PySpark context
        from pyspark import  SparkContext, SQLContext, Row
        sc = SparkContext('local[*]', '--executor-memory=10g pyspark')
        sc.defaultParallelism

        # stopwords_spark = sc.broadcast(stopwords)
```

```
Out[1]: 16
```

I worked on this iteratively, but ended up using "books-xlarge.csv" and "books-xlarge2.csv", which I created from the first 50,000 and the next 50,000 titles. I read it into an RDD object and confirm the format as a list of tuples of the form (list of tokens, idx):

```
In [2]: book_filename = "BX-Books.csv"
        book_filename = "books-mini.csv"
        book_filename = "books-small.csv"
        book_filename = "books-medium.csv"
        book_filename = "books-large.csv"
        book_filename = "books-xlarge.csv"
        book_filename = "books-xxlarge.csv"
        # book_filename = "books-xlarge-tail.csv"
        book_filename = "books-xlarge2.csv"

        # Read in as RDD
        book_raw_data = sc.textFile(book_filename, minPartitions=6)

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

        book_raw_data =  book_raw_data.map(lambda line: [ word.replace("\"", "") for word in line.split
                                    .filter(lambda line: line != book_raw_data_header) \
                                    .map(lambda line: "%s %s" % (line[0], line[1])) \
                                    .map(lambda line: [word for word in line.split(" ")
                                                       if word.lower() not in stopwords]).zipWithIndex(
        # Show first few titles
        book_raw_data.take(2)
```

```
Out[2]: [([u'Flight',
           u"Stoneman's",
           u'Son',
           u'(The',
           u'Flight',
           u'Stoneman)',
           u'Terence',
           u'Munsey'],
          0),
         ([u'Son',
           u'Smaller',
           u'Hero',
           u'(New',
           u'Canadian',
           u'Library)',
           u'M.',
           u'Richler'],
          1)]
```

I created a SQL context around this to allow for easy lookups of books:

```
In [ ]: sqlContext = SQLContext(sc)

        book_table = book_raw_data.map(lambda p: Row(idx=p[1], title=p[0]))
        schema_books = sqlContext.createDataFrame(book_table)
        schema_books.registerTempTable("books")
```

Here are examples of how to do so:

```
In [3]: # example
        book = sqlContext.sql("SELECT * FROM books WHERE idx = 400 LIMIT 1")
        book.collect()
```

```
Out[3]: [Row(idx=400, title=[u'Mouth', u'Mouth', u'Kevin', u'Elyot'])]
```

```
In [4]: titles = sqlContext.sql("SELECT title FROM books").map(lambda row: row[0])
        titles.take(1)
```

```
Out[4]: [[u'Flight',
          u"Stoneman's",
          u'Son',
          u'(The',
          u'Flight',
          u'Stoneman)',
          u'Terence',
          u'Munsey']]
```

Next, I take all the titles and create a TFIDF representation using MLLib:

```
In [5]: start = time.time()
        from pyspark.mllib.feature import HashingTF
        hashingTF = HashingTF()
        tf = hashingTF.transform(titles)
        from pyspark.mllib.feature import IDF
        tf.cache()
        idf = IDF().fit(tf)
        tfidf = idf.transform(tf)
        end = time.time()
```

```
In [7]: print end - start
```

```
2.0401930809
```

Next, using that, I send the TFIDF representation through the LSH python module, using a certain set of parameters. This also took quite a lot of tweaking to get right, and to understand how the params affect runtime cpu and memory usage:

```
In [8]: start = time.time()

        from pyspark_lsh import lsh
        # import pyspark_lsh
        # reload(pyspark_lsh)

        # p : integer, larger than the largest value in data.
        # m : integer, number of bins for hashing.
        # n : integer, number of rows to split the signatures into.
        # b : integer, number of bands. Each band will have (n / b) element
        # c : integer, minimum allowable cluster size.
        lsh_model = lsh.run(tfidf, p = 100, m = 100, n = 50, b = 10, c = 5)
        end = time.time()
        print end - start
```

```
bp 1
6
bp 2
6
bp 3
bp 4
6
bp 5
6
bp 6
12
bp 7
12
bp 8
24.3293390274
```

Next, we use the LSH model scores RDD, which maps a bucket to a ‘score’, which is an average Jaccard distance of its members. I do some more filtering to help reduce the size of the results. I print out the number of buckets to check and how long it took, and how many partitions the results are spead over:

```
In [9]: start = time.time()
        buckets_to_check = lsh_model.scores.filter(lambda bucket_score: bucket_score[1] > 200).collect(
        # print buckets_to_check
        print len(buckets_to_check)
        end = time.time()
```

```
1869
```

```
In [11]: print end - start
```

```
2221.714535
```

```
In [12]: lsh_model.scores.getNumPartitions()
```

```
Out[12]: 12
```

Here I setup some useful functions to implement a disjoint data structure, which I use to merge books together to a single book (i.e. if b is mapped to a, and c is mapped to be, a useful merge would be to map both b and c to a)

```
In [13]: # https://www.quora.com/How-do-you-implement-a-Disjoint-set-data-structure-in-Python
         global_merge_list = {}

         def parent(rep, v):
             if rep[v] == v:
                 return v
             rep[v] = parent(rep, rep[v])
             return rep[v]

         def merge(rep, L):
             for edge in L:
                 u, v = edge
                 if u not in rep:
                     rep[u] = u
                 if v not in rep:
                     rep[v] = v
                 rep[parent(rep, v)] = parent(rep, u)
             return rep

         def jaccard(a, b):
             c = a.intersection(b)
             return float(len(c)) / (len(a) + len(b) - len(c))
```

For each bucket, compare all titles to one another (only half-cartesian) and if the Jaccard distance is > than 0.85, I consider it a candidate for merging and add it to the global dictionary (and use the functions above to get the correct location to place the merge). I print out how long each bcket evaluation takes, which caused too much output (on average, it took about 0.7s) so I cleared it from the notebook:

```
In [ ]: start = time.time()
        for (bucket_idx, score) in buckets_to_check:
            bstart = time.time()

            bv = lsh_model.buckets_vectors.filter(lambda bv: bv[0] == bucket_idx)
            # print bv.collect()

            values = bv.groupByKey().collect()[0][1]
```

```
#     books_in_bucket = []
#     for x in bv.collect():
#         books_in_bucket.append(sqlContext.sql("select * from books where idx = %s LIMIT 1" % 
#                                 .map(lambda row: (row[0], row[1])).collect()[0])
    idxs = str(values.data).replace('[', '(').replace('L', '').replace(']', ')')
    books_in_bucket = sqlContext.sql("select * from books where idx in %s" % idxs) \
                            .map(lambda row: (row[0], row[1])).collect()

    to_merge = []
    for idx1, book1 in enumerate(books_in_bucket):
        (gidx1, title1) = book1
        for idx2 in range(idx1+1, len(books_in_bucket)):
            book2 = books_in_bucket[idx2]
            (gidx2, title2) = book2

            x = set(title1)
            y = set(title2)

            if jaccard(x,y) > 0.85:
                to_merge.append((gidx1, gidx2))

    #pprint(to_merge)

    merge(global_merge_list, to_merge)

    print(time.time() - bstart)

#pprint(global_merge_list)

end = time.time()
print end - start
```

Display a section of the results and see if they are indeed good matches for merging:

```
In [15]: count = 0
         for merge1, merge2 in global_merge_list.iteritems():
             if merge1 != merge2:
                 book = sqlContext.sql("SELECT * FROM books WHERE idx = %s LIMIT 1" % merge1)
                 print book.collect()
                 book = sqlContext.sql("SELECT * FROM books WHERE idx = %s LIMIT 1" % merge2)
                 print book.collect()
                 print
                 count += 1
                 if count > 10: break
```

```
[Row(idx=8192, title=[u"Fortune's", u'Wheel', u'Cynthia', u'Voigt'])]
[Row(idx=3176, title=[u"Fortune's", u'Wheel', u'Cynthia', u'Voigt'])]

[Row(idx=24578, title=[u'Silicon', u'Snake', u'Oil:', u'Second', u'Thoughts', u'Information', u'Highway
[Row(idx=17567, title=[u'Silicon', u'Snake', u'Oil:', u'Second', u'Thoughts', u'Information', u'Highway

[Row(idx=8199, title=[u'Secret', u'Garden', u'(Penguin', u'Classics)', u'Frances', u'Hodgson', u'Burnet
[Row(idx=1145, title=[u'Secret', u'Garden', u'(Penguin', u'Popular', u'Classics)', u'Frances', u'Hodgso
[Row(idx=47788, title=[u'Stones', u'Summer', u'Dow', u'Mossman'])]
[Row(idx=4610, title=[u'Stones', u'Summer', u'Dow', u'Mossman'])]

[Row(idx=24612, title=[u'Tan', u'Veloz', u'Como', u'El', u'Deseo', u'Laura', u'Esquivel'])]
[Row(idx=18724, title=[u'Tan', u'Veloz', u'Como', u'El', u'Deseo', u'Laura', u'Esquivel'])]

[Row(idx=24619, title=[u'Unholy', u'Fire', u'Whitley', u'Strieber'])]
[Row(idx=62, title=[u'Unholy', u'Fire', u'Whitley', u'Strieber'])]

[Row(idx=16432, title=[u'Zen', u'Art', u'Motorcycle', u'Maintenance:', u'Inquiry', u'Values', u'Robert'
[Row(idx=4483, title=[u'Zen', u'Art', u'Motorcycle', u'Maintenance:', u'Inquiry', u'Values', u'Robert',

[Row(idx=32833, title=[u'Dracula', u'Bram', u'Stoker'])]
[Row(idx=2119, title=[u'Dracula', u'Bram', u'Stoker'])]

[Row(idx=24645, title=[u'Arrowsmith', u'(Signet', u'Classics', u'(Paperback))', u'Sinclair', u'Lewis'])
[Row(idx=24500, title=[u'Arrowsmith', u'(Signet', u'Classics', u'(Paperback))', u'Sinclair', u'Lewis'])

[Row(idx=45069, title=[u'BITCH', u'Judy', u'Collins'])]
[Row(idx=6275, title=[u'BITCH', u'Judy', u'Collins'])]

[Row(idx=32854, title=[u'Return', u'King', u'(The', u'Lord', u'Rings,', u'Part', u'3)', u'J.', u'R.', u
[Row(idx=1971, title=[u'Return', u'King', u'(The', u'Lord', u'Rings,', u'Part', u'3)', u'J.', u'R.', u'
```

Save the merge dictionary to the disk so if the spot instance dies, I have saved work. From here, I use this in the main notebook to perform the actual merging.

In total, I generated merge_dict_v1_p100m100n50b10c5_xlarge and merge_dict_v1_p100m100n50b10c5_xlarge2, which are generated from the first 50,000 and the next 50,000 books.

```
In [16]: import pickle
         f=open('merge_dict_v1_p100m100n50b10c5_xlarge2', 'wb')
         pickle.dump(global_merge_list, f, protocol=2)
         f.close()
```