# HW8 CS432

Ondra Torkilson

December 6, 2020

# Q1

To start this homework, I manually compiled 100 verified twitter accounts with greater than 10,000 followers and greater than 5000 tweets into the text file, 'users.txt' which can be seen in the github for this project.

To collect the tweets, I will discuss the code listing below which references my script 'getTweets.py'. This file is also available on the github. I used my credentials for the Twitter API to authenticate with tweepy. Then, using tweepy, I accessed the tweets of each user from my 'user.txt,' collecting their latest 200 tweets into an object. Then, I sent each tweets object to a file line for line for later parsing.

```
import tweepy

consumer_key = " "
consumer_secret = " "
access_key = " "
access_secret = " "

auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_key, access_secret)
api = tweepy.API(auth)

filepath = 'fixedUsers.txt'
with open(filepath, 'r', encoding="utf-8") as fp:
    line = fp.readline()
    cnt = 0
    while line:
        #print(line)
        tweets = api.user_timeline(screen_name=line, count=200,include_rts =
                                                    False,tweet_mode = 'extended')
        user=line
        line = fp.readline()
        cnt += 1
        filepath2 = str(cnt)+'tweets.txt'
        with open(filepath2, 'w', encoding="utf-8") as file:
            file.write(user+'\n')
            for item in tweets:
                file.write(item.full_text+'\n')
```

# Q2

Question two had a lot of scripts, so bear with me. There are separate scripts to parse the tweets for each account, count the total unique words and fake removing the stopwords, create the dictionary of 1000 words with the number of accounts it was present in, create a nested dictionary that counts each word within each account and draws the account matrix, and converge the accounts' tweets. Okay, so that was five separate scripts for question 2!

The script for parsing the tweets from each account is on the following page, but I will discuss it now. I defined all the functions for what I needed to do to parse the tweets at the top of the script, and then I put it all together at the bottom of the script as a function 'parseFiles.' For the functions 'removeURIS,' removeUserNames,' and 'removeShortAndLong,' I used regular expressions to match the words needed for removal. For removing non-ascii words, I had to split the tweets string into words, then run the 'containsNonAscii' function to determine which words to exclude from the new tweets string. To remove punctuation, I used a string translator to replace punctuation characters with null. To finish, I wrote the parsed tweets to the same file it came from as an overwrite. I already saved the not parsed tweets in a different place. Both the parsed files and the not parsed files for each account can be accessed in the github.

```
def toLowercase(tweetFile):
    tweetFile = tweetFile.lower()
    return tweetFile

def removeURIS(tweetFile):
    tweetFile = re.sub(r"http\S+", "", tweetFile)
    return tweetFile

def containsNonAscii(s):
    return any(ord(i)>127 for i in s)

def removeUserNames(tweetFile):
    tweetFile = re.sub(r'@(\w+)', '', tweetFile, flags=re.MULTILINE)
    return tweetFile

#this should come after removeUserNames, because it also removes @ symbol,
# and I want the whole username removed, not just the symbol
def removePunctuation(tweetFile):
    translator = str.maketrans('', '', string.punctuation)
    tweetFile = tweetFile.translate(translator)
    return tweetFile

def removeShortAndLong(tweetFile):
    tweetFile = re.sub(r'\b\w{1,2}\b', '', tweetFile)
    tweetFile = re.sub(r'\b\w{16,}\b', '', tweetFile)
    return tweetFile

def parseFiles():
    directory = 'tweetsFrom2Users'
    for filename in os.listdir(directory):
        f=open('tweetsFrom2Users/'+filename, 'r+', encoding="utf-8")

        tweets = f.read().replace('\n', ' ')
        #print(tweets)

        tweets = removeURIS(tweets)
        tweets = removeUserNames(tweets)
        tweets = removePunctuation(tweets)
        tweets = removeShortAndLong(tweets)
        tweets = toLowercase(tweets)

        words = tweets.split()
        cleaned_words = [word for word in words if  not containsNonAscii(word)]
        cleaned_tweets = ' '.join(cleaned_words)

        f.seek(0)
        f.write(cleaned_tweets)
        f.truncate()

parseFiles()
```

The script on the following page first finds the unique words from all the accounts tweets using a set (words cannot be duplicated). There was about 1350 unique words from all the tweets across all accounts. Then, I counted how many blogs each unique word appeared in, and I created a dictionary called apcount to hold the word (key) and number of accounts it appeared in (value.) To see whether an account contained a word, I used the function seen at the top of the script called contains_words. This is sure to exclude instances where a word is inside of another word by specifying that the match has to be surrounded by spaces.Once I had that dictionary of words and account counts, I was able to fake removing stop words, using the code from the PCI example. Then, I saved the unique words to a list which I will use in the next script to find the most tweeted words.

```
#checks to see if a string of text contains a word
#excludes words within words (example: 'top' will return false for string 'stop')
def contains_word(s, w):
    return (' ' + w + ' ') in (' ' + s + ' ')

#get unique words from all tweets
uniqueWords = set()
directory = 'tweetsToParse'
for filename in os.listdir(directory):
    f=open('tweetsToParse/'+filename, 'r', encoding="utf-8")
    for line in f:
        for word in line.split():
            uniqueWords.add(word)

#count how many blogs each unique word appears in
#make dictionary apcount with unique word as key and number of accounts as value
apcount = {}
for word in uniqueWords:
    numberOfAccounts = 0
    for filename in os.listdir(directory):
        f=open('tweetsToParse/'+filename, 'r', encoding="utf-8")
        tweets = f.read()
        if contains_word(tweets,word) == True:
            numberOfAccounts+=1
        else:
            continue
    apcount[word] = numberOfAccounts

#create a list of words that appear in greater than 10% of accounts, but less that 50% of accounts
wordlist=[]
for w,c in apcount.items():
  frac=float(c)/100
  if frac>0.1 and frac<0.5:
    wordlist.append(w)
pp.pprint(wordlist)

#write wordlist to file for later use
writeTo = open("wordList2.txt","w+")
for word in wordlist:
    writeTo.write(str(word)+'\n')
```

So, to find the most tweeted, unique words across all accounts, I used the string containing all the tweets converged from a separate script (convergeTweets.py) as well as the unique words list. I created a dictionary with each unique word as the key, and the total count across all tweets as its value. I then sorted it by value (using a list object representation of the dictionary), reversed order, and truncated the list object to only keep the range from 0 up to 1000, thus keeping the 1000 most tweeted words from the unique word list. I saved this to the file 'mostTweetedWords2.txt', which is in the github repo.

```
allTweetsFile = open('allTweets2.txt', 'r', encoding="utf-8")
allTweets = allTweetsFile.read()
wordListFile = open('wordList.txt', 'r')
numberOfOccurences = {}

#stackoverflow
def count_occurrences(word, sentence):
    return sentence.lower().split().count(word)

for line in wordListFile:
    line = line.rstrip()
    totalTimes = count_occurrences(line, allTweets)
    numberOfOccurences[line] = totalTimes

sorted_occurrences = sorted(numberOfOccurences.items(), key=operator.itemgetter(1), reverse=True)
mostTweetedWords = sorted_occurrences[0:1000]
#pp.pprint(mostTweetedWords)
listWithQuotes = [i[0] for i in mostTweetedWords]
listOf1000words = [i.replace('"', '') for i in listWithQuotes]

with open('mostTweetedWords2.txt', 'w+') as fp:
    [fp.write(i+'\n') for i in listOf1000words]
```

Next is the most important part of the question; creating the account matrix. Things got a lot easier from this point forward in the project. By far, parsing the tweets, fake removing the stopwords, and counting the number of accounts per word and the total instances of the word across all tweets took me a really long time. I may have overcomplicated things, and part of the problem may be that I did not start using the colab and book references until this part in the code. I am glad I did, because I discovered that I needed a nested dictionary for the code from PCI to build the matrix.

So, I read in the most tweeted words from the saved file and stripped the newline characters. I created a dictionary called wordcounts. Within a for-loop, I open the file for each account's tweets to count how many times each of the words from wordList is mentioned. For each account, a dictionary is created where each word is a key and the number of times it appears in that account's tweets is the value. To get the name of each account, I had saved the account name at the start of the file. I accessed it by splitting the tweets string, and accessing the first element. From here, I constructed the account matrix using the code from PCI. It was pretty much plugging in my variables and naming the file to my liking.

```
def count_occurrences(word, sentence):
    return sentence.lower().split().count(word)

# reading the data from the file
with open('mostTweetedWords2.txt') as f:
    wordList = f.readlines()
    wordList = [x.strip() for x in wordList]

wordCounts = {}
directory = 'tweetsToParse'
for filename in os.listdir(directory):
    f=open('tweetsToParse/'+filename, 'r', encoding="utf-8")
    tweets = f.read()
    accountName = tweets.split()[0]
    wordCounts[accountName] = {}
    for word in wordList:
        count = count_occurrences(word, tweets)
        wordCounts[accountName][word] = count

out=open('accountdata2.txt','w')
out.write('Account')
for word in wordList: out.write('\t%s' % word)
out.write('\n')
for account,wc in wordCounts.items():
  #deal with unicode outside the ascii range
  account = str(account)
  out.write(account)
  for word in wordList:
    if word in wc: out.write('\t%d' % wc[word])
    else: out.write('\t0')
  out.write('\n')
```

## Q3, Q4, Q5

These questions were much easier for me than the previous two, because the code was already written. Only thing I had to really do was make sure I understood how all of the functions interacted which was laid out clearly by both the week 12 slides, the colab notebook, and chapter 3 of the PCI book.

The script 'dendrogram.py' is below, I used all of the functions from the colab notebook, authored both by Dr. Weigle and from Segaran's book.

For questions 3, 4, and 5 I used the readfile composed by Dr. Weigle to read in my tab-separated matrix of accounts vs. words. This is seen at the top of the script on the first line of code. I clustered the data using the hcluster function. The results of this can be seen in Figure 1 on the next page.

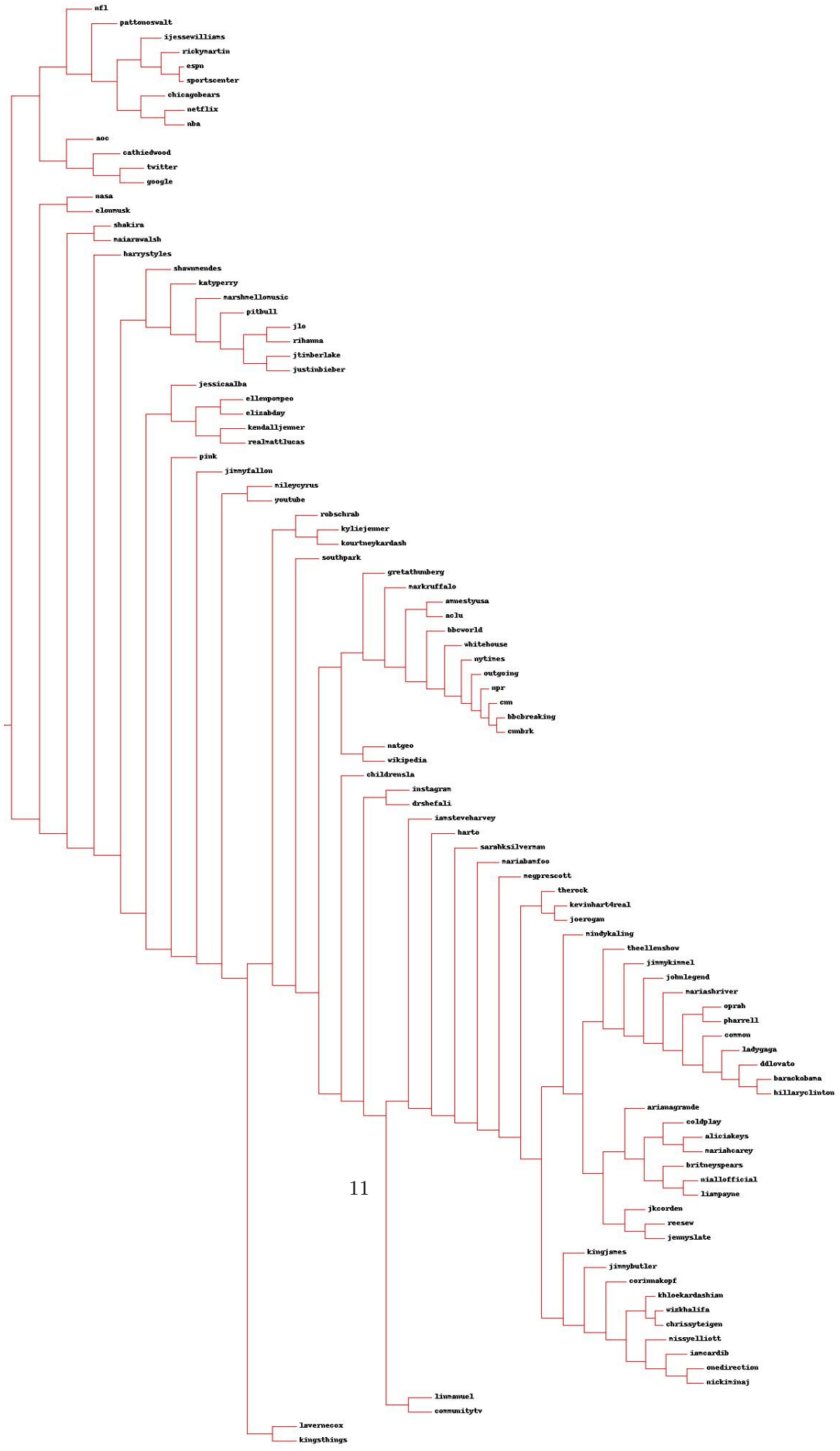




Figure 1: JPEG Dendrogram of Accounts

For question 4, I used the function kcluster, and I changed the value of k from 5 to 10 to 20. Five clusters took six iterations, ten clusters took four iterations, and twenty clusters took five iterations. Interestingly, the ten clusters seemed to me to be the most reasonable, and it took the least iterations. I think this was the best clustering, because this is where I saw the most characterized groups. One group contained NFL and ChicagoBears. Another contained all the news accounts. A different group seemed to be all musicians. Another group got natgeo and nasa together. . Although I thought this was the best clustering, it still has a lot of random placements. Overall, I think that clustering based on tweeted words is not super accurate, unless you are clustering non-personal accounts. This is why the news sites always lumped together so nicely, I think.

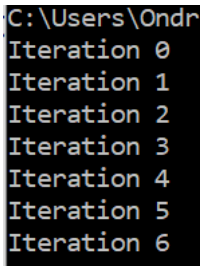


Figure 2: k-Means 5 Clusters

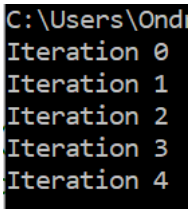


Figure 3: k-Means 10 Clusters

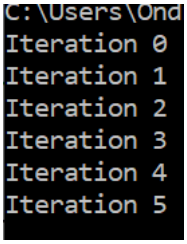


Figure 4: k-Means 20 Clusters

For question 5, I used the build-in scaledown function, and the draw2d function to render the 2d image of the data. That image can be seen below. Seeing the data like this shows me why the k-Means clustering had a hard time finding meaningful clusters.

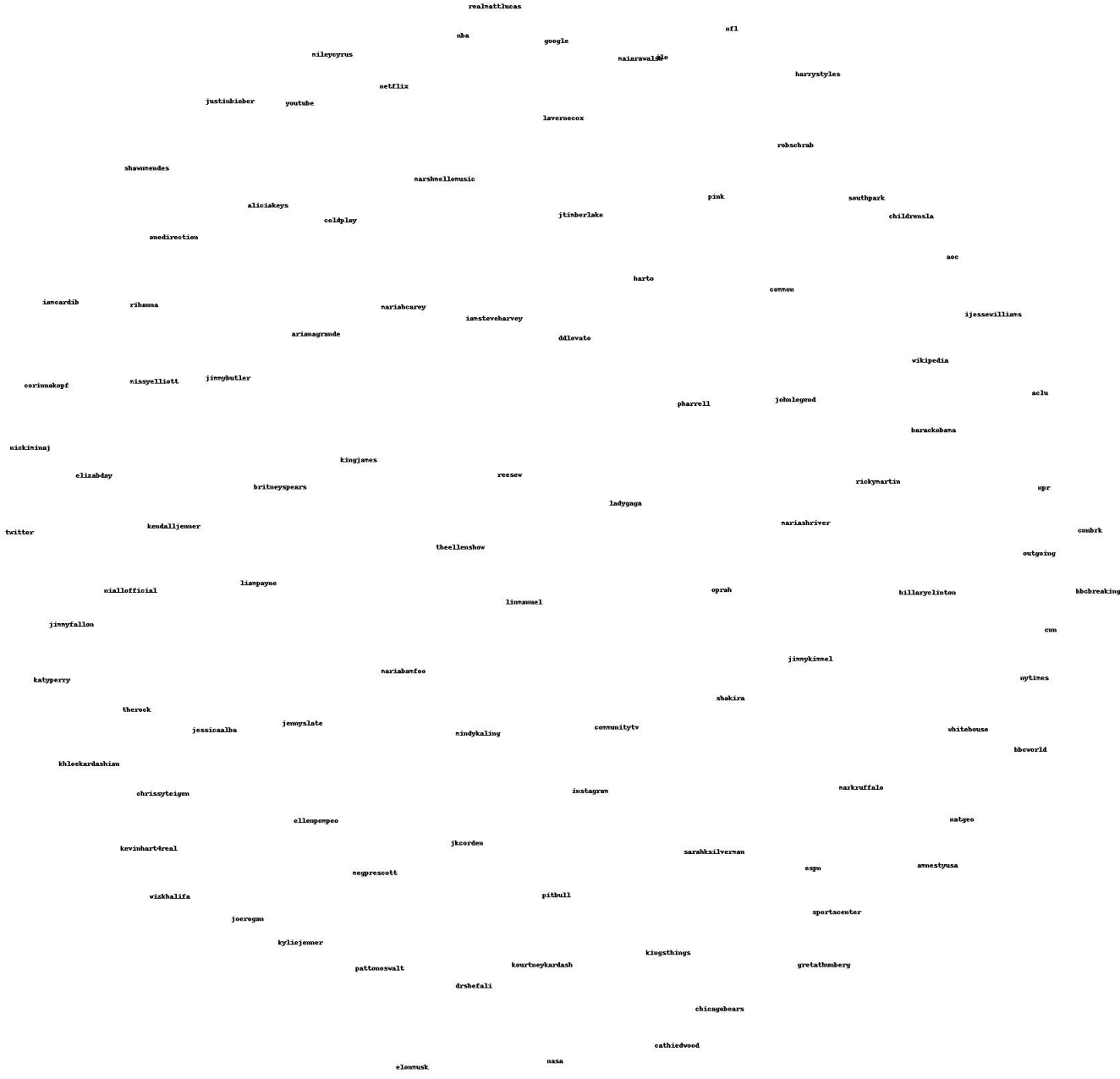




Figure 5: MDS of Accounts

```
accounts, words, data = readfile('accountdata2.txt')

#5
mds = scaledown(data)
draw2d(mds, accounts, jpeg='mds.jpg')

#3&4
asciiCluster = hcluster(data)

#4
kclust = kcluster(data, k=10)

for i in range(len(kclust)):
    sys.stdout = open("kcluster20.txt", "a")
    print("cluster ", i+1, ": ", len(kclust[i]))
    print([accounts[r] for r in kclust[i]])
    sys.stdout.close()

#3
drawdendrogram(asciiCluster, labels=accounts, jpeg='clusters.jpg')
sys.stdout = open("asciiCluster2.txt", "w+")
printclust(asciiCluster, labels=accounts)
sys.stdout.close()
```

# References


[1] Segaran, T. (2007). Programming Collective Intelligence. O'Reilly Media. Retrieved December 6, 2020, from https://learning.oreilly.com/library/view/programming-collective-intelligence/9780596529321/ch03.htmlsupervised$_versus_unsupervised_learning$

[2] Weigle, M. (2020). 432-PCI-Ch03.ipynb. Retrieved December 6, 2020, from https://colab.research.google.com/github/cs432-websci-fall20/assignments/blob/master/432$_PCI_Ch03.ipynbscrollTo =_u -CWzEYkILC$