HW8 - Clustering

Brenden Lewis
Due: 12/6/2020 11:59PM

Q1

Listing 1 below shows the Python code used to collect Twitter data needed for the assignment. I utilized the Tweepy library to collect tweets across my chosen accounts. When choosing accounts to collect data from, I initially looked at a list of the Top 100 most followed accounts of all time on Twitter and started from there. The lists I looked at had total tweet counts across the accounts and they all happened to be verified, so I chose all the accounts that fit the criteria; after filtering the Top 100 list, I had about 40 accounts collected.

I was not able to figure out how to automate finding accounts through Tweepy, so I opted to get the remaining accounts *semi*-manually. What I mean by that is I had to identify accounts that I knew were extremely popular, checking if they were verified, and double-checking their follower counts. These accounts included musicians, celebrities, businesses, restaurants, sports players and networks, streaming services, and random comedy accounts. I did this in batches, as I had to take screen names of the accounts and run them through tweepy to check their tweet counts; this was handled in the *collectAccounts()* method. Each account I scanned that were over the 5000+ tweets threshold was added to the collection.

```
import tweepy
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream
import json

# Keys ommitted
consumer_key="***"
consumer_secret="***"
access_token="***"
access_secret="***"

# Handles authorization with Twitter
auth = OAuthHandler(consumer_key,consumer_secret)
auth.set_access_token(access_token,access_secret)
api = tweepy.API(auth, wait_on_rate_limit=True,
    wait_on_rate_limit_notify=True)
#api = tweepy.API(auth)

accounts=[]
tweets_per_account = []

```

```

def retrieveTweets():
    count=1
    for user in accounts:
        print("Processing Account #"+str(count)+": "+user)
        tweets = []
        statuses = api.user_timeline(screen_name=user,count=200, tweet_mode="extended")
        for status in statuses:
            #tweet_dict = {"USER":user,"ID":status.id,"TEXT":status.full_text}
            tweet_dict = {"ID":status.id,"TEXT":status.full_text}
            tweets.append(tweet_dict)

        writeTweetsToFile(user,tweets)
        #writeTweetsToJSON(user, tweets)
        #account_dict = {"USER":user, "TWEETS":tweets}
        account_dict = {user: {tweets}}
        tweets_per_account.append(account_dict)
        print("Done"+'\n')
        count+=1


def writeTweetsToFile(account,tweets):
    path = r"C:\Users\Brenden\Desktop\ODU\Fall 2020\CS432\HW_8\hw8-cluster-blewis1014\account_tweets\tweets_{}.txt".format(account)
    fileToWrite = open(path,"a", encoding="utf-8")
    for tweet in tweets:
        fileToWrite.write(str(tweet["ID"])+"\n")
        fileToWrite.write(tweet["TEXT"]+"\n\n")
    fileToWrite.close()

def writeTweetsToJSON():
    with open("H9_tweets.json","w") as f:
        json.dump(tweets_per_account, f, indent=2)

def readAccounts():
    with open("100Accounts.txt",'r') as f:
        for line in f:
            line = line.rstrip('\n')
            accounts.append(line)

def collectAccounts():
    user_ID = ""
    #user_IDs = ["","","",
    #           "","","",
    #           "","","",
```

```
    #            "", "", "",
    #            "", "", ""]

    user = api.get_user(user_ID)
    #users = api.lookup_users(screen_names=user_IDs)

    #for user in users:
    #    tweet_count = user.statuses_count
    #    print(user.screen_name+"\n"+"Tweets: "+str(tweet_count))
    #    print("Followers: "+str(user.followers_count)+"\n")

    tweet_count = user.statuses_count
    print(user.screen_name+"\n"+"Tweets: "+str(tweet_count))
    print("Followers: "+str(user.followers_count)+"\n")

if __name__ == '__main__':
    #collectAccounts()
    readAccounts()
    retrieveTweets()
    writeTweetsToJSON()
```

Listing 1: Python code used to collecting tweets for Q1

Listing 2 below shows the final list of accounts used for tweet processing. This collection was processed in the *retrieveTweets()* method to collect the 200 most recent tweets from each account. When pulling tweets from each account's timeline, I found I had to add the parameter *tweet_mode = "extended"* to the *api.user.timeline()* call to pull the full tweet, otherwise only a snippet of the tweet would be pulled.

```
BarackObama
justinbieber
katyperry
rihanna
realDonaldTrump
ladygaga
ArianaGrande
TheEllenShow
YouTube
KimKardashian
selenagomez
CNN
ddlovato
shakira
jimmyfallon
KingJames
nytimes
MileyCyrus
JLo
```

```
Oprah
NASA
BeingSalmanKhan
elonmusk
KevinHart4real
wizkhalifa
KylieJenner
espn
KendallJenner
aliciakeys
HillaryClinton
pitbull
NatGeo
coldplay
Google
MariahCarey
NICKIMINAJ
davidguetta
ricky_martin
JERICHO
SavinTheBees
NBA
WHO
Ninja
PostMalone
TheDailyShow
FoxNews
matthewmercer
hulu
netflix
adamlevine
NFL
MLB
TheRock
WNBA
Sony
Microsoft
Windows
NHL
NintendoAmerica
NintendoUK
PaulMcCartney
FIFAcom
EA
Blizzard_Ent
RealRonHoward
BigMikeJ73
```

```
michaelb4jordan
BET
BETMusic
bethesda
WuTangClan
AOC
NathanFillion
ToddHaberkorn
TroyBakerVA
WayneBrady
Crunchyroll
Wendys
dominos
tacobell
McDonalds
BurgerKing
PapaJohns
littlecaesars
pizzahut
MrPeanut
RoosterTeeth
Monstercat
PegboardNerds
StephenAtHome
KenJennings
TheTweetOfGod
TheOnion
Bungie
Treyarch
Respawn
Charmin
marshmellomusic
MerriamWebster
BaskinRobbins
```

Listing 2: 100 Accounts used

Each tweet was stored as a dictionary containing the tweet ID and the full body of text. This dictionary was then appended to a list for each user, which then another dictionary was created for each user containing their screen name and list of tweets. After all the tweets were pulled for a single account, the screen name and tweets were passed to *writeTweetsToFile()* to automatically write a text file containing each tweet for each account. Each of these final user collections were then combined to a single list of dictionaries to exported to JSON for processing later.

Listing 3 below shows the first few tweets pulled from Barack Obama's account @*Barack-Obama* and their IDs.

```
1336679633769680898
```

```
In A Promised Land, I talk about the decisions I had to make during the first few years of my presidency. Here are some thoughts on how I approach tough questions: https://t.co/GFumGnNPln

1336370071917240321
With COVID-19 cases reaching an all-time high this week, we've got to continue to do our part to protect one another. This pandemic is far from over and your actions can help save lives. https://t.co/XJ4dbsCihB

1335954913344565258
To all of you in Georgia, today is the last day to register to vote in the upcoming runoff election. Take a few minutes right now to register to vote, and then make sure everybody you know is registered, too. https://t.co/2Ob571igh3 https://t.co/SslnFSWMfi
```

Listing 3: Sample of tweets pulled from Barack Obama's account

Q2

Listing 4 below is code used for processing the collected tweets, their text, collecting the final data needed to form the account-term matrix.

The first step was to read in the accounts and JSON file containing the tweets. Once read-in, the method *matchTweets()* was used to organize the tweets by creating a nested list *tweets_per_account* containing each list of 200 tweets with the same index as the account the came from in the *accounts* list (so tweets_per_account[0] contains the list of tweets from the account at accounts[0])

The next step was to simultaneously gather the words from each tweet, clean them, and store them in another nested list *words_per_account* to store the tweet words with their respective account index. Cleaning the tweets means removing any links, punctuation, account mentions, emojis, and any non-alpha character. The *cleanText()* method was called for each tweet, which lower cased each tweet, used regular expressions to remove the unneeded content, and ran an additional check to remove words less than 3 characters and more than 15 characters. The clean collection of words is then appended to *words_per_account* before moving onto the next user's tweets.

Once the the tweets were cleaned and there was collection of words for each account, I needed to fake removing stopwords with the *fakeStopwordsTFIDF()* method. But before I could call that method, I needed a dictionary containing each word and how many accounts had tweets containing that word for calculation; *getAPCount()* was used to build this dictionary. The easiest way I found was to use list comprehension to build a list of all words per account excluding duplicate entries and using *dictionary.get()* to either add the word with a default count to the dictionary or increment the count of the word if it already exists in the dictionary. This final dictionary was then sorted in descending order by the number of accounts each word appeared in.

The *apcount* was then passed into *fakeStopwordsTFIDF()*. This method simulated TFIDF calculations for each word to remove stopwords from the overall list of words. This returned a new list containing words that appeared in more than 10% and fewer than 50% of the accounts. With this final list of words, it was time to count the frequency of each word across all tweets using *countFinalWords()*. Using similar logic to getting *apcount*, each final word was compared across all tweets per account and appended to a new dictionary using *dictionary.get()* to increment the frequency for duplicate occurrences.

Of note, I'm aware my method for iterating through each word is extremely inefficient, but by this point it was much faster for me to process it the way I did than to go back and properly reorganize my data structures.

Finally, this dictionary of words was sorted by frequency counts in descending order. From this sorted dictionary, the first 1000 terms (top 1000 most common words) were stored in a final dictionary. With this list, the *createMatrix()* method was called to create a two-dimensional list *matrix* containing rows with the final words and their frequency across each account. For each account, a *row* dictionary was created containing each of the 1000 terms and a default value of 0. Cross referencing each term within *row* with all words across all tweets from the current account, the term's frequency count was incremented each time a match occurred. This row was then appended to *matrix*.

```
import json
import re
import nltk
from nltk.corpus import stopwords
from collections import Counter
from operator import itemgetter
from progress.bar import IncrementalBar
import itertools

accounts = []    # [User1, User2, ...]
tweets_per_account = [] # [User1Tweets, User2Tweets, ....]
                        # {USER: TEXT:},
                        # {USER: TEXT:}

words_per_account = []  # [User1[], User2, ....]
                        # {word1,
                        #  word2,
                        #  ...},

all_words = []       # {"WORD":word, "COUNT":count_of_accounts_feat}
final_words = []
# Need to save to file
common_words_per_account = []    # [User1[], User2[], ....]
                                 # [(word1,count),
                                 #  (word2,count),
                                 #  ...],
```

```

accounts_with_common = [] # {USER: user, "WORD":word, "OCCUR":
    count_of_accounts_feat}}

common_all_words = {}  # Top 1000, used for Matrix
apcount={}
final_common = {}
matrix_words = []
matrix = []

def process():
    global apcount, common_all_words, final_common
    data = readJSONFile()
    matchTweets(data)
    print("Getting words per account...")
    for user in tweets_per_account:
        clean_words=[]
        for tweet in user:
            clean_words.append(cleanText(tweet["TEXT"]))
        words_per_account.append(clean_words)

    print("Getting apcount...")
    apcount = getApCount(words_per_account)
    print("Getting all words...")
    populateAllWords()

    print("Removing stopwords per account...")

    fakeStopwordsTFIDF(apcount)
    countFinalWords()

    sorted_common = dict(sorted(common_all_words.items(), key=
    itemgetter(1),reverse=True))

    final_common = dict(itertools.islice(sorted_common.items(),1000))

    createMatrix()

def createMatrix():
    fwords = list(final_common.keys())
    temp = []
    for user in words_per_account:
        row = {x:0 for x in fwords}
        #row = {account: {x:0 for x in fwords} for account in accounts}
        for key,value in row.items():
            for words in user:
                for word in words:
```

```
                    if word == key:
                        row[word]+=1
        matrix.append(row)

    #for user in temp:
        #row = {account: {word:key for word,key in user.items()} for account in accounts}
        #matrix.append(row)

def countFinalWords():
    global common_all_words
    #top_words = {}
    bar = IncrementalBar('Counting', max=len(final_words))
    for word in final_words:
        for user in words_per_account:
            for w in user:
                for x in w:
                    if word == x:
                        common_all_words[word]=common_all_words.get(word,0)+1
        bar.next()
    bar.finish()

def populateAllWords():
    for user in words_per_account:
        for words in user:
            for word in words:
                all_words.append(word)

def getApCount(words_list):
    global common_words_per_account
    ap = {}
    for user in words_list:
        common_words = []
        words_no_dupes = []
        for words in user:
            [words_no_dupes.append(x) for x in words if x not in words_no_dupes]

            temp = sorted(getMostCommonTerms(words),key=itemgetter(1), reverse=True)

            common_words.append(temp)

        for word in words_no_dupes:
            ap[word]=ap.get(word,0)+1

```

```
        #sorted_common = common_words.sort(key=itemgetter(0))
        common_words_per_account.append(common_words)

    return dict(sorted(ap.items(), key=itemgetter(1),reverse=True))

def getMostCommonTerms(list):
    common_words = Counter(list)

    most_common = common_words.most_common(1000)

    return most_common

def matchTweets(data):
    global tweets_per_account
    for account in accounts:
        tweets_from_account = []
        for item in data:
            if account == item["USER"]:
                for tweet in item["TWEETS"]:
                    tweet_dict = {'USER':account,'TEXT':tweet["TEXT"]}
                    tweets_from_account.append(tweet_dict)
        tweets_per_account.append(tweets_from_account)

def matchCommonTerms():
    for index, user in enumerate(accounts):
        for words in common_words_per_account[index]:
            for word in words:
                word_dict = {"USER":user,"WORD":word[0],"FREQ":word[1]}
                accounts_with_common.append(word_dict)

def readJSONFile():
    with open("h9_tweets.json",'r') as f:
        data = json.load(f)
    return data

def readAccounts():
    with open("100Accounts.txt",'r') as f:
        for line in f:
            line = line.rstrip('\n')
            accounts.append(line)

def cleanText(text):
    clean_text = text.lower()

    clean_no_mention = re.sub(r"@[\w]+","",clean_text) # remove @mentions

```

```
    # remove links, punctuation, emojis, and hashtags
    clean_no_chars = " ".join(re.sub(r"([^0-9A-Za-z \t])|(\w+:\/\/\S+)", "", clean_no_mention).split())

    text_word_list = clean_no_chars.split()

    clean_words_list = []

    for word in text_word_list:
        if len(word) >= 3 and len(word) <= 15:
            clean_words_list.append(word)

    return clean_words_list

def fakeStopwordsTFIDF(apcount):
    global final_words
    #num_accounts = 100

    bar = IncrementalBar('Processing', max=len(apcount))
    for word,ac in apcount.items():
        frac = float(ac) / 100
        if frac>0.1 and frac<0.5:
            final_words.append(word)
        bar.next()
    bar.finish()

def writeFinalWordsToFile():
    with open("FinalWords.json","w") as f:
        json.dump(final_common,f,indent=2)
        #for word in final_common:
            #f.write(word+"\n")

def readInFinalWords():
    with open("FinalWords.json",'r') as f:
        data = json.load(f)

    for key,value in data.items():
        temp_dict = {"WORD":key,"COUNT":value}
        matrix_words.append(temp_dict)

def writeCommonTermsPerAccount():
    with open("Common_Terms_Per_File.txt","w") as f:
        for item in accounts_with_common:
            f.write("User: "+item["USER"]+"\n")
            f.write("Word: "+item["WORD"]+"\n")
            f.write("Occurences across account: "+str(item["FREQ"])+"\n\n")
```

```python

def writeMatrixData():
    with open("MatrixData.json","w") as f:
        json.dump(matrix,f,indent=2)
        #for row in matrix:
            #json.dump(row,f)
            #f.write("\n")

def testOutput():
    #for tweet in tweets_per_account[0]:
        #print(tweet)
        #print("\n")

    #for words in words_per_account[1]:
        #print(words)
        #print("\n")
    #print(all_words)

    for tweet in common_words_per_account[0]:
        for word in tweet:
            print(word)
        print("\n"+("-"*25))

if __name__ == '__main__':
    readAccounts()
    #readInFinalWords()
    #processFinalWords()
    process()
    writeMatrixData()
    #writeCommonTermsPerAccount()
    #writeFinalWordsToFile()
```

Listing 4: Python code used for word processing in Q2

The final 2D list of data *matrix* was then exported to JSON for use in another file to build the physical matrix. Listing 5 below shows an extremely small beginning snippet of the JSON file, highlighting the first 25 most common terms across all tweets and their frequency in tweets posted by @*BarackObama*.


```json
[
  {
    "sorry": 0,
    "election": 44,
    "covid19": 3,
    "app": 0,
    "christmas": 0,
    "song": 0,
    "details": 0,
```

```
    "birthday": 0,
    "phone": 1,
    "store": 0,
    "trump": 0,
    "address": 1,
    "news": 1,
    "location": 0,
    "nba": 4,
    "name": 1,
    "collection": 1,
    "games": 0,
    "email": 0,
    "december": 1,
    "via": 0,
    "health": 13,
    "care": 17,
    "album": 0,
    "order": 0,
```

Listing 5: First 25 terms for @BarackObama in MatrixData.json

Listing 6 below shows the code used to draw and export the final account-term matrix. The matrix data from *q2.py* and the list of accounts was read in and stored within the file. To assist with creating the matrix, the matrix data in particular was appended to a new list with each ”row” of data being attached to the account name it belonged to (which was easy considering the row of data and it’s associated account shared the same index value in their respective lists). With this list structure, the list could easily be passed into *pd.DataFrame().T* to create a transposed data frame to represent the account-term matrix.

While the final matrix is extremely large and can’t be fully represented in this report, printing the table in my IDE shows an extremely condensed, more easily readable version of the table, shown in Listing 7.

```
import json
import re
from collections import Counter
from operator import itemgetter
from progress.bar import IncrementalBar
import itertools
import array
import pandas as pd
import csv

matrix = []
final_matrix = {}
accounts = []

def process():
```

```
    bar = IncrementalBar('Processing', max=len(matrix))
    for index,user in enumerate(matrix):
        final_matrix[accounts[index]] = {word:key for word,key in user.items()}
        bar.next()
    bar.finish()

    table = pd.DataFrame(final_matrix).T
    print(table)

    #with open('matrix.txt',"w") as f:
        #f.write(table.to_string())

def readInData():
    global matrix
    with open("MatrixData.json","r") as f:
        data=json.load(f)

    for item in data:
        matrix.append(item)

    with open("100Accounts.txt",'r') as a:
        for line in a:
            line = line.rstrip('\n')
            accounts.append(line)

if __name__ == '__main__':
    readInData()
    process()
```

Listing 6: Python code used to draw the account-term matrix after processing

```
                   sorry  election  covid19  ...  present  finish  blast
BarackObama            0        44        3  ...        0       0      0
justinbieber           0         0        0  ...        0       0      0
katyperry              2         4        0  ...        1       0      0
rihanna                0         0        1  ...        0       0      0
realDonaldTrump        0         5        0  ...        0       0      0
...                  ...       ...      ...  ...      ...     ...    ...
Respawn                0         0        0  ...        1       1      0
Charmin                4         0        0  ...        0       0      1
marshmellomusic        0         0        0  ...        2       1      0
MerriamWebster         0         1        0  ...        0       0      0
BaskinRobbins         67         0        0  ...        0       0      1

[100 rows x 1000 columns]
```

Listing 7: Condensed account-term matrix captured in output

Q3

Listing 8 below shows all the code involved in all forms of clustering and drawing graphs; this code is used for answering the remaining questions. The large majority of this code pulled from the example code with some minor changes for reading in data and output.

My resulting account-term matrix was read-in through *readFile()*. Within this function, the matrix was stripped apart to get the screen names, the 1000 terms, and a 2-dimensional array of data containing the frequency counts for each word in each account. These three lists were returned and stored to be used for clustering algorithms.

```
import plotly as py
import plotly.figure_factory as FF
import json
from operator import itemgetter
from math import sqrt
import random
from PIL import Image, ImageDraw
import numpy as np
import matplotlib.pyplot as plt

def process():
    users, words, counts = readFile()    # rownames,colnames,data

    clust = hcluster(counts)

    printclust(clust, labels=users)

    rdata = rotatematrix(counts)
    rclust = hcluster(rdata)

    arr = np.array(rdata)

    fig = FF.create_dendrogram(arr, orientation='left',labels=words)
    fig.update_layout(width=800, height=11000)
    fig.show()
    drawdendrogram(rclust,labels=words, jpeg='clusters.jpg')

    print("For k=5")
    k1=kcluster(counts, k=5)
    writeToFile(k1,users,5)

    print("For k=10")
    k2=kcluster(counts, k=10)
    writeToFile(k2,users,10)

    print("For k=20")
```

```
    k3=kcluster(counts, k=20)
    writeToFile(k3,users,20)

    cords = scaledown(counts)
    draw2d(cords,users, jpeg='mds2d.jpg')
    draw2dAgain(cords,users)


class bicluster:

    def __init__(
        self,
        vec,
        left=None,
        right=None,
        distance=0.0,
        id=None,
        ):
        self.left = left
        self.right = right
        self.vec = vec
        self.id = id
        self.distance = distance

def pearson(v1, v2):
  # Simple sums
    sum1 = sum(v1)
    sum2 = sum(v2)

  # Sums of the squares
    sum1Sq = sum([pow(v, 2) for v in v1])
    sum2Sq = sum([pow(v, 2) for v in v2])

  # Sum of the products
    pSum = sum([v1[i] * v2[i] for i in range(len(v1))])

  # Calculate r (Pearson score)
    num = pSum - sum1 * sum2 / len(v1)
    den = sqrt((sum1Sq - pow(sum1, 2) / len(v1)) * (sum2Sq - pow(sum2, 2)
                / len(v1)))
    if den == 0:
        return 0

    return 1.0 - num / den

#Prints ASCII clusters
```

```
def hcluster(rows, distance=pearson):
    distances = {}
    currentclustid = -1

  # Clusters are initially just the rows
    clust = [bicluster(rows[i], id=i) for i in range(len(rows))]

    while len(clust) > 1:
        lowestpair = (0, 1)
        closest = distance(clust[0].vec, clust[1].vec)

    # loop through every pair looking for the smallest distance
        for i in range(len(clust)):
            for j in range(i + 1, len(clust)):
        # distances is the cache of distance calculations
                if (clust[i].id, clust[j].id) not in distances:
                    distances[(clust[i].id, clust[j].id)] = \
                        distance(clust[i].vec, clust[j].vec)

                d = distances[(clust[i].id, clust[j].id)]

                if d < closest:
                    closest = d
                    lowestpair = (i, j)

    # calculate the average of the two clusters
        mergevec = [(clust[lowestpair[0]].vec[i] + clust[lowestpair[1]].vec[i])
                    / 2.0 for i in range(len(clust[0].vec))]

    # create the new cluster
        newcluster = bicluster(mergevec, left=clust[lowestpair[0]],
                               right=clust[lowestpair[1]], distance=closest,
                               id=currentclustid)

    # cluster ids that weren't in the original set are negative
        currentclustid -= 1
        del clust[lowestpair[1]]
        del clust[lowestpair[0]]
        clust.append(newcluster)

    return clust[0]

def readFile():
    with open ('matrix.txt','r') as f:
        lines = [line for line in f]
```

```


  # First line is the column titles
    #colnames = lines[0].strip().split('\t')[1:]
    colnames = lines[0].strip().split()
    rownames = []
    data = []
    for line in lines[1:]:
        p = line.strip().split()
    # First column in each row is the rowname
        rownames.append(p[0])
    # The data for this row is the remainder of the row
        data.append([float(x) for x in p[1:]])
    return (rownames, colnames, data)

def rotatematrix(data):
    newdata = []
    for i in range(len(data[0])):
        newrow = [data[j][i] for j in range(len(data))]
        newdata.append(newrow)
    return newdata

def printclust(clust, labels=None, n=0):
    with open("ASCII_Cluster.txt","a") as f:

  # indent to make a hierarchy layout
        for i in range(n):
            print(' ')
            f.write(' ')
        if clust.id < 0:
        # negative id means that this is branch
            print ('-')
            f.write('-'+'\n')
        else:
        # positive id means that this is an endpoint
            if labels == None:
                print(clust.id)
                f.write(clust.id+'\n')
            else:
                print(labels[clust.id])
                f.write(labels[clust.id]+'\n')

        # now print the right and left branches
        if clust.left != None:
            printclust(clust.left, labels=labels, n=n + 1)
        if clust.right != None:
            printclust(clust.right, labels=labels, n=n + 1)
```

```

def getheight(clust):
  # Is this an endpoint? Then the height is just 1
    if clust.left == None and clust.right == None:
        return 1

  # Otherwise the height is the same of the heights of
  # each branch
    return getheight(clust.left) + getheight(clust.right)

def getdepth(clust):
  # The distance of an endpoint is 0.0
    if clust.left == None and clust.right == None:
        return 0

  # The distance of a branch is the greater of its two sides
  # plus its own distance
    return max(getdepth(clust.left), getdepth(clust.right)) + clust.distance

def drawdendrogram(clust, labels, jpeg='clusters.jpg'):
  # height and width
    h = getheight(clust) * 20
    w = 1200
    depth = getdepth(clust)

  # width is fixed, so scale distances accordingly
    scaling = float(w - 150) / depth

  # Create a new image with a white background
    img = Image.new('RGB', (w, h), (255, 255, 255))
    draw = ImageDraw.Draw(img)

    draw.line((0, h / 2, 10, h / 2), fill=(255, 0, 0))

  # Draw the first node
    drawnode(
        draw,
        clust,
        10,
        h / 2,
        scaling,
        labels,
        )
    img.save(jpeg, 'JPEG')

def drawnode(
```

```
    draw,
    clust,
    x,
    y,
    scaling,
    labels,):
    if clust.id < 0:
        h1 = getheight(clust.left) * 20
        h2 = getheight(clust.right) * 20
        top = y - (h1 + h2) / 2
        bottom = y + (h1 + h2) / 2
    # Line length
        ll = clust.distance * scaling
    # Vertical line from this cluster to children
        draw.line((x, top + h1 / 2, x, bottom - h2 / 2), fill=(255, 0,
  0))

    # Horizontal line to left item
        draw.line((x, top + h1 / 2, x + ll, top + h1 / 2), fill=(255,
  0, 0))

    # Horizontal line to right item
        draw.line((x, bottom - h2 / 2, x + ll, bottom - h2 / 2), fill
  =(255, 0,
                  0))

    # Call the function to draw the left and right nodes
        drawnode(
            draw,
            clust.left,
            x + ll,
            top + h1 / 2,
            scaling,
            labels,
            )
        drawnode(
            draw,
            clust.right,
            x + ll,
            bottom - h2 / 2,
            scaling,
            labels,
            )
    else:
    # If this is an endpoint, draw the item label
        draw.text((x + 5, y - 7), labels[clust.id], (0, 0, 0))

```

```
def kcluster(rows, distance=pearson, k=4):
  # Determine the minimum and maximum values for each point
    ranges = [(min([row[i] for row in rows]), max([row[i] for row in rows]))
              for i in range(len(rows[0]))]

  # Create k randomly placed centroids
    clusters = [[random.random() * (ranges[i][1] - ranges[i][0]) + ranges[i][0]
                for i in range(len(rows[0]))] for j in range(k)]

    lastmatches = None
    for t in range(100):
        print ('Iteration %d' % t)
        bestmatches = [[] for i in range(k)]

    # Find which centroid is the closest for each row
        for j in range(len(rows)):
            row = rows[j]
            bestmatch = 0
            for i in range(k):
                d = distance(clusters[i], row)
                if d < distance(clusters[bestmatch], row):
                    bestmatch = i
            bestmatches[bestmatch].append(j)

    # If the results are the same as last time, this is complete
        if bestmatches == lastmatches:
            break
        lastmatches = bestmatches

    # Move the centroids to the average of their members
        for i in range(k):
            avgs = [0.0] * len(rows[0])
            if len(bestmatches[i]) > 0:
                for rowid in bestmatches[i]:
                    for m in range(len(rows[rowid])):
                        avgs[m] += rows[rowid][m]
                for j in range(len(avgs)):
                    avgs[j] /= len(bestmatches[i])
                clusters[i] = avgs

    return bestmatches

def scaledown(data, distance=pearson, rate=0.01):
    n = len(data)
```

```
# The real distances between every pair of items
  realdist = [[distance(data[i], data[j]) for j in range(n)] for i in
              range(0, n)]

# Randomly initialize the starting points of the locations in 2D
  loc = [[random.random(), random.random()] for i in range(n)]
  fakedist = [[0.0 for j in range(n)] for i in range(n)]

  lasterror = None
  for m in range(0, 1000):
  # Find projected distances
      for i in range(n):
          for j in range(n):
              fakedist[i][j] = sqrt(sum([pow(loc[i][x] - loc[j][x], 2)
                                    for x in range(len(loc[i]))]))

  # Move points
      grad = [[0.0, 0.0] for i in range(n)]

      totalerror = 0
      for k in range(n):
          for j in range(n):
              if j == k:
                  continue
      # The error is percent difference between the distances
              errorterm = (fakedist[j][k] - realdist[j][k]) / realdist[j][k]

      # Each point needs to be moved away from or towards the other
      # point in proportion to how much error it has
              grad[k][0] += (loc[k][0] - loc[j][0]) / fakedist[j][k] \
                  * errorterm
              grad[k][1] += (loc[k][1] - loc[j][1]) / fakedist[j][k] \
                  * errorterm

      # Keep track of the total error
              totalerror += abs(errorterm)
      print (totalerror)

  # If the answer got worse by moving the points, we are done
      if lasterror and lasterror < totalerror:
          break
      lasterror = totalerror
```

```
    # Move each of the points by the learning rate times the gradient
        for k in range(n):
            loc[k][0] -= rate * grad[k][0]
            loc[k][1] -= rate * grad[k][1]

    return loc

def draw2d(data, labels, jpeg='mds2d.jpg'):
    img = Image.new('RGB', (2000, 2000), (255, 255, 255))
    draw = ImageDraw.Draw(img)
    for i in range(len(data)):
        x = (data[i][0] + 0.5) * 1000
        y = (data[i][1] + 0.5) * 1000
        draw.text((x, y), labels[i], (0, 0, 0))
    img.save(jpeg, 'JPEG')

def draw2dAgain(data,labels):
    x_axis = []
    y_axis = []

    for i in range(len(data)):
        x = (data[i][0] + 0.5) * 1000
        y = (data[i][1] + 0.5) * 1000
        x_axis.append(x)
        y_axis.append(y)
        plt.text(x*(1+0.01),y*(1+0.01),labels[i],fontsize=10)

    plt.scatter(x_axis,y_axis)
    plt.show()


def writeToFile(klist, users,k):
    with open("k{}_results.txt".format(k),'w') as f:
        f.write("Iterations at k=("+str(k)+"):  "+str(len(klist))+"\n\n")
        for cluster in klist:
            for u in cluster:
                f.write(users[u]+"\n")
            f.write("\n")


if __name__ == '__main__':
    process()
```

Listing 8: Python code to handle all clustering and dendrograms

The method *hcluster()* takes the data array and creates the hierarchical clusters of the accounts. This list of clusters is passed to *printclust()* to print an ASCII dendrogram of the clusters. Figure

1 below shows a small snapshot of the ASCII dendrogram, showcasing it's hierarchical structure. *drawdendrogram()* was then called to create a JPEG image of a new set of clusters created by calling *hcluster()* again after rotating the data array with *rotatematrix()* (rotating the array was only needed to print a vertical dendrogram like the ASCII dendrogram). The resulting dendrogram is shown in Figure 2; the image itself was both massive and extremely long, so I took cropped snapshot of the very top of the dendrogram to showcase.

ASCII_Cluster - Notepad

File Edit Format View Help

```
   MariahCarey
   NBA
  -
   YouTube
    hulu
     Sony
      BET
      BETMusic
     -
    -
   -
  -
 -
-
  NASA
  elonmusk
 -
   NatGeo
   WHO
  -
    shakira
    ricky_martin
   -
    pitbull
     adamlevine
      selenagomez
        KendallJenner
         RoosterTeeth
          Ninja
           KimKardashian
           KylieJenner
          -
         -
        -
       -
        michaelb4jordan
          NathanFillion
           KevinHart4real
           TheRock
          -
         -
           TheEllenShow
            PegboardNerds
             MrPeanut
              PapaJohns
               tacobell
               McDonalds
              -
```

Ln 94, Col 9 | 50% | Windows (CRLF) | UTF-8

Figure 1: Snippet of ASCII dendrogram

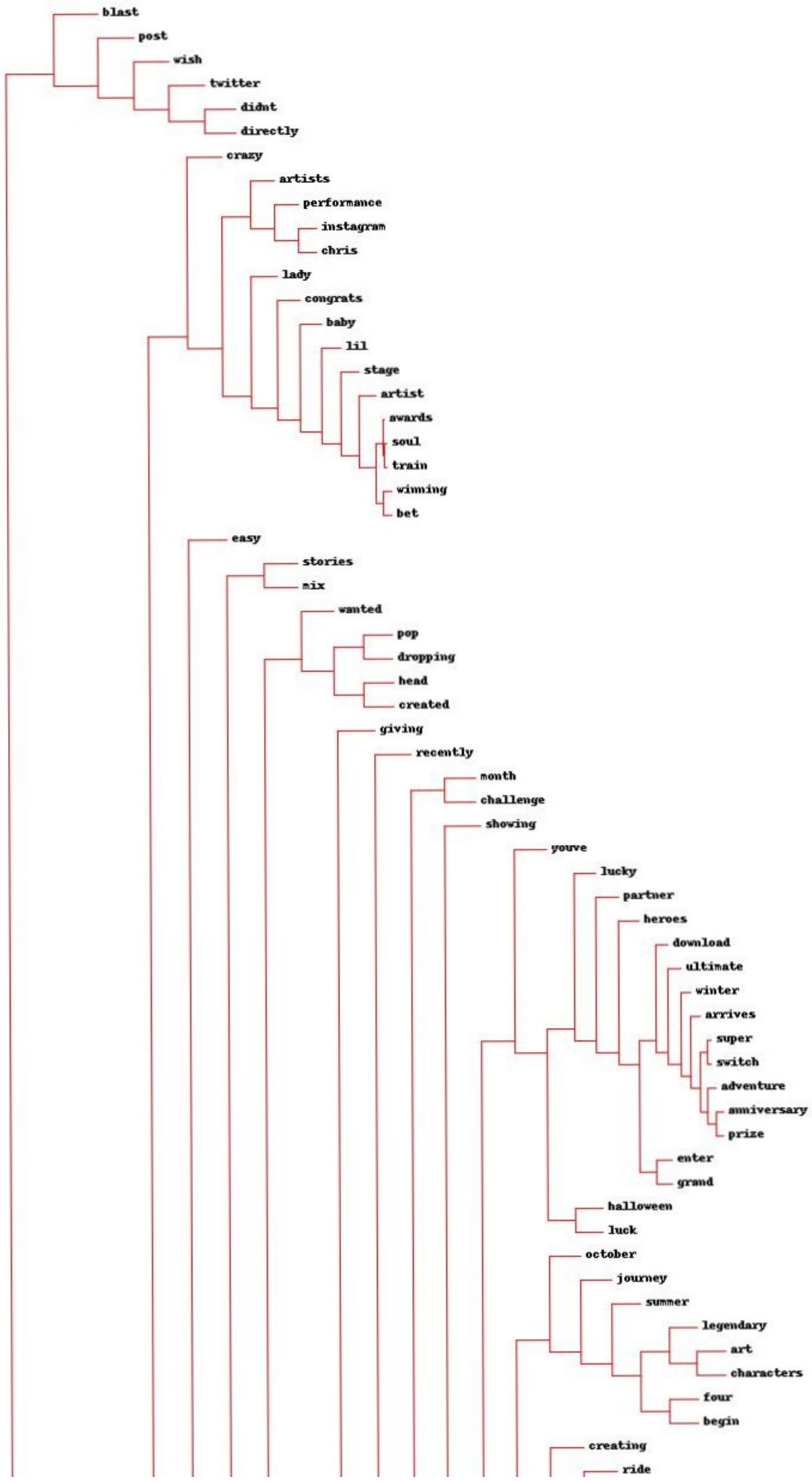

Figure 2: Snippet of dendrogram

Q4

Clustering the accounts using k-means was done through the *kcluster()* method which takes the data array and a *k* value as parameters. I called the method three times for k=5, k=10, and k=20, and wrote the resulting account clusters to their own files.

Listing 9 below shows the number of iterations taken for each k-value. At k=5, five iterations were needed to cluster the accounts. At k=10, thirteen iterations were needed to cluster the accounts. At k=20, only four iterations were needed to cluster the accounts.

```
For k=5
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5

For k=10
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13

For k=20
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

Listing 9: Iterations for each k-value

Listing 10 shows the different account clusters at k=5:

- The first cluster is primarily fast food accounts, but also features a sports network and voice actor account.

- The second cluster is not too coherent, as it features a mix of political figures, celebrities, musicians, video game companies, and miscellaneous businesses.
- The third cluster is very similar to the second, featuring a mix of celebrities, musicians, streaming services, influencers, and sports networks.
- The fourth cluster can't really be characterized as it only contains 5 members who are all different from one another.
- The fifth cluster predominantly features news websites, the remaining political figures, and parody accounts.

```
Iterations at k=(5):  5

matthewmercer
NFL
Wendys
dominos
McDonalds
BurgerKing
PapaJohns
pizzahut
BaskinRobbins

BarackObama
ladygaga
selenagomez
ddlovato
shakira
Oprah
NASA
elonmusk
HillaryClinton
pitbull
NatGeo
ricky_martin
WHO
Microsoft
NintendoAmerica
EA
Blizzard_Ent
bethesda
WuTangClan
littlecaesars
Bungie
Treyarch
Respawn
Charmin
```

```

justinbieber
katyperry
rihanna
ArianaGrande
TheEllenShow
YouTube
KimKardashian
KingJames
MileyCyrus
JLo
BeingSalmanKhan
KevinHart4real
wizkhalifa
KylieJenner
KendallJenner
aliciakeys
coldplay
MariahCarey
NICKIMINAJ
davidguetta
JERICHO
SavinTheBees
Ninja
PostMalone
hulu
netflix
adamlevine
MLB
TheRock
WNBA
Sony
NintendoUK
FIFAcom
michaelb4jordan
BET
BETMusic
NathanFillion
TroyBakerVA
WayneBrady
tacobell
MrPeanut
RoosterTeeth
Monstercat
PegboardNerds
marshmellomusic
MerriamWebster
```

```

espn
Google
NBA
Windows
PaulMcCartney

realDonaldTrump
CNN
jimmyfallon
nytimes
TheDailyShow
FoxNews
NHL
RealRonHoward
BigMikeJ73
AOC
ToddHaberkorn
Crunchyroll
StephenAtHome
KenJennings
TheTweetOfGod
TheOnion
```

Listing 10: Clustering at k=5

Listing 11 shows the different account clusters at k=10:

- The first cluster seems to be related to current politics.
- The second cluster has a good mix of celebrities, musicians, and game companies; nothing to specific.
- The third cluster is similar to the last, being a list of celebrities, musicians, and game companies.
- The fourth cluster is primarily actors, musicians, and game companies, however, these actors and musicians often interact together such as Kevin Hart and Dwayne Johnson, and Justin Bieber and Selena.
- The NHL is it's own cluster for some reason, when ideally it would be grouped with the other sports networks.
- The sixth cluster is probably the most focused in this set. The cluster is largely comprised of accounts related to music, streaming, and games. There are some unrelated famous figures mixed in such as LeBron James, Michael B. Jordan (the actor), and Wayne Brady.
- The seventh cluster is mostly fast food restaurants, but the other half of the accounts are unrelated to either restaurants nor each other.

- The last three clusters aren't at all coherent and are quite small. So it's difficult to determine how they could be clustered together.

```
Iterations at k=(10):  13

realDonaldTrump
CNN
nytimes
NatGeo
TheDailyShow
FoxNews
AOC
ToddHaberkorn
TheTweetOfGod
TheOnion

BarackObama
ladygaga
TheEllenShow
YouTube
ddlovato
Oprah
NASA
elonmusk
HillaryClinton
bethesda
NathanFillion
tacobell
StephenAtHome
Treyarch
marshmellomusic

KimKardashian
shakira
KylieJenner
KendallJenner
pitbull
ricky_martin
WHO
Microsoft
NHL
PaulMcCartney
EA
Blizzard_Ent
WuTangClan
littlecaesars
MrPeanut
KenJennings
```

```
Bungie
Respawn

justinbieber
selenagomez
BeingSalmanKhan
KevinHart4real
TheRock
NintendoAmerica
NintendoUK
RoosterTeeth

NFL

katyperry
rihanna
ArianaGrande
KingJames
MileyCyrus
JLo
wizkhalifa
aliciakeys
coldplay
MariahCarey
NICKIMINAJ
davidguetta
JERICHO
SavinTheBees
Ninja
PostMalone
hulu
netflix
adamlevine
Sony
michaelb4jordan
BET
BETMusic
TroyBakerVA
WayneBrady
Monstercat
PegboardNerds

matthewmercer
Windows
RealRonHoward
Wendys
dominos
```

```
McDonalds
PapaJohns
Charmin
MerriamWebster
BaskinRobbins

WNBA
BigMikeJ73
BurgerKing
pizzahut

jimmyfallon
espn
NBA
Crunchyroll

Google
MLB
FIFAcom
```

Listing 11: Clustering at k=10

Listing 12 shows the different account clusters at k=20. Most the of the clusters for k= 20 are quite small, with some only containing a single account. I do want to highlight a few clusters I believe to be quite accurate:

- The very first cluster with @KevinHart4Real and @TheRock makes sense, as these two are both good friends and have appeared together in many popular movies.
- The seventh cluster includes all the athletes alongside the @espn sports network. The others in that cluster share very similar Twitter personalities.
- The thirteenth cluster houses all accounts with political affiliation and news outlets (outside of Little Caesars Pizza, unsure about that one). These accounts may have tweeted heavily in regards to the most recent election.
- The sixteenth cluster is small, but groups @NASA and @elonmusk together, which makes sense given their involvement in space programs.
- The final cluster is the largest, but also almost entirely made up of popular musicians and music streaming platforms; the only oddballs are @WHO (World Health Organization) and @NatGeo

```
Iterations at k=(20):  4


KevinHart4real
```

```
TheRock

Google
NHL
NintendoUK
dominos

NathanFillion
TheTweetOfGod
Treyarch

NFL

Microsoft

MariahCarey
JERICHO
Ninja
bethesda

rihanna
KingJames
espn
NICKIMINAJ
SavinTheBees
BigMikeJ73
michaelb4jordan

wizkhalifa
PapaJohns
MrPeanut
StephenAtHome

ArianaGrande
shakira
BeingSalmanKhan
matthewmercer
netflix
WNBA
Sony
WuTangClan
BaskinRobbins

TheEllenShow
KendallJenner
MLB
FIFAcom
```

```
EA
BETMusic
Bungie
Charmin

pitbull
hulu
NintendoAmerica
PaulMcCartney
RealRonHoward
Respawn

NBA

BarackObama
realDonaldTrump
ladygaga
CNN
nytimes
Oprah
HillaryClinton
TheDailyShow
FoxNews
AOC
littlecaesars
TheOnion

jimmyfallon
BET
ToddHaberkorn
Crunchyroll
tacobell

KimKardashian
KylieJenner
Wendys
McDonalds
BurgerKing
pizzahut
RoosterTeeth

NASA
elonmusk
MerriamWebster

Windows
Blizzard_Ent
```

```
TroyBakerVA
WayneBrady
PegboardNerds
KenJennings

justinbieber
katyperry
YouTube
selenagomez
ddlovato
MileyCyrus
JLo
aliciakeys
NatGeo
coldplay
davidguetta
ricky_martin
WHO
PostMalone
adamlevine
Monstercat
marshmellomusic
```

Listing 12: Clustering at k=20

Comparing the different k-values and their results on the data, it appears to me that k = 20 is the most accurate value for clustering my data. While each value shared a number of inaccuracies in it's clustering, the clusters at k = 20 seem to make the most amount of sense. This comes as a surprise, as I was initially expecting k = 10 to provide the most accurate clusters.

Q5

MDS was handled through both the *scaledown()* and *draw2d()* methods. *scaledown()* took in the 2D array of data as an arguments and returned the coordinates of each account. These coordinates allowed the data to be clustered and plotted in a 3-dimensional format. Listing 13 shows a list of the total percent differences per iteration. Counting the output, a total of 92 iterations was needed to move each point the the best possible positions.

This graph was plotted by passing the coordinates into *draw2d()* alongside the list of accounts; this method used the imaging library *Pillow* to plot and export the image. The final MDS image is shown below in Figure 3. Unfortunately, due to it's extreme size, it had to be scaled down to capture the full, 3-dimensional effect.

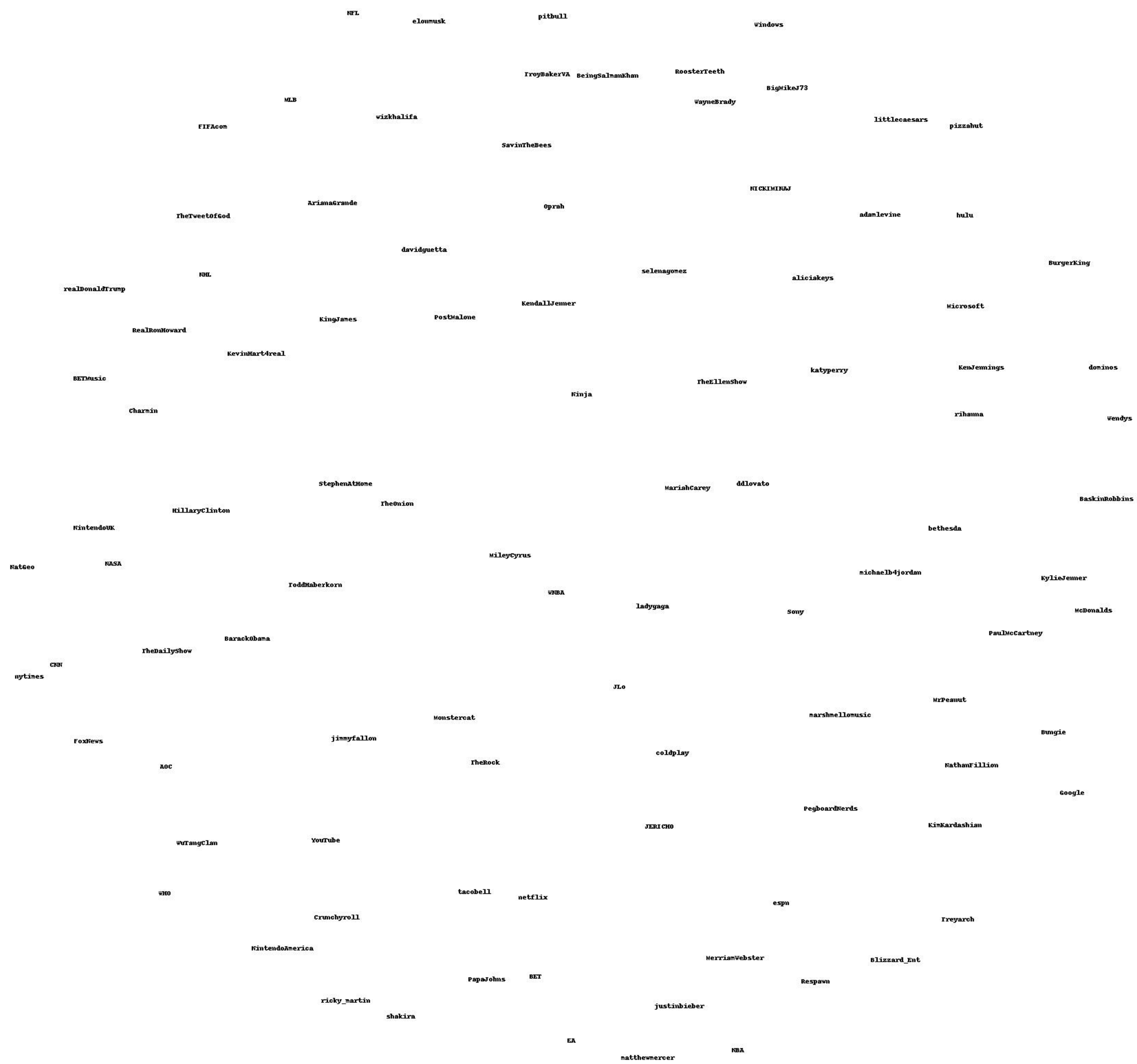

Figure 3: MDS

```
4292.093979274119
3468.6963378241735
3443.1605133694075
3433.8771891736046
3428.3435921231517
3424.1729984429485
3421.120766401213
3418.7469839225996
3416.4118497193167
3414.4440844275505
```

```
3412.440863751611
3410.6465210733254
3409.0135600255144
3407.828801597129
3406.614686573917
3405.348619533868
3403.879972210023
3402.4599352017785
3400.6049413304436
3398.4835025635516
3396.4503913590497
3394.498279149446
3392.6385673344253
3390.574933345304
3388.2408982400316
3385.945677397648
3383.644519964651
3381.3471770664746
3379.071768466391
3376.9061201468085
3374.86572041902
3372.7470693075834
3370.679695018536
3368.8020144909347
3367.078404372282
3365.4284726916517
3363.5180227427154
3361.5044352044524
3359.567836902659
3357.6994851742597
3356.075201494575
3354.6152534564762
3353.3437071308554
3352.335510480809
3351.321018381074
3350.1888786790323
3348.985883309517
3347.780406455964
3346.5534605142593
3345.199031162315
3343.751169397896
3342.243964528533
3340.5999192427143
3339.0387078681683
3337.450064806221
3335.923911769075
3334.5058497032032
```

```
3333.1150908016352
3331.899056124837
3330.6460139440615
3329.390198500818
3328.2085720215514
3326.942245522004
3325.6903331491635
3324.731009673131
3323.9414807388216
3323.252399237465
3322.621871481046
3321.966053962298
3321.3863407367453
3320.7252626481722
3320.1258695526835
3319.5913971475425
3319.042673119391
3318.3179305943013
3317.530619079285
3316.6666483553036
3315.6227629071536
3314.6336580402417
3313.929465632325
3313.255518125941
3312.461244955732
3311.738311092058
3310.991010500956
3310.235490785789
3309.330568837568
3308.2826330791786
3307.2518796395166
3306.6054579567126
3305.799861116069
3305.253415207178
3305.255269419066
```

Listing 13: Output from scaledown()

Q7 (Extra)

Figure 4 below shows an alternate version of the dendrogram shown in Figure 2 using the same data. This version of the dendrogram was generated using the *plotly.figure_facory* library and the *create_dendrogram()* method. In order to get a clear view of the words, the image height had to be scaled up considerably to the point where it would be impossible to pass into this report. I trimmed and scaled it to best capture the overall shape from a portion of the final dendrogram.

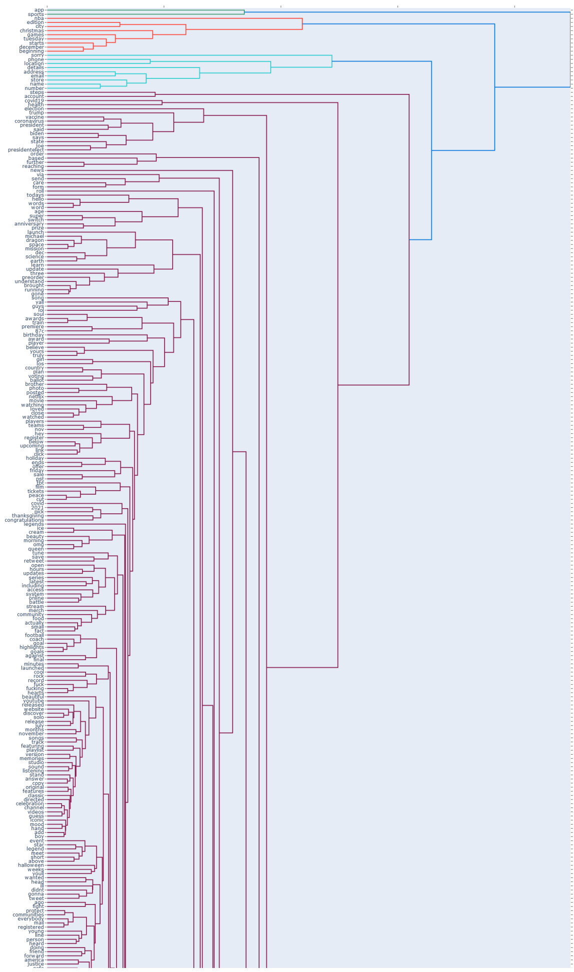

Figure 4: Snippet of dendrogram plotted using Plotly

Q8 (Extra)

Using the same coordinate gathering method for each account from Q5, I created a scatter plot mimicking the generated MDS figure. This was done using the *matplotlib.pyplot* library within the *draw2dAgain()* method. All the x and y-axis values were captured and passed to *plt.scatter()*, with *plt.text()* to attach the account labels to each point. Due to running *scaledown()* again for this graph, the points are not in the same position as the original MDS figure.

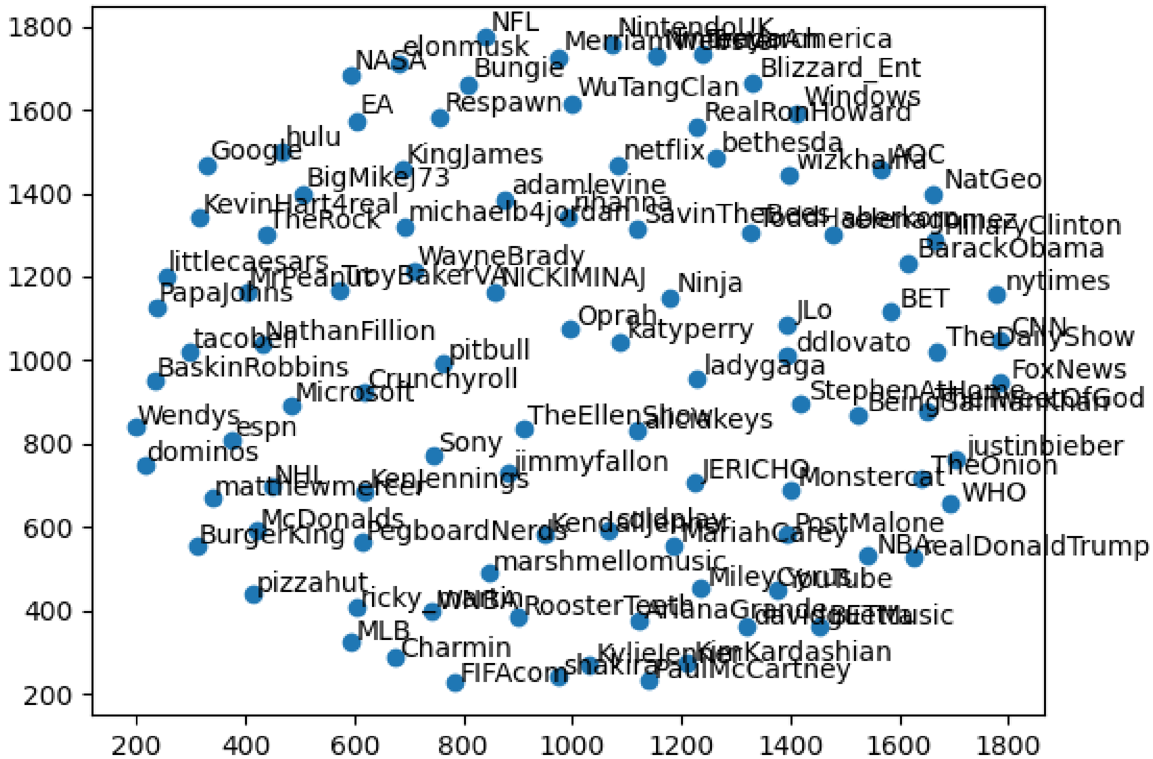

Figure 5: MDS plotted using PyPlot