MovieLens Recommendation

Names: Lúcia Moreira and Nirbhaya Shaji

Date: April, 19th, 2020

Google Cloud projects ID: *propane-primacy-268509*
bubbly-team-268210

BigQuery datasets:	*tiny1*	*propane-primacy-268509*
	tiny2	*propane-primacy-268509*
	tiny3	*propane-primacy-268509*
	tiny4	*propane-primacy-268509*
	*medium1**	*propane-primacy-268509*
	*medium2**	*bubbly-team-268210*
	*medium3**	*bubbly-team-268210*
	*medium4**	*bubbly-team-268210*
	*large1**	*bubbly-team-268210*
	*large2**	*bubbly-team-268210*
	*large3**	*bubbly-team-268210*
	*large4**	*bubbly-team-268210*

* no Jaccard index implementation

Summary

This project considers the use of the TF-IDF metric and variants for implementing a simple recommendation system for movies based on a keyword search. Twelve different datasets were considered based on MovieLens data.

Extra work involved the following two tasks: 1- implementation of a weighted movie search algorithm, and 2 – computation of the similarity metric between two movies based on the Jaccard index. Task 2 was implemented only for the tiny datasets once this task was intensively time-consuming regarding computation time for the bigger datasets.

It was observed that, for instance, the first 7 recommendations based on the same keywords (comedy + magic + children) for the tf_idf search and the weighted search only match in one of the recommendations and in an order slightly different, for the medium1 dataset. Moreover, in the weighed search the first recommendation is not a magical comedy for children while the second one it is. The first recommendation probably occurred due to the high rating plus the great number of ratings for that movie. In future searches, maybe the weights given to each parcel of the present weighed search metric (0.5:0.5) used may need to be reconsidered, but this task was out of the scope of the present project.

Implementation

Calculation of the TF-IDF metric

Figure 1 shows the code considered for this task and the partial printed output.

```
import pyspark.sql.functions as F
df2= all_words.groupBy('movieId').agg(F.collect_list('word').alias('words'))
aggWords = all_words.groupBy('movieId', 'word')\
                    .agg(F.count('word').alias('numWords'))
max_numWords = aggWords.groupBy('movieId').agg(F.max('numWords').alias('max_numWords'))
TF = max_numWords.join(aggWords,'movieId')
TF2 = TF.withColumn("tf", F.col('numWords')/F.col("max_numWords"))
N_DOCS=df2.count()
IDF = TF2.groupBy('word').agg(F.count('movieId').alias('n'))
IDF2 = IDF.withColumn("idf", F.log2(N_DOCS/F.col("n")))
IDF3 = IDF2.join(TF2,'word')
TF_IDF = IDF3.withColumn("tf_idf", F.col('tf')*F.col("idf"))
TF_IDF2= TF_IDF.select("movieId","word","tf_idf")
TF_IDF2_MIN = TF_IDF2.filter(TF_IDF2.tf_idf>=MIN_TF_IDF)
```

```
TF_IDF2_MIN.show()
```

```
+-------+----------+-------------------+
|movieId|      word|             tf_idf|
+-------+----------+-------------------+
|  72117|        07| 1.6715104009236168|
| 162988|     anime|  6.772589503896928|
| 190481|     anime|  3.386294751948464|
| 122982|     anime|  6.772589503896928|
|  76049|       art|  6.772589503896928|
| 118348|       art|  6.772589503896928|
|   3119|      bava|  2.785850668206028|
|   2890|    biting| 0.3979786668865754|
|   6279|     carlo|  2.785850668206028|
|  48043|     cures|0.10446940005772605|
|   8167|    curtiz| 1.8572337788040185|
```

Fig. 1- TF_IDF implementation code.

Movie similarity based on the Jaccard index

Figure 2 shows the code considered for this task and the partial printed output.

```
[ ] from pyspark.sql.types import *

    def jaccard(df):
      ji = []
      for i in range(df.count()):
        mp = df.collect()[i]
        intersection = spark.sql(
          '''
          SELECT userId from tags_ratings where tags_ratings.movieId = %s
          INTERSECT
          SELECT userId from tags_ratings where tags_ratings.movieId = %s
          ''' %(mp.mov1,mp.mov2)
        )
        union = spark.sql(
          '''
          SELECT userId from tags_ratings where tags_ratings.movieId = %s
          UNION
          SELECT userId from tags_ratings where tags_ratings.movieId = %s
          ''' %(mp.mov1,mp.mov2)
        )
        ji.append([mp.mov1,mp.mov2,intersection.count()/union.count()])

      cSchema = StructType([StructField("movie1", IntegerType()),StructField("movie2", IntegerType()),StructField("j_Index", DoubleType())])
      return spark.createDataFrame(ji, schema=cSchema)

    JaccardTable = jaccard(movie_pairs)
```

Fig. 2- Jaccard index implementation code.

Output data

Figure 3 presents the code for the output of the data obtained.

```
[ ] %%capture
    # Clean up first
    !rm -fr "$DATASET"/output
    !rm -f "$DATASET"/"$OUTPUT_ZIP_FILE"

    if DEBUG:
      !ls -l $DATASET

    if DEBUG:
      writeParquet(movies_agg, DATASET + '/output/' + 'movies_agg.parquet')
      writeParquet(TF_IDF2_MIN, DATASET + '/output/' + 'tf_idf.parquet')
      writeParquet(JaccardTable, DATASET + '/output/' + 'jaccard_index.parquet')

    if DEBUG:
      print('Creating ZIP file ...')

    !cd "$DATASET"/output  && zip -9qr ../"$OUTPUT_ZIP_FILE" .

    if DEBUG:
      !ls -l $DATASET "$DATASET"/output
```

Fig. 3- Code for the output data

Loader Cloud Function

Partial code inserted into the created Google Cloud Function, named LCF. Fig. 4 shows the code function related to the population of the tf_idf table while Fig. 5 shows the function code for the Jaccard index table population.

```
def load_tfidf_data(dataset_id):
  tid = 'tf_idf'
  table_name = '%s.%s.%s' % (PROJECT_ID, dataset_id, tid)

  # Read parquet file
  parquet_files_path = '%s/%s.parquet' % (TMP_DIR, tid)
  debug('Reading Parquet files from %s' % parquet_files_path)
  pdf = pd.read_parquet(parquet_files_path)
  debug(str(pdf.head(5)))


  # Create BigQuery table
  table = bq.Table(table_name)

  table.schema = (
        bq.SchemaField("movieId", "INTEGER", "REQUIRED"),
        bq.SchemaField("word",  "STRING", "REQUIRED"),
        bq.SchemaField("tf_idf", "FLOAT", "REQUIRED"),
  )
  debug('Creating %s' % table_name)
  BQ_CLIENT.create_table(table)

  debug('Populating %s with %d rows' % (table_name, len(pdf)))
  load_job = BQ_CLIENT.load_table_from_dataframe(pdf, table)
  while load_job.running():
     debug('waiting for load job to complete')
     time.sleep(1)

  debug('Done with table %s' % table_name)
```

Fig. 4 – Loading the tf_idf table.

```python
def load_jaccard_index(dataset_id):
  tid = 'jaccard_index'
  table_name = '%s.%s.%s' % (PROJECT_ID, dataset_id, tid)

  # Read parquet file
  parquet_files_path = '%s/%s.parquet' % (TMP_DIR, tid)
  debug('Reading Parquet files from %s' % parquet_files_path)
  pdf = pd.read_parquet(parquet_files_path)
  debug(str(pdf.head(5)))


  # Create BigQuery table
  table = bq.Table(table_name)

  table.schema = (
        bq.SchemaField("movie1", "INTEGER", "REQUIRED"),
        bq.SchemaField("movie2", "INTEGER", "REQUIRED"),
        bq.SchemaField("j_Index", "FLOAT", "REQUIRED"),
  )
  debug('Creating %s' % table_name)
  BQ_CLIENT.create_table(table)

  debug('Populating %s with %d rows' % (table_name, len(pdf)))
  load_job = BQ_CLIENT.load_table_from_dataframe(pdf, table)
  while load_job.running():
     debug('waiting for load job to complete')
     time.sleep(1)

  debug('Done with table %s' % table_name)
```

Fig. 5- Jaccard Index table loading.

LCF Cloud function

Fig. 6 shows some of the logs observed directly in the Google Platform for the LCF function while populating the medium1 dataset.

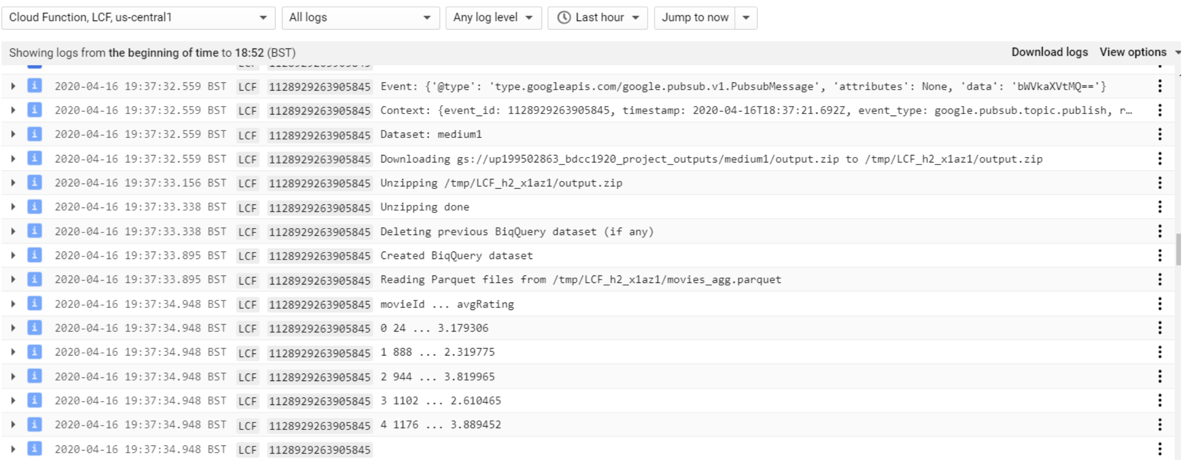

Fig. 6 – Log Views of the LCF function.

Search information in the created data sets

Fig. 7 shows the code used for movies recommendation based on the keyword search by tf_idf and the search for movie similarity (Jaccard index).

```
def list_tfidf(request):
  ds_id = '%s.%s' % (PROJECT_ID, request.args.get('dataset'))
  query = BQ_CLIENT.query(
      '''
      SELECT * FROM `%s.tf_idf`
      ORDER BY movieId,word
      LIMIT %s
      ''' % (ds_id, request.args.get('max_results')))
  df = query.to_dataframe()
  debug('Returning result with %d rows' % len(df))
  return df.to_html()

def tfidf_search(request):

  li = tuple(map(str, request.args.get('words').split(' ')))
  ds_id = '%s.%s' % (PROJECT_ID, request.args.get('dataset'))
  query = BQ_CLIENT.query(
      '''
      SELECT tf.movieId, ma.title, AVG(tf_idf) as Average_TFIDF FROM `%s.tf_idf` as tf,`%s.movies_agg` as ma
      WHERE word in %s AND tf.movieId = ma.movieId
      GROUP BY tf.movieId,ma.title
      ORDER BY Average_TFIDF DESC
      LIMIT %s
      ''' % (ds_id,ds_id, li, request.args.get('max_results')))
  df = query.to_dataframe()
  debug('Returning result with %d rows' % len(df))
  return df.to_html()
```

```
def jaccard_index_search(request):
  movieId = request.args.get('movieId')
  ds_id = '%s.%s' % (PROJECT_ID, request.args.get('dataset'))
  query = BQ_CLIENT.query(
      '''
      SELECT movie2 as Similar_Movie, j_Index as Jaccard_Index FROM `%s.jaccard_index`
      where movie1 = %s
      UNION ALL
      SELECT movie1 as Similar_Movie, j_Index as Jaccard_Index FROM `%s.jaccard_index`
      where movie2 = %s
      ORDER BY Jaccard_Index DESC
      LIMIT %s
      ''' % (ds_id, movieId, ds_id, movieId,request.args.get('max_results')))
  df = query.to_dataframe()
  debug('Returning result with %d rows' % len(df))
  return df.to_html()
```

Fig 7 – Recommendation code based on tf-idf and the Jaccard index based similarity search.

Fig. 8 shows the recommendation code based on the keyword search using the weighted search strategy.

```
def weighted_search(request):

  li = tuple(map(str, request.args.get('words').split(' ')))
  ds_id = '%s.%s' % (PROJECT_ID, request.args.get('dataset'))
  query1 = BQ_CLIENT.query(
      '''
      SELECT count(avgRating) as count from %s.movies_agg
      ''' % (ds_id))
  query2 = BQ_CLIENT.query(
      '''
      SELECT max(numRatings) as maximum from %s.movies_agg
      ''' % (ds_id))
  c = query1.to_dataframe().at[0,'count']
  m = query2.to_dataframe().at[0,'maximum']

  query3 = BQ_CLIENT.query(
      '''
      SELECT tf.movieId, movies.title,

      0.5*AVG(tf.tf_idf)/LOG(%s,2) +
      0.5*AVG(movies.avgRating)*LOG(SUM(movies.numRatings+0.01),2)/(5*LOG(%s,2))
      AS Average_Weights

      FROM %s.movies_agg movies JOIN %s.tf_idf tf ON (movies.movieId=tf.movieId)
      WHERE tf.word in %s
      GROUP BY tf.movieId, movies.title
      ORDER BY Average_Weights DESC LIMIT %s;
      ''' % (c,m,ds_id,ds_id,li,request.args.get('max_results')))

  df = query3.to_dataframe()
  debug('Returning result with %d rows' % len(df))
  return df.to_html()
```

Fig 8 – Recommendation code based on the weighted search.

Fig. 9 shows the top 7 recommendations using the tf_idf and as keywords: 'comedy + magic + children', for the medium1 dataset. For instance, the 7th recommendation is an 'Aladdin' movie that is indeed a magical comedy for children.

```
[ ] dataset = 'medium1' #@param ["tiny1", "tiny2", "tiny3", "tiny4", "medium1", "medium2", "
    words = 'comedy magic children'  #@param {type: "string"}
    max_results = 25 #@param {type:"slider", min:5, max:100, step:5}

    class TFIDFSearch:
       args = {
                'op': 'tfidf_search',       \
                'dataset': DATASET,         \
                'words': words,             \
                'max_results': max_results \
             }

    HTML(handle_request(TFIDFSearch()))
```

```
dataset: medium1, op: tfidf_search
Returning result with 25 rows
```

	movieId	title	Average_TFIDF
0	122982	Shônen Sarutobi Sasuke	4.035624
1	145964	Children of Eve	4.035624
2	177369	Tri tolstyaka	4.035624
3	183387	Sorochinskaya yarmarka	4.035624
4	153332	Snegurochka	4.035624
5	184963	Taking Flight	4.035624
6	138948	Aladdin and the Death Lamp	3.386295

Fig. 9 – Recommended movies from medium1 dataset with keywords: 'comedy + magic + children' and the TF-IDF metric.

Fig. 10, on its turn, shows the first 7 recommendations based on the same keywords (comedy + magic + children) as above, but now with the weighted search metric. Here, 50 % of the recommendation comes from the tf_idf value while the other 50 % comes from the ratings and number of ratings for the movies retrieved in the keyword search.

One can see that only one of the recommendations match the tf_idf search in the first 7 recommendations. Besides, the order is slightly different. However, for a higher number of recommendations there are more matchings (not shown). Also, in this search the first suggestion is not a movie for children, neither a comedy nor is magical. So, the choice probably was due to high rating of the movie and the number of ratings. In future searches, maybe reconsider the weights given to each parcel of the weighted search metric used. Despite of that, the movie Dogma that appears in number 2 and the movie 'Ted' that appears in number 4 are indeed fantasy comedy movies with good ratings.

```
[ ] dataset = 'medium1' #@param ["tiny1", "tiny2", "tiny3", "tiny4", "medium1", "medium2", "
    words = 'comedy magic children'  #@param {type: "string"}
    max_results = 25 #@param {type:"slider", min:5, max:100, step:5}

    class WeightedSearch:
       args = {
                'op': 'weighted_search',   \
                'dataset': dataset,        \
                'words': words,            \
                'max_results': max_results \
              }

    HTML(handle_request(WeightedSearch()))
```

```
dataset: medium1, op: weighted_search
Returning result with 25 rows
```

	movieId	title	Average_Weights
0	68157	Inglourious Basterds	0.492039
1	3052	Dogma	0.416312
2	2890	Three Kings	0.402918
3	95441	Ted	0.329642
4	177369	Tri tolstyaka	0.309873
5	105217	Zambezia	0.306577
6	45431	Over the Hedge	0.305170

Fig. 10 – Recommended movies from medium1 dataset with keywords: comedy + magic + children, and the weighted search metric.

Finally, Fig. 11 shows a simple query for the similarity search based on the Jaccard index for dataset tiny4, showing the 7 most similar movies to movieId=4. The movie most similar to movieId=4 is the movieId=46.

```python
dataset = 'tiny4' #@param ["tiny1", "tiny2", "tiny3", "tiny4", "medium1", "medium2", "me
movieId = 4  #@param {}
max_results = 100 #@param {type:"slider", min:100, max:1000, step:100}

class JISearch:
   args = {
            'op': 'jaccard_index_search',       \
            'dataset': dataset,         \
            'movieId': movieId,            \
            'max_results': max_results \

          }
HTML(handle_request(JISearch()))
```

```
dataset: tiny4, op: jaccard_index_search
Returning result with 49 rows
```

	Similar_Movie	Jaccard_Index
0	46	0.166176
1	27	0.106842
2	45	0.104952
3	20	0.097149
4	31	0.084462
5	22	0.082092
6	15	0.081314

Fig. 11- Similarity search based on the Jaccard index for dataset tiny4, showing the 7 most similar movies to movieId=4.