Instagram Search Engine

Fast and Scalable Inverted Index construction with Spark

Mirko Mantovani
Computer Science department
University of Illinois at Chicago
Chicago, Illinois
mmanto2@uic.edu

ABSTRACT

This document is a report for the final project of IDS 561 Big Data Analytics at University of Illinois at Chicago.

I decided to build a search engine for Instagram based on users' biography. Since Instagram itself only allows the finding of users whose username contains your query as a substring, finding users based on their biography, which in most cases contains important keywords, could be beneficial to users.

The repository containing the software can be accessed through GitHub at:

/mirkomantovani/Spark-Instagram-search-engine

1 SOFTWARE DESCRIPTION

It is clear that a big data approach is needed to solve this problem. Search engines achieve good performances by using an Inverted Index which, given a word (key), returns all the documents (values) which contain that word and their Term Frequencies.

The construction of an inverted index is an extremely parallel task and in principle, it would be possible to preprocess and index every profile present on Instagram in parallel. This suggests using a scalable and distributed framework to build the Inverted Index.

2 DATASET

As I mentioned in the project proposal, crawling data from Instagram is not an easy task, especially since, due to the recent privacy issues, Instagram limited the number of requests per hour using unofficial APIs to 200.
Therefore, to have significant amount of data, I deployed a Python script that used the InstagramAPI Python package on an EC2 instance of Amazon Web Services and let it run for a few days. I was able to collect 20950 unique profiles with non-null biography, a small dataset that was sufficient for the purpose of this demo project.

3 TECHNICAL DETAILS

3.1 Software used

I used Apache Spark and in particular, its Python implementation (PySpark) on the cloudera virtual machine that we used during the course. I used Python version 2.7 to write the code.

3.2 Preprocessing and user representation

The users' biography is initially in a text form. The initial preprocessing consists in creating the user representation following a bag of word approach and using their biography. The preprocessing is done by the function preprocess(doc) in preprocess.py. This named function is passed to the Spark mapValues, after a key-value pair RDD consisting of username-biography is built.

In particular, the preprocessing function does this: given the biography as a text string, it splits it on whitespaces, removes punctuations in each word, removes the digits, removes word which have a length less or equal to 2 and converts all the words to lowercase, then returns the list of words.

3.3 Building the inverted index

After the preprocessing step, the key-(list of tokens) pairs are transformed in key-word pairs using the Spark flatMapValues function. The next step was to invert key-value in value-key, in order to have a key which is the word and not the user anymore. The last step was to combineByKey, to build a word-(list of usernames) pair, which basically is our inverted index. I then computed the IDF of each word and created a TF-IDF function that uses the inverted index and the IDF to compute the TF-IDF of that word for a specific user.

3.4 Query lookup

At query time, the user inserts the query, which is preprocessed by the program, and each word present in the query is used as key to retrieve all the documents containing that word. Using the TF-IDF of each word, the cosine similarity between the query and each biography that contains at least one word present in the query is computed, the profiles are then ordered by cosine similarity and the results are displayed. The user can decide to show more results or open the profile of a user in the retrieved list.

4 RESULTS

Evaluating the results is not an easy task for this type of applications, because of the lack of labels in the data, but I manually tested it and the retrieved results are relevant for the query, which is usually very specific and short for this type of search engine.

Spark code

```
mimport os
import sys

os.environ['SPARK_HOME'] = '/usr/lib/spark'
os.environ['PYSPARK_PYTHON'] = '/usr/local/bin/python2.7'
os.environ['PYSPARK_SUBMIT_ARGS'] = ('--packages com.databricks:spark-csv_2.10:1.5.0 pyspark-shell')
os.environ['JAVA_TOOL_OPTIONS'] = "-Dhttps.protocols=TLSv1.2"

sys.path.append('/usr/lib/spark/python')
sys.path.append('/usr/lib/spark/python/lib/py4j-0.9-src.zip')

# Rerun to clear space (ctrl+F5), select all above this and execute in console option+shift+E, and go on execution in
# console the code
from pyspark import SparkContextfi
from pyspark import HiveContext
from preprocess import *
import math
import CustomGUI as gui
from collections import Counter
import operator
import webbrowser

RESULTS_PER_PAGE = 10
n_profiles = 20950

sc = SparkContext()
sqlContext = HiveContext(sc)

df = sc.textFile('file:///home/cloudera/Desktop/instagram_bio_dataset')
profiles = df.map(lambda line: (line.split(',')[1],line.split(',')[2])).mapValues(preprocess)
keyvalueflat = profiles.flatMapValues(lambda word: word)
invertkey = keyvalueflat.map(lambda (k,v): (v,k))

def to_list(a):
    return [a]

def append(a, b):
    a.append(b)
    return a

def extend(a, b):
    a.extend(b)
    return a

profiles_per_word = invertkey.combineByKey(to_list,append,extend)
```

```
inverted_i = profiles_per_word.collect()
inverted_index = {}

for w in inverted_i:
    word = w[0].encode('ascii')
    for u in w[1]:
        user = u.encode('ascii')
        inverted_index.setdefault(word, {})[user] = inverted_index.setdefault(word, {}).get(user, 0) + 1

# document frequency = number of docs containing a specific word, dictionary with key = word, value = num of docs
df = {}
# inverse document frequency
idf = {}

for key in inverted_index.keys():
    df[key] = len(inverted_index[key].keys())
    idf[key] = math.log(n_profiles / df[key], 2)

def tf_idf(w, doc):
    return inverted_index[w][doc] * idf[w]

def inner_product_similarities(query):
    # dictionary in which I'll sum up the similarities of each word of the query with each document in
    # which the word is present, key is the doc number,
    # value is the similarity between query and document
    similarity = {}
    for w in query:
        wq = idf.get(w, 0)
        if wq != 0:
            for doc in inverted_index[w].keys():
                similarity[doc] = similarity.get(doc, 0) + tf_idf(w, doc) * wq
    return similarity

def doc_length(userid):
    words_accounted_for = []
    length = 0
    for w in profiles[userid]:
        if w not in words_accounted_for:
            length += tf_idf(w, userid) ** 2
            words_accounted_for.append(w)
    return math.sqrt(length)

def query_length(query):
    # IMPORTANT: in this HW no query has repeated words, so I can skip the term frequency calculation
    # for the query, and just use idfs quared
    length = 0
    cnt = Counter()
    for w in query:
        cnt[w] += 1
    for w in cnt.keys():
        length += (cnt[w]*idf.get(w, 0)) ** 2
    return math.sqrt(length)

def cosine_similarities(query):
    similarity = inner_product_similarities(query)
    for doc in similarity.keys():
        similarity[doc] = similarity[doc] / doc_length(doc) / query_length(query)
    return similarity

def rank_docs(similarities):
    return sorted(similarities.items(), key=operator.itemgetter(1), reverse=True)

def new_query():
    query = gui.ask_query()
    if query is None:
        exit()
    # print(query)
    query_tokens = preprocess(query)
    ranked_similarities = rank_docs(cosine_similarities(query_tokens))
    handle_show_query(ranked_similarities, query_tokens, RESULTS_PER_PAGE)

def handle_show_query(ranked_similarities, query_tokens, n):
    choice = gui.display_query_results(ranked_similarities[:n], query_tokens)

    if choice == 'Show more results':
        handle_show_query(ranked_similarities, query_tokens, n + RESULTS_PER_PAGE)
    else:
        if choice is None:
            new_query()
        else:
            open_website(choice)

def open_website(url):
    webbrowser.open('https://www.instagram.com/'+url.split()[0]+'/', new=2, autoraise=True)

new_query()
sc.stop()
```