Advanced Big Data - Assignment #1

Maanit Mehra - mm4694@columbia.edu

February 18, 2016

The entire code for this Assignment has been uploaded to: https://github.com/maanitmehra/xzvf

Problem 1

Download and Install Spark. Learn how to use it.

`Solution`

Steps to complete installation:

```
wget http://shinyfeather.com/spark/spark-1.6.0/spark-1.6.0-bin-hadoop2.6.tgz
tar -xzvf spark-1.6.0-bin-hadoop2.6.tgz

sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer

echo "export PATH=$PATH:/home/ubuntu/spark-1.6.0-bin-hadoop2.6/bin" >> ~/.bashrc
source ~/.bashrc
```

Problem 2

Download Wikipedia dataset. Extract about 100 pages (items) based on your own interest. You may use snowball method to crawl a few related/linked pages. Create TF-IDF of each page.

Solution

Solutions to Q2 include two parts:

1. Webcrawling, cleaning files, parsing files
2. calculating the TF-IDF values for the above data.

Part 1

To do so, we first work with Python code for the actual cleaning and parsing of data.

The above approach involves the usage of a parser called BeautifulSoup that is used to clean up and acquire data from the html feeds. To actually acquire the feeds, we use Wikipedia's "Speacial:Random" link page which autogenerates random links. We call this a finite number of times using the `NUMBER_OF_PAGES` variable.

```
from bs4 import BeautifulSoup as bs
import os

NUMBER_OF_PAGES = 100
WIKI_RANDOM="https://en.wikipedia.org/wiki/Special:Random"
```

Following this we implement the following functions to each extract text, parse the XMLs of the wiki links & finally extract the URLs themselves.

```
def download_file(link, filename):
        try:
                file = open (filename, 'r+')
        except:
                file = open (filename, 'w')
        r = requests.get(link)
        file.write(r.text.encode('utf-8'))
        file.truncate()
        file.close()

def find_text(page, filename):
        file = fileOpen (filename)

        with open(page) as pag:
                soup = bs(pag,'xml')
```

```
        a=soup.get_text()
        start= a.find("From Wikipedia, the free encyclopedia")
        end= a.find("Retrieved fro")
        a = a[start:end]
        file.write(a.encode('utf-8'))
        file.truncate()
        file.close()

def find_urls(i, page, filename):
        file = fileOpen (filename)
        file.write("Doc_Num\tLink")
        with open(page, 'r+') as pag:
                soup = bs(pag.read(),'html.parser')
        all_links = soup.find_all('link')
        dict_array=[]
        for link in all_links:
                dict={'rel':link.get('rel'),'href':link.get('href')}
                dict_array.append(dict)
        for dict in dict_array:
                if dict['rel'][0]==u'canonical':
                        url= dict['href']
                        file.write("%d\t%s\n"%(i+1,url))
        file.close()
```

This generates a list of urls in a file called urlinks.txt. Below is an extract of that page. It extracts all 100 main links. The associated files are also extracted (not shown here, but attached in main project).

```
No. Link
1 https://en.wikipedia.org/wiki/Bourberain
2 https://en.wikipedia.org/wiki/Texas_State_Highway_11
3 https://en.wikipedia.org/wiki/Lenovo_IdeaPad_S100
4 https://en.wikipedia.org/wiki/Government_and_politics_of_Darien,_Connecticut
5 https://en.wikipedia.org/wiki/70_Sculptors
.
.
.
100 https://en.wikipedia.org/wiki/Kate_Sheil
```

Also included, a methodology to extract embedded links ('Snowballing'):

```
def find_Links(page, filename):
  file = fileOpen (filename)
  with open(page) as pag:
    soup = bs(pag.read(),'html.parser')
  all_anchors = soup.find_all('a')
  links=[]
  for link in all_anchors:
```

```
    list=link.get('href')
    if list:
      if 'html' in list:
        links.append(list.encode('utf-8'))
  file.write(str(links))
  file.truncate()
  file.close()
```

This approach generates a list of snowballed links for each page. These are further stored in individual files for reference.

For example, for the 11th Random link, the code generates `Random_11_links.txt` page with the snowballed links stored in an array.

```
['http://web.archive.org/web/20060515054228/www.menufoods.com/about_us/facilities.html']
```

`Part 2`

We proceed with the actual TF-IDF calculations. For the same we use the following code snippet, `Q2.py`

```

def TFIDF(source, destination):

  if destination[-1] != '/':
    destination=destination+'/'
  ## typically define the source message
  rdd=sc.wholeTextFiles(source).map(lambda (name,text): text.split())
  tf=HashingTF()
  tfVectors=tf.transform(rdd).cache()
  a = tfVectors.collect()
  # Storing the TF values above in individual files, one per link
  ind = 0
  for vector in a:
    dest_path = destination + "TF_%d"%ind + ".txt"
    ind = ind + 1
    file = open(dest_path,'w')
    file.write(str(vector))
    file.close()
```

The above approach yields a TF value for each page, stored in individual files. For example, `Random_23.html` generates a `TF_23.txt` page which stores the TF values.

The output for `TF_23.txt` is as follows:

```
(1048576,[10768,14726,27033,36757,48248,....1043653],
[3.0,1.0,1.0,2.0,1.0,....1.0])
```

Following this we proceed towards storing IDF values for the entire dataset in a file called `TF-IDF.txt`

```
# Calculating IDF Values for each case.
idf=IDF()
idfModel=idf.fit(tfVectors)
tfIdfVectors=idfModel.transform(tfVectors)
# Writing TF-IDF values to a single file.
file = open(destination+"TF-IDF.txt", 'w')
file.write(str(tfIdfVectors.collect()))
```

The output `TF-IDF.txt` looks as below.

```
[SparseVector(1048576, {10440: 11.7659, 11811: 4.1003,...., 1043949: 1.7078})]
```

The complete file has been provided separately.

Problem 3

Use Twitter Streaming API to receive real-time twitter data. Collect 30 mins of Twitter data on 5 companies using keyword=xxx (e.g., ibm). Consider all Twitter data from a company is one document. Create TF-IDF of each companys tweets in that 30 minutes.

Solution

Solutions to Q3 include two parts:

1. Streaming & Storing all the tweets in individual files per company
2. calculating the TF-IDF values for the above data.

Part 1

We use the `tweepy` library for this section to read streaming tweets.

This library is very well documented and we make use of the same. Following our authentication steps (not shown here, available in the source code), we begin by creating a class derived from the tweepy.Streaming class.

```
import tweepy
```

Within, we use the `on_data()` function to define the logic our code takes to actually acquire the data.

```
1
2  class StdOutListener(tweepy.StreamListener):
3      def __init__(self, runtime=1,company_list=['IBM', 'Google','Yahoo','Apple
       ','Facebook']):
4    self.time  =time.time()
5    self.limit =runtime*60
6    self.list = company_list
7
8      def file_write(self, company,a):
9    pass
10
11     def on_data(self, data):
12         # Twitter returns data in JSON format - we need to decode it first
13         decoded = json.loads(data)
14         # Also, we convert UTF-8 to ASCII ignoring all bad characters sent by
        users
15         try:
16            a= '@%s: %s' % (decoded['user']['screen_name'], decoded['text'].
      encode('ascii', 'ignore'))
17       for sym in self.list:
18           #print sym+ " is the company name"
19           filename = './Q3_files/'+sym+'.txt'
20           try:
21         file = open(filename,'a')
22           except:
23         file = open(filename,'w')
24
25           for line in a.split(' '):
26        if sym.lower() in line.lower():
27          print sym
28          file.write(str(a)+"\n")
29
30           file.close()
31          except Exception,e:
32             pass
33
34   if (time.time()-self.time > self.limit):
35     exit()
36         return True
37
38     def on_error(self, status):
39         print status
```

The actual `company_list` is fed from a csv file, that we also share with Q4. We call an instance of the above class and execute our program as below.

We use a `RUNNING_TIME` of 30 minutes for this exercise.

```
1    com_obj = StdOutListener(runtime=RUNNING_TIME, company_list=company_list)
2
```

```
    try:
    stream = tweepy.Stream(auth, com_obj)
      stream.filter(track=company_list, languages=['en'])
    except KeyboardInterrupt:
    pass
```

This generates a stream of files with tweets embedded in them. Consider for example, `Apple.txt` This includes a stream of tweets relating to Apple.

```
@Picardijn: RT @Puri_sm: Dear #Apple &amp; #Google:
#Privacy is more than just a marketing message.
#AppleVsFBI https://t.co/GUONNmOU6M
@Picardijn: RT @Puri_sm: Dear #Apple &amp; #Google:
#Privacy is more than just a marketing message.
#AppleVsFBI https://t.co/GUONNmOU6M
@buyamazingitems: Apple iPhone 4s - 32GB - Black (Unlocked) Smartphone https://t.co/tCF9c8w
```

`Part 2`

We proceed with the actual TF-IDF calculations. For the same we use the following code snippet, `Q3.py`, similar to the code snippet in Q2.

```

def TFIDF(source, destination):

  if destination[-1] != '/':
    destination=destination+'/'
  ## typically define the source message
  rdd=sc.wholeTextFiles(source).map(lambda (name,text): text.split())
  tf=HashingTF()
  tfVectors=tf.transform(rdd).cache()
  a = tfVectors.collect()
  # Storing the TF values above in individual files, one per link
  ind = 0
  for vector in a:
    dest_path = destination + "TF_%d"%ind + ".txt"
    ind = ind + 1
    file = open(dest_path,'w')
    file.write(str(vector))
    file.close()
```

The above approach yields a TF value for each page, stored in individual files. Each of the companies generate a txt file of tweets, below we store a TF file for each company.

For example one of the `TF.txt` files is as follows:

```
(1048576,[10768,14726,27033,36757,48248,....1043653],
[3.0,1.0,1.0,2.0,1.0,....1.0])
```

Following this we proceed towards storing IDF values for the entire dataset in a file called `TF-IDF.txt`

```
# Calculating IDF Values for each case.
idf=IDF()
idfModel=idf.fit(tfVectors)
tfIdfVectors=idfModel.transform(tfVectors)
# Writing TF-IDF values to a single file.
file = open(destination+"TF-IDF.txt", 'w')
file.write(str(tfIdfVectors.collect()))
```

The output `TF-IDF.txt` looks as below.

```
[SparseVector(1048576, {10440: 11.7659, 11811: 4.1003,...., 1043949: 1.7078})]
```

Problem 4

Use Yahoo Finance to receive the Stock price data. Collect 30 mins of Finance data on 5 companies, one value per minute. Use the outlier function to display outliers that are large than two standard deviation.

Solution

Similar to the earlier problems, we divide the problem into two parts:

1. Acquiring the Data from the Yahoo Finance Stream.
2. Creating an Outlier function for the same to generate Outliers and Cleaned data for all the data that comes in.

For the code below, we use a file titled `Yahoo_symbols.csv` which stores company names and their corresponding symbols.

```
SYMBOL COMPANY

AAPL Apple
GOOG Google
IBM IBM
MSFT Microsoft
YHOO Yahoo
```

```
ORCL Oracle
TWTR Twitter
GE GeneralElectric
HON Honeywell
BAC-PL BankofAmerica
FB Facebook
TSLA Tesla
GS-PC GoldmanSachs
```

Part 1

We begin this section working with the `yahoo-finance` API.

```
import yahoo_finance
from yahoo_finance import Share
```

In our functions we further look at extracting data off the Yahoo Finance stream as follows, while storing it in files titled by their company names.

```
        for i in range(0,TIME_IN_MIN*60/TIME_BETWEEN_ITERATIONS):
    reader=csv.DictReader(open('Yahoo_symbols.csv','rb'))
        for sym in reader:

    company=sym["COMPANY"]
    symbol=sym["SYMBOL"]

    filename = "./Q4_files/"+company+".csv"
    file = fileOpen(filename)

    share_name=Share(symbol)
    timestamp = share_name.get_trade_datetime()
    price = share_name.get_price()

    writer = csv.DictWriter(file, fieldnames=["Time","Price"], delimiter=",
", lineterminator="\n")
    writer.writerow({"Time":timestamp ,"Price":price})
    time.sleep(TIME_BETWEEN_ITERATIONS)

        file.close()
```

The files store data in the `.csv` file, with categories of `Timestamp, Price.`

For example, for `Google(sym:GOOG)`, we create a file called `Google.csv`

The output of the same is as follows:

```
2016-02-17 20:28:00 UTC+0000,707.00
2016-02-17 20:30:00 UTC+0000,707.216
2016-02-17 20:31:00 UTC+0000,706.80
```

.
.
.
2016-02-17 20:58:00 UTC+0000,708.05
2016-02-17 20:59:00 UTC+0000,708.06
2016-02-17 21:00:00 UTC+0000,708.40

Part 2

For this part the focus remains on creating an outlier function to distinguish "clean" & "outlier" data.

We define the outlier function as follows:

```
## Creating a SparkContext instance
sc = SparkContext()

## removeOutliers, as the name suggests, cleans the data, removing outliers. Is a function called by outlier function.
def removeOutliers(nums,number_of_std_devs):
        stats = nums.stats()
        sig = math.sqrt(stats.variance())
        cleaned= nums.filter(lambda x: math.fabs(x - stats.mean()) <= number_of_std_devs * sig)
        return cleaned

## arrOfOutliers, as the name suggests, creates a vector with stored outliers. Is a function called by outlier function.
def arrOfOutliers(nums,number_of_std_devs):
        stats = nums.stats()
        sig = math.sqrt(stats.variance())
        outliers= nums.filter(lambda x: math.fabs(x - stats.mean()) > number_of_std_devs * sig)
        return outliers

## this outlier function was created with the spark code.
## We parallelize the data here and make use of it in our calculations.
def outlier(arr, number_of_std_devs):
        nums = sc.parallelize(arr)
        val = sorted(removeOutliers(nums,number_of_std_devs).collect())
        out = sorted(arrOfOutliers(nums,number_of_std_devs).collect())
        return val, out
```

We store this response in a `csv` file. For example, for `Google.csv` above, we save a file as `Google_cleaned.csv` (Note: not all values are shown.)

```
Google
Cleaned Data: [701.08, 701.5, 701.27...700.44, 700.68]
Outliers: [700.14, 700.071]
```