

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.

```
1 import tweepy
2 from tweepy.streaming import StreamListener
3 from tweepy import OAuthHandler
4 from tweepy import Stream
5 import json
6
7 # Keys omitted
8 consumer_key="***"
9 consumer_secret="***"
10 access_token="***"
11 access_secret="***"
12
13 # Handles authorization with Twitter
14 auth = OAuthHandler(consumer_key, consumer_secret)
15 auth.set_access_token(access_token, access_secret)
16 api = tweepy.API(auth, wait_on_rate_limit=True,
17                  wait_on_rate_limit_notify=True)
17 #api = tweepy.API(auth)
18
19 accounts=[]
20 tweets_per_account = []
21
```

```
22
23 def retrieveTweets():
24     count=1
25     for user in accounts:
26         print("Processing Account #" +str(count)+": " +user)
27         tweets = []
28         statuses = api.user_timeline(screen_name=user, count=200,
tweet_mode="extended")
29         for status in statuses:
30             #tweet_dict = {"USER":user, "ID":status.id, "TEXT":status.
full_text}
31             tweet_dict = {"ID":status.id, "TEXT":status.full_text}
32             tweets.append(tweet_dict)
33
34             writeTweetsToFile(user, tweets)
35             #writeTweetsToJSON(user, tweets)
36             #account_dict = {"USER":user, "TWEETS":tweets}
37             account_dict = {user: {tweets}}
38             tweets_per_account.append(account_dict)
39             print("Done" +'\n')
40             count+=1
41
42
43 def writeTweetsToFile(account, tweets):
44     path = r"C:\Users\Brenden\Desktop\ODU\Fall 2020\CS432\HW_8\hw8-
cluster-blewis1014\account_tweets\tweets_{}.txt".format(account)
45     fileToWrite = open(path, "a", encoding="utf-8")
46     for tweet in tweets:
47         fileToWrite.write(str(tweet["ID"]) + "\n")
48         fileToWrite.write(tweet["TEXT"] + "\n\n")
49     fileToWrite.close()
50
51 def writeTweetsToJSON():
52     with open("H9_tweets.json", "w") as f:
53         json.dump(tweets_per_account, f, indent=2)
54
55 def readAccounts():
56     with open("100Accounts.txt", 'r') as f:
57         for line in f:
58             line = line.rstrip('\n')
59             accounts.append(line)
60
61 def collectAccounts():
62     user_ID = ""
63     #user_IDs = ["", "", "",
64     #
65     #
66     #
67     #
68     #
69     #
70     #
71     #
72     #
73     #
74     #
75     #
76     #
77     #
78     #
79     #
80     #
81     #
82     #
83     #
84     #
85     #
86     #
87     #
88     #
89     #
90     #
91     #
92     #
93     #
94     #
95     #
96     #
97     #
98     #
99     #
100    #
101    #
102    #
103    #
104    #
105    #
106    #
107    #
108    #
109    #
110    #
111    #
112    #
113    #
114    #
115    #
116    #
117    #
118    #
119    #
120    #
121    #
122    #
123    #
124    #
125    #
126    #
127    #
128    #
129    #
130    #
131    #
132    #
133    #
134    #
135    #
136    #
137    #
138    #
139    #
140    #
141    #
142    #
143    #
144    #
145    #
146    #
147    #
148    #
149    #
150    #
151    #
152    #
153    #
154    #
155    #
156    #
157    #
158    #
159    #
160    #
161    #
162    #
163    #
164    #
165    #
166    #
167    #
168    #
169    #
170    #
171    #
172    #
173    #
174    #
175    #
176    #
177    #
178    #
179    #
180    #
181    #
182    #
183    #
184    #
185    #
186    #
187    #
188    #
189    #
190    #
191    #
192    #
193    #
194    #
195    #
196    #
197    #
198    #
199    #
200    #
201    #
202    #
203    #
204    #
205    #
206    #
207    #
208    #
209    #
210    #
211    #
212    #
213    #
214    #
215    #
216    #
217    #
218    #
219    #
220    #
221    #
222    #
223    #
224    #
225    #
226    #
227    #
228    #
229    #
230    #
231    #
232    #
233    #
234    #
235    #
236    #
237    #
238    #
239    #
240    #
241    #
242    #
243    #
244    #
245    #
246    #
247    #
248    #
249    #
250    #
251    #
252    #
253    #
254    #
255    #
256    #
257    #
258    #
259    #
260    #
261    #
262    #
263    #
264    #
265    #
266    #
267    #
268    #
269    #
270    #
271    #
272    #
273    #
274    #
275    #
276    #
277    #
278    #
279    #
280    #
281    #
282    #
283    #
284    #
285    #
286    #
287    #
288    #
289    #
290    #
291    #
292    #
293    #
294    #
295    #
296    #
297    #
298    #
299    #
300    #
301    #
302    #
303    #
304    #
305    #
306    #
307    #
308    #
309    #
310    #
311    #
312    #
313    #
314    #
315    #
316    #
317    #
318    #
319    #
320    #
321    #
322    #
323    #
324    #
325    #
326    #
327    #
328    #
329    #
330    #
331    #
332    #
333    #
334    #
335    #
336    #
337    #
338    #
339    #
340    #
341    #
342    #
343    #
344    #
345    #
346    #
347    #
348    #
349    #
350    #
351    #
352    #
353    #
354    #
355    #
356    #
357    #
358    #
359    #
360    #
361    #
362    #
363    #
364    #
365    #
366    #
367    #
368    #
369    #
370    #
371    #
372    #
373    #
374    #
375    #
376    #
377    #
378    #
379    #
380    #
381    #
382    #
383    #
384    #
385    #
386    #
387    #
388    #
389    #
390    #
391    #
392    #
393    #
394    #
395    #
396    #
397    #
398    #
399    #
400    #
401    #
402    #
403    #
404    #
405    #
406    #
407    #
408    #
409    #
410    #
411    #
412    #
413    #
414    #
415    #
416    #
417    #
418    #
419    #
420    #
421    #
422    #
423    #
424    #
425    #
426    #
427    #
428    #
429    #
430    #
431    #
432    #
433    #
434    #
435    #
436    #
437    #
438    #
439    #
440    #
441    #
442    #
443    #
444    #
445    #
446    #
447    #
448    #
449    #
450    #
451    #
452    #
453    #
454    #
455    #
456    #
457    #
458    #
459    #
460    #
461    #
462    #
463    #
464    #
465    #
466    #
467    #
468    #
469    #
470    #
471    #
472    #
473    #
474    #
475    #
476    #
477    #
478    #
479    #
480    #
481    #
482    #
483    #
484    #
485    #
486    #
487    #
488    #
489    #
490    #
491    #
492    #
493    #
494    #
495    #
496    #
497    #
498    #
499    #
500    #
501    #
502    #
503    #
504    #
505    #
506    #
507    #
508    #
509    #
510    #
511    #
512    #
513    #
514    #
515    #
516    #
517    #
518    #
519    #
520    #
521    #
522    #
523    #
524    #
525    #
526    #
527    #
528    #
529    #
530    #
531    #
532    #
533    #
534    #
535    #
536    #
537    #
538    #
539    #
540    #
541    #
542    #
543    #
544    #
545    #
546    #
547    #
548    #
549    #
550    #
551    #
552    #
553    #
554    #
555    #
556    #
557    #
558    #
559    #
560    #
561    #
562    #
563    #
564    #
565    #
566    #
567    #
568    #
569    #
570    #
571    #
572    #
573    #
574    #
575    #
576    #
577    #
578    #
579    #
580    #
581    #
582    #
583    #
584    #
585    #
586    #
587    #
588    #
589    #
590    #
591    #
592    #
593    #
594    #
595    #
596    #
597    #
598    #
599    #
600    #
601    #
602    #
603    #
604    #
605    #
606    #
607    #
608    #
609    #
610    #
611    #
612    #
613    #
614    #
615    #
616    #
617    #
618    #
619    #
620    #
621    #
622    #
623    #
624    #
625    #
626    #
627    #
628    #
629    #
630    #
631    #
632    #
633    #
634    #
635    #
636    #
637    #
638    #
639    #
640    #
641    #
642    #
643    #
644    #
645    #
646    #
647    #
648    #
649    #
650    #
651    #
652    #
653    #
654    #
655    #
656    #
657    #
658    #
659    #
660    #
661    #
662    #
663    #
664    #
665    #
666    #
667    #
668    #
669    #
670    #
671    #
672    #
673    #
674    #
675    #
676    #
677    #
678    #
679    #
680    #
681    #
682    #
683    #
684    #
685    #
686    #
687    #
688    #
689    #
690    #
691    #
692    #
693    #
694    #
695    #
696    #
697    #
698    #
699    #
700    #
701    #
702    #
703    #
704    #
705    #
706    #
707    #
708    #
709    #
710    #
711    #
712    #
713    #
714    #
715    #
716    #
717    #
718    #
719    #
720    #
721    #
722    #
723    #
724    #
725    #
726    #
727    #
728    #
729    #
730    #
731    #
732    #
733    #
734    #
735    #
736    #
737    #
738    #
739    #
740    #
741    #
742    #
743    #
744    #
745    #
746    #
747    #
748    #
749    #
750    #
751    #
752    #
753    #
754    #
755    #
756    #
757    #
758    #
759    #
760    #
761    #
762    #
763    #
764    #
765    #
766    #
767    #
768    #
769    #
770    #
771    #
772    #
773    #
774    #
775    #
776    #
777    #
778    #
779    #
780    #
781    #
782    #
783    #
784    #
785    #
786    #
787    #
788    #
789    #
790    #
791    #
792    #
793    #
794    #
795    #
796    #
797    #
798    #
799    #
800    #
801    #
802    #
803    #
804    #
805    #
806    #
807    #
808    #
809    #
810    #
811    #
812    #
813    #
814    #
815    #
816    #
817    #
818    #
819    #
820    #
821    #
822    #
823    #
824    #
825    #
826    #
827    #
828    #
829    #
830    #
831    #
832    #
833    #
834    #
835    #
836    #
837    #
838    #
839    #
840    #
841    #
842    #
843    #
844    #
845    #
846    #
847    #
848    #
849    #
850    #
851    #
852    #
853    #
854    #
855    #
856    #
857    #
858    #
859    #
860    #
861    #
862    #
863    #
864    #
865    #
866    #
867    #
868    #
869    #
870    #
871    #
872    #
873    #
874    #
875    #
876    #
877    #
878    #
879    #
880    #
881    #
882    #
883    #
884    #
885    #
886    #
887    #
888    #
889    #
890    #
891    #
892    #
893    #
894    #
895    #
896    #
897    #
898    #
899    #
900    #
901    #
902    #
903    #
904    #
905    #
906    #
907    #
908    #
909    #
910    #
911    #
912    #
913    #
914    #
915    #
916    #
917    #
918    #
919    #
920    #
921    #
922    #
923    #
924    #
925    #
926    #
927    #
928    #
929    #
930    #
931    #
932    #
933    #
934    #
935    #
936    #
937    #
938    #
939    #
940    #
941    #
942    #
943    #
944    #
945    #
946    #
947    #
948    #
949    #
950    #
951    #
952    #
953    #
954    #
955    #
956    #
957    #
958    #
959    #
960    #
961    #
962    #
963    #
964    #
965    #
966    #
967    #
968    #
969    #
970    #
971    #
972    #
973    #
974    #
975    #
976    #
977    #
978    #
979    #
980    #
981    #
982    #
983    #
984    #
985    #
986    #
987    #
988    #
989    #
990    #
991    #
992    #
993    #
994    #
995    #
996    #
997    #
998    #
999    #
1000   #
```

```
66 #         "", "", "",
67 #         "", "", "" ]
68
69 user = api.get_user(user_ID)
70 #users = api.lookup_users(screen_names=user_IDs)
71
72 #for user in users:
73 #    tweet_count = user.statuses_count
74 #    print(user.screen_name+"\n"+"Tweets: "+str(tweet_count))
75 #    print("Followers: "+str(user.followers_count)+"\n")
76
77 tweet_count = user.statuses_count
78 print(user.screen_name+"\n"+"Tweets: "+str(tweet_count))
79 print("Followers: "+str(user.followers_count)+"\n")
80
81 if __name__ == '__main__':
82     #collectAccounts()
83     readAccounts()
84     retrieveTweets()
85     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.

```
1 BarackObama
2 justinbieber
3 katyperry
4 rihanna
5realDonaldTrump
6 ladygaga
7 ArianaGrande
8 TheEllenShow
9 YouTube
10 KimKardashian
11 selenagomez
12 CNN
13 ddlovato
14 shakira
15 jimmyfallon
16 KingJames
17 nytimes
18 MileyCyrus
19 JLo
```

```
20 Oprah
21 NASA
22 BeingSalmanKhan
23 elonmusk
24 KevinHart4real
25 wizkhalifa
26 KylieJenner
27 espn
28 KendallJenner
29 aliciakeys
30 HillaryClinton
31 pitbull
32 NatGeo
33 coldplay
34 Google
35 MariahCarey
36 NICKIMINAJ
37 davidguetta
38 ricky_martin
39 JERICHO
40 SavinTheBees
41 NBA
42 WHO
43 Ninja
44 PostMalone
45 TheDailyShow
46 FoxNews
47 matthewmercercer
48 hulu
49 netflix
50 adamlevine
51 NFL
52 MLB
53 TheRock
54 WNBA
55 Sony
56 Microsoft
57 Windows
58 NHL
59 NintendoAmerica
60 NintendoUK
61 PaulMcCartney
62 FIFAcorn
63 EA
64 Blizzard_Ent
65 RealRonHoward
66 BigMikeJ73
```

```
67 michaelb4jordan
68 BET
69 BETMusic
70 bethesda
71 WuTangClan
72 AOC
73 NathanFillion
74 ToddHaberKorn
75 TroyBakerVA
76 WayneBrady
77 Crunchyroll
78 Wendys
79 dominos
80 tacobell
81 McDonalds
82 BurgerKing
83 PapaJohns
84 littlecaesars
85 pizzahut
86 MrPeanut
87 RoosterTeeth
88 Monstercat
89 PegboardNerds
90 StephenAtHome
91 KenJennings
92 TheTweetOfGod
93 TheOnion
94 Bungee
95 Treyarch
96 Respawn
97 Charmin
98 marshmellomusic
99 MerriamWebster
100 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.

```
1 1336679633769680898
```

```
2 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
3
4 1336370071917240321
5 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
6
7 1335954913344565258
8 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/2Ob57ligh3 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*.

```
1 import json
2 import re
3 import nltk
4 from nltk.corpus import stopwords
5 from collections import Counter
6 from operator import itemgetter
7 from progress.bar import IncrementalBar
8 import itertools
9
10 accounts = [] # [User1, User2, ...]
11 tweets_per_account = [] # [User1Tweets, User2Tweets, ....]
12 # {USER: TEXT:},
13 # {USER: TEXT:}
14
15 words_per_account = [] # [User1[], User2, ....]
16 # {word1,
17 # word2,
18 # ...},
19
20 all_words = [] # {"WORD":word, "COUNT":count_of_accounts_feat}
21 final_words = []
22 # Need to save to file
23 common_words_per_account = [] # [User1[], User2[], ....]
24 # [(word1,count),
25 # (word2,count),
26 # ...],
```

```
27
28 accounts_with_common = [] # {USER: user, "WORD":word, "OCCUR":
    count_of_accounts_feat}}
29
30 common_all_words = {} # Top 1000, used for Matrix
31 apcount={}
32 final_common = {}
33 matrix_words = []
34 matrix = []
35
36 def process():
37     global apcount, common_all_words, final_common
38     data = readJSONFile()
39     matchTweets(data)
40     print("Getting words per account...")
41     for user in tweets_per_account:
42         clean_words=[]
43         for tweet in user:
44             clean_words.append(cleanText(tweet["TEXT"]))
45         words_per_account.append(clean_words)
46
47     print("Getting apcount...")
48     apcount = getApCount(words_per_account)
49     print("Getting all words...")
50     populateAllWords()
51
52     print("Removing stopwords per account...")
53
54     fakeStopwordsTFIDF(apcount)
55     countFinalWords()
56
57     sorted_common = dict(sorted(common_all_words.items(), key=
itemgetter(1),reverse=True))
58
59     final_common = dict(itertools.islice(sorted_common.items(),1000))
60
61     createMatrix()
62
63 def createMatrix():
64     fwords = list(final_common.keys())
65     temp = []
66     for user in words_per_account:
67         row = {x:0 for x in fwords}
68         #row = {account: {x:0 for x in fwords} for account in accounts}
69         for key,value in row.items():
70             for words in user:
71                 for word in words:
```



```
72         if word == key:
73             row[word] += 1
74     matrix.append(row)
75
76     #for user in temp:
77         #row = {account: {word:key for word,key in user.items()} for
account in accounts}
78         #matrix.append(row)
79
80 def countFinalWords():
81     global common_all_words
82     #top_words = {}
83     bar = IncrementalBar('Counting', max=len(final_words))
84     for word in final_words:
85         for user in words_per_account:
86             for w in user:
87                 for x in w:
88                     if word == x:
89                         common_all_words[word] = common_all_words.get(
word, 0) + 1
90     bar.next()
91     bar.finish()
92
93 def populateAllWords():
94     for user in words_per_account:
95         for words in user:
96             for word in words:
97                 all_words.append(word)
98
99 def getApCount(words_list):
100     global common_words_per_account
101     ap = {}
102     for user in words_list:
103         common_words = []
104         words_no_dupes = []
105         for words in user:
106             [words_no_dupes.append(x) for x in words if x not in
words_no_dupes]
107
108         temp = sorted(getMostCommonTerms(words), key=itemgetter(1),
reverse=True)
109
110         common_words.append(temp)
111
112         for word in words_no_dupes:
113             ap[word] = ap.get(word, 0) + 1
114
```

```
115     #sorted_common = common_words.sort(key=itemgetter(0))
116     common_words_per_account.append(common_words)
117
118     return dict(sorted(ap.items(), key=itemgetter(1), reverse=True))
119
120 def getMostCommonTerms(list):
121     common_words = Counter(list)
122
123     most_common = common_words.most_common(1000)
124
125     return most_common
126
127 def matchTweets(data):
128     global tweets_per_account
129     for account in accounts:
130         tweets_from_account = []
131         for item in data:
132             if account == item["USER"]:
133                 for tweet in item["TWEETS"]:
134                     tweet_dict = {'USER':account, 'TEXT':tweet["TEXT"]}
135                     tweets_from_account.append(tweet_dict)
136         tweets_per_account.append(tweets_from_account)
137
138 def matchCommonTerms():
139     for index, user in enumerate(accounts):
140         for words in common_words_per_account[index]:
141             for word in words:
142                 word_dict = {"USER":user, "WORD":word[0], "FREQ":word[1]}
143                 accounts_with_common.append(word_dict)
144
145 def readJSONFile():
146     with open("h9_tweets.json", 'r') as f:
147         data = json.load(f)
148     return data
149
150 def readAccounts():
151     with open("100Accounts.txt", 'r') as f:
152         for line in f:
153             line = line.rstrip('\n')
154             accounts.append(line)
155
156 def cleanText(text):
157     clean_text = text.lower()
158
159     clean_no_mention = re.sub(r"@[\w]+", "", clean_text) # remove
160     @mentions
```

```
161     # remove links, punctuation, emojis, and hashtags
162     clean_no_chars = " ".join(re.sub(r"([0-9A-Za-z \t])|(\w+:\/\/\S+) ",
163     , "", clean_no_mention).split())
164
165     text_word_list = clean_no_chars.split()
166
167     clean_words_list = []
168
169     for word in text_word_list:
170         if len(word) >= 3 and len(word) <= 15:
171             clean_words_list.append(word)
172
173     return clean_words_list
174
175 def fakeStopwordsTFIDF(apcount):
176     global final_words
177     #num_accounts = 100
178
179     bar = IncrementalBar('Processing', max=len(apcount))
180     for word,ac in apcount.items():
181         frac = float(ac) / 100
182         if frac>0.1 and frac<0.5:
183             final_words.append(word)
184         bar.next()
185     bar.finish()
186
187 def writeFinalWordsToFile():
188     with open("FinalWords.json","w") as f:
189         json.dump(final_common,f,indent=2)
190         #for word in final_common:
191             #f.write(word+"\n")
192
193 def readInFinalWords():
194     with open("FinalWords.json",'r') as f:
195         data = json.load(f)
196
197     for key,value in data.items():
198         temp_dict = {"WORD":key,"COUNT":value}
199         matrix_words.append(temp_dict)
200
201 def writeCommonTermsPerAccount():
202     with open("Common_Terms_Per_File.txt","w") as f:
203         for item in accounts_with_common:
204             f.write("User: "+item["USER"]+"\n")
205             f.write("Word: "+item["WORD"]+"\n")
206             f.write("Occurences across account: "+str(item["FREQ"])+"\n\n")
```

```
206
207 def writeMatrixData():
208     with open("MatrixData.json", "w") as f:
209         json.dump(matrix, f, indent=2)
210         #for row in matrix:
211             #json.dump(row, f)
212             #f.write("\n")
213
214 def testOutput():
215     #for tweet in tweets_per_account[0]:
216         #print(tweet)
217         #print("\n")
218
219     #for words in words_per_account[1]:
220         #print(words)
221         #print("\n")
222     #print(all_words)
223
224     for tweet in common_words_per_account[0]:
225         for word in tweet:
226             print(word)
227             print("\n"+"-"*25)
228
229 if __name__ == '__main__':
230     readAccounts()
231     #readInFinalWords()
232     #processFinalWords()
233     process()
234     writeMatrixData()
235     #writeCommonTermsPerAccount()
236     #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.

```
1 [
2   {
3     "sorry": 0,
4     "election": 44,
5     "covid19": 3,
6     "app": 0,
7     "christmas": 0,
8     "song": 0,
9     "details": 0,
```

```
10     "birthday": 0,  
11     "phone": 1,  
12     "store": 0,  
13     "trump": 0,  
14     "address": 1,  
15     "news": 1,  
16     "location": 0,  
17     "nba": 4,  
18     "name": 1,  
19     "collection": 1,  
20     "games": 0,  
21     "email": 0,  
22     "december": 1,  
23     "via": 0,  
24     "health": 13,  
25     "care": 17,  
26     "album": 0,  
27     "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.

```
1 import json  
2 import re  
3 from collections import Counter  
4 from operator import itemgetter  
5 from progress.bar import IncrementalBar  
6 import itertools  
7 import array  
8 import pandas as pd  
9 import csv  
10  
11 matrix = []  
12 final_matrix = {}  
13 accounts = []  
14  
15 def process():
```

```

16 bar = IncrementalBar('Processing', max=len(matrix))
17 for index, user in enumerate(matrix):
18     final_matrix[accounts[index]] = {word:key for word, key in user.
    items()}
19     bar.next()
20 bar.finish()
21
22 table = pd.DataFrame(final_matrix).T
23 print(table)
24
25 #with open('matrix.txt', "w") as f:
26     #f.write(table.to_string())
27
28 def readInData():
29     global matrix
30     with open("MatrixData.json", "r") as f:
31         data=json.load(f)
32
33     for item in data:
34         matrix.append(item)
35
36     with open("100Accounts.txt", 'r') as a:
37         for line in a:
38             line = line.rstrip('\n')
39             accounts.append(line)
40
41 if __name__ == '__main__':
42     readInData()
43     process()

```

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

	sorry	election	covid19	...	present	finish	blast
1 BarackObama	0	44	3	...	0	0	0
2 justinbieber	0	0	0	...	0	0	0
3 katyperry	2	4	0	...	1	0	0
4 rihanna	0	0	1	...	0	0	0
5 realDonaldTrump	0	5	0	...	0	0	0
6
7 Respawn	0	0	0	...	1	1	0
8 Charmin	4	0	0	...	0	0	1
9 marshmellomusic	0	0	0	...	2	1	0
10 MerriamWebster	0	1	0	...	0	0	0
11 BaskinRobbins	67	0	0	...	0	0	1
12							
13							
14 [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.

```
1 import plotly as py
2 import plotly.figure_factory as FF
3 import json
4 from operator import itemgetter
5 from math import sqrt
6 import random
7 from PIL import Image, ImageDraw
8 import numpy as np
9 import matplotlib.pyplot as plt
10
11 def process():
12     users, words, counts = readFile()    # rownames,colnames,data
13
14     clust = hcluster(counts)
15
16     printclust(clust, labels=users)
17
18     rdata = rotatematrix(counts)
19     rclust = hcluster(rdata)
20
21     arr = np.array(rdata)
22
23     fig = FF.create_dendrogram(arr, orientation='left', labels=words)
24     fig.update_layout(width=800, height=11000)
25     fig.show()
26     drawdendrogram(rclust, labels=words, jpeg='clusters.jpg')
27
28     print("For k=5")
29     k1=kcluster(counts, k=5)
30     writeToFile(k1,users,5)
31
32     print("For k=10")
33     k2=kcluster(counts, k=10)
34     writeToFile(k2,users,10)
35
36     print("For k=20")
```

```
37     k3=kcluster(counts, k=20)
38     writeToFile(k3,users,20)
39
40     cords = scaledown(counts)
41     draw2d(cords,users, jpeg='mds2d.jpg')
42     draw2dAgain(cords,users)
43
44
45 class bicluster:
46
47     def __init__(
48         self,
49         vec,
50         left=None,
51         right=None,
52         distance=0.0,
53         id=None,
54     ):
55         self.left = left
56         self.right = right
57         self.vec = vec
58         self.id = id
59         self.distance = distance
60
61 def pearson(v1, v2):
62     # Simple sums
63     sum1 = sum(v1)
64     sum2 = sum(v2)
65
66     # Sums of the squares
67     sum1Sq = sum([pow(v, 2) for v in v1])
68     sum2Sq = sum([pow(v, 2) for v in v2])
69
70     # Sum of the products
71     pSum = sum([v1[i] * v2[i] for i in range(len(v1))])
72
73     # Calculate r (Pearson score)
74     num = pSum - sum1 * sum2 / len(v1)
75     den = sqrt((sum1Sq - pow(sum1, 2) / len(v1)) * (sum2Sq - pow(sum2,
76         2)
77         / len(v1)))
78     if den == 0:
79         return 0
80
81     return 1.0 - num / den
82 #Prints ASCII clusters
```



```
83 def hcluster(rows, distance=pearson):
84     distances = {}
85     currentclustid = -1
86
87     # Clusters are initially just the rows
88     clust = [bicluster(rows[i], id=i) for i in range(len(rows))]
89
90     while len(clust) > 1:
91         lowestpair = (0, 1)
92         closest = distance(clust[0].vec, clust[1].vec)
93
94         # loop through every pair looking for the smallest distance
95         for i in range(len(clust)):
96             for j in range(i + 1, len(clust)):
97                 # distances is the cache of distance calculations
98                 if (clust[i].id, clust[j].id) not in distances:
99                     distances[(clust[i].id, clust[j].id)] = \
100                         distance(clust[i].vec, clust[j].vec)
101
102                 d = distances[(clust[i].id, clust[j].id)]
103
104                 if d < closest:
105                     closest = d
106                     lowestpair = (i, j)
107
108         # calculate the average of the two clusters
109         mergevec = [(clust[lowestpair[0]].vec[i] + clust[lowestpair
110 [1]].vec[i])
111                     / 2.0 for i in range(len(clust[0].vec))]
112
113         # create the new cluster
114         newcluster = bicluster(mergevec, left=clust[lowestpair[0]],
115                                right=clust[lowestpair[1]], distance=
116                                closest,
117                                id=currentclustid)
118
119         # cluster ids that weren't in the original set are negative
120         currentclustid -= 1
121         del clust[lowestpair[1]]
122         del clust[lowestpair[0]]
123         clust.append(newcluster)
124
125     return clust[0]
126
127 def readFile():
128     with open ('matrix.txt', 'r') as f:
129         lines = [line for line in f]
```

```
128
129
130 # First line is the column titles
131 #colnames = lines[0].strip().split('\t')[1:]
132 colnames = lines[0].strip().split()
133 rownames = []
134 data = []
135 for line in lines[1:]:
136     p = line.strip().split()
137     # First column in each row is the rowname
138     rownames.append(p[0])
139     # The data for this row is the remainder of the row
140     data.append([float(x) for x in p[1:]])
141     return (rownames, colnames, data)
142
143 def rotatematrix(data):
144     newdata = []
145     for i in range(len(data[0])):
146         newrow = [data[j][i] for j in range(len(data))]
147         newdata.append(newrow)
148     return newdata
149
150 def printclust(clust, labels=None, n=0):
151     with open("ASCII_Cluster.txt", "a") as f:
152
153         # indent to make a hierarchy layout
154         for i in range(n):
155             print(' ')
156             f.write(' ')
157         if clust.id < 0:
158             # negative id means that this is branch
159             print ('-')
160             f.write('-'+'\n')
161         else:
162             # positive id means that this is an endpoint
163             if labels == None:
164                 print(clust.id)
165                 f.write(clust.id+'\n')
166             else:
167                 print(labels[clust.id])
168                 f.write(labels[clust.id]+'\n')
169
170         # now print the right and left branches
171         if clust.left != None:
172             printclust(clust.left, labels=labels, n=n + 1)
173         if clust.right != None:
174             printclust(clust.right, labels=labels, n=n + 1)
```

```
175
176 def getheight(clust):
177     # Is this an endpoint? Then the height is just 1
178     if clust.left == None and clust.right == None:
179         return 1
180
181     # Otherwise the height is the same of the heights of
182     # each branch
183     return getheight(clust.left) + getheight(clust.right)
184
185 def getdepth(clust):
186     # The distance of an endpoint is 0.0
187     if clust.left == None and clust.right == None:
188         return 0
189
190     # The distance of a branch is the greater of its two sides
191     # plus its own distance
192     return max(getdepth(clust.left), getdepth(clust.right)) + clust.
        distance
193
194 def drawdendrogram(clust, labels, jpeg='clusters.jpg'):
195     # height and width
196     h = getheight(clust) * 20
197     w = 1200
198     depth = getdepth(clust)
199
200     # width is fixed, so scale distances accordingly
201     scaling = float(w - 150) / depth
202
203     # Create a new image with a white background
204     img = Image.new('RGB', (w, h), (255, 255, 255))
205     draw = ImageDraw.Draw(img)
206
207     draw.line((0, h / 2, 10, h / 2), fill=(255, 0, 0))
208
209     # Draw the first node
210     drawnode(
211         draw,
212         clust,
213         10,
214         h / 2,
215         scaling,
216         labels,
217     )
218     img.save(jpeg, 'JPEG')
219
220 def drawnode(
```

```
221     draw,
222     clust,
223     x,
224     y,
225     scaling,
226     labels,):
227     if clust.id < 0:
228         h1 = getheight(clust.left) * 20
229         h2 = getheight(clust.right) * 20
230         top = y - (h1 + h2) / 2
231         bottom = y + (h1 + h2) / 2
232         # Line length
233         ll = clust.distance * scaling
234         # Vertical line from this cluster to children
235         draw.line((x, top + h1 / 2, x, bottom - h2 / 2), fill=(255, 0,
236         0))
237         # Horizontal line to left item
238         draw.line((x, top + h1 / 2, x + ll, top + h1 / 2), fill=(255,
239         0, 0))
240         # Horizontal line to right item
241         draw.line((x, bottom - h2 / 2, x + ll, bottom - h2 / 2), fill
242         =(255, 0,
243         0))
244         # Call the function to draw the left and right nodes
245         drawnode(
246             draw,
247             clust.left,
248             x + ll,
249             top + h1 / 2,
250             scaling,
251             labels,
252             )
253         drawnode(
254             draw,
255             clust.right,
256             x + ll,
257             bottom - h2 / 2,
258             scaling,
259             labels,
260             )
261     else:
262         # If this is an endpoint, draw the item label
263         draw.text((x + 5, y - 7), labels[clust.id], (0, 0, 0))
264
```

```
265 def kcluster(rows, distance=pearson, k=4):
266     # Determine the minimum and maximum values for each point
267     ranges = [(min([row[i] for row in rows]), max([row[i] for row in
268         rows]))
269               for i in range(len(rows[0]))]
270     # Create k randomly placed centroids
271     clusters = [[random.random() * (ranges[i][1] - ranges[i][0]) +
272         ranges[i][0]
273                 for i in range(len(rows[0]))] for j in range(k)]
274     lastmatches = None
275     for t in range(100):
276         print ('Iteration %d' % t)
277         bestmatches = [[] for i in range(k)]
278     # Find which centroid is the closest for each row
279     for j in range(len(rows)):
280         row = rows[j]
281         bestmatch = 0
282         for i in range(k):
283             d = distance(clusters[i], row)
284             if d < distance(clusters[bestmatch], row):
285                 bestmatch = i
286             bestmatches[bestmatch].append(j)
287     # If the results are the same as last time, this is complete
288     if bestmatches == lastmatches:
289         break
290     lastmatches = bestmatches
291     # Move the centroids to the average of their members
292     for i in range(k):
293         avgs = [0.0] * len(rows[0])
294         if len(bestmatches[i]) > 0:
295             for rowid in bestmatches[i]:
296                 for m in range(len(rows[rowid])):
297                     avgs[m] += rows[rowid][m]
298             for j in range(len(avgs)):
299                 avgs[j] /= len(bestmatches[i])
300             clusters[i] = avgs
301     return bestmatches
302
303 def scaledown(data, distance=pearson, rate=0.01):
304     n = len(data)
```

```
310 # The real distances between every pair of items
311 realdist = [[distance(data[i], data[j]) for j in range(n)] for i in
312             range(0, n)]
313
314 # Randomly initialize the starting points of the locations in 2D
315 loc = [[random.random(), random.random()] for i in range(n)]
316 fakedist = [[0.0 for j in range(n)] for i in range(n)]
317
318 lasterror = None
319 for m in range(0, 1000):
320     # Find projected distances
321     for i in range(n):
322         for j in range(n):
323             fakedist[i][j] = sqrt(sum([pow(loc[i][x] - loc[j][x],
324                                         2)
325                                         for x in range(len(loc[i]))]))
326
327     # Move points
328     grad = [[0.0, 0.0] for i in range(n)]
329
330     totalerror = 0
331     for k in range(n):
332         for j in range(n):
333             if j == k:
334                 continue
335             # The error is percent difference between the distances
336             errorterm = (fakedist[j][k] - realdist[j][k]) /
337             realdist[j][k]
338
339             # Each point needs to be moved away from or towards the other
340             # point in proportion to how much error it has
341             grad[k][0] += (loc[k][0] - loc[j][0]) / fakedist[j][k]
342             \
343             * errorterm
344             grad[k][1] += (loc[k][1] - loc[j][1]) / fakedist[j][k]
345             \
346             * errorterm
347
348             # Keep track of the total error
349             totalerror += abs(errorterm)
350             print (totalerror)
351
352     # If the answer got worse by moving the points, we are done
353     if lasterror and lasterror < totalerror:
354         break
355     lasterror = totalerror
```

```


353     # Move each of the points by the learning rate times the gradient
354     for k in range(n):
355         loc[k][0] -= rate * grad[k][0]
356         loc[k][1] -= rate * grad[k][1]
357
358     return loc
359
360 def draw2d(data, labels, jpeg='mds2d.jpg'):
361     img = Image.new('RGB', (2000, 2000), (255, 255, 255))
362     draw = ImageDraw.Draw(img)
363     for i in range(len(data)):
364         x = (data[i][0] + 0.5) * 1000
365         y = (data[i][1] + 0.5) * 1000
366         draw.text((x, y), labels[i], (0, 0, 0))
367     img.save(jpeg, 'JPEG')
368
369 def draw2dAgain(data, labels):
370     x_axis = []
371     y_axis = []
372
373     for i in range(len(data)):
374         x = (data[i][0] + 0.5) * 1000
375         y = (data[i][1] + 0.5) * 1000
376         x_axis.append(x)
377         y_axis.append(y)
378         plt.text(x*(1+0.01), y*(1+0.01), labels[i], fontsize=10)
379
380     plt.scatter(x_axis, y_axis)
381     plt.show()
382
383
384 def writeToFile(klist, users, k):
385     with open("k{}_results.txt".format(k), 'w') as f:
386         f.write("Iterations at k="+str(k)+"): "+str(len(klist))+ "\n\n")
387         for cluster in klist:
388             for u in cluster:
389                 f.write(users[u]+ "\n")
390             f.write("\n")
391
392
393 if __name__ == '__main__':
394     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 its 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

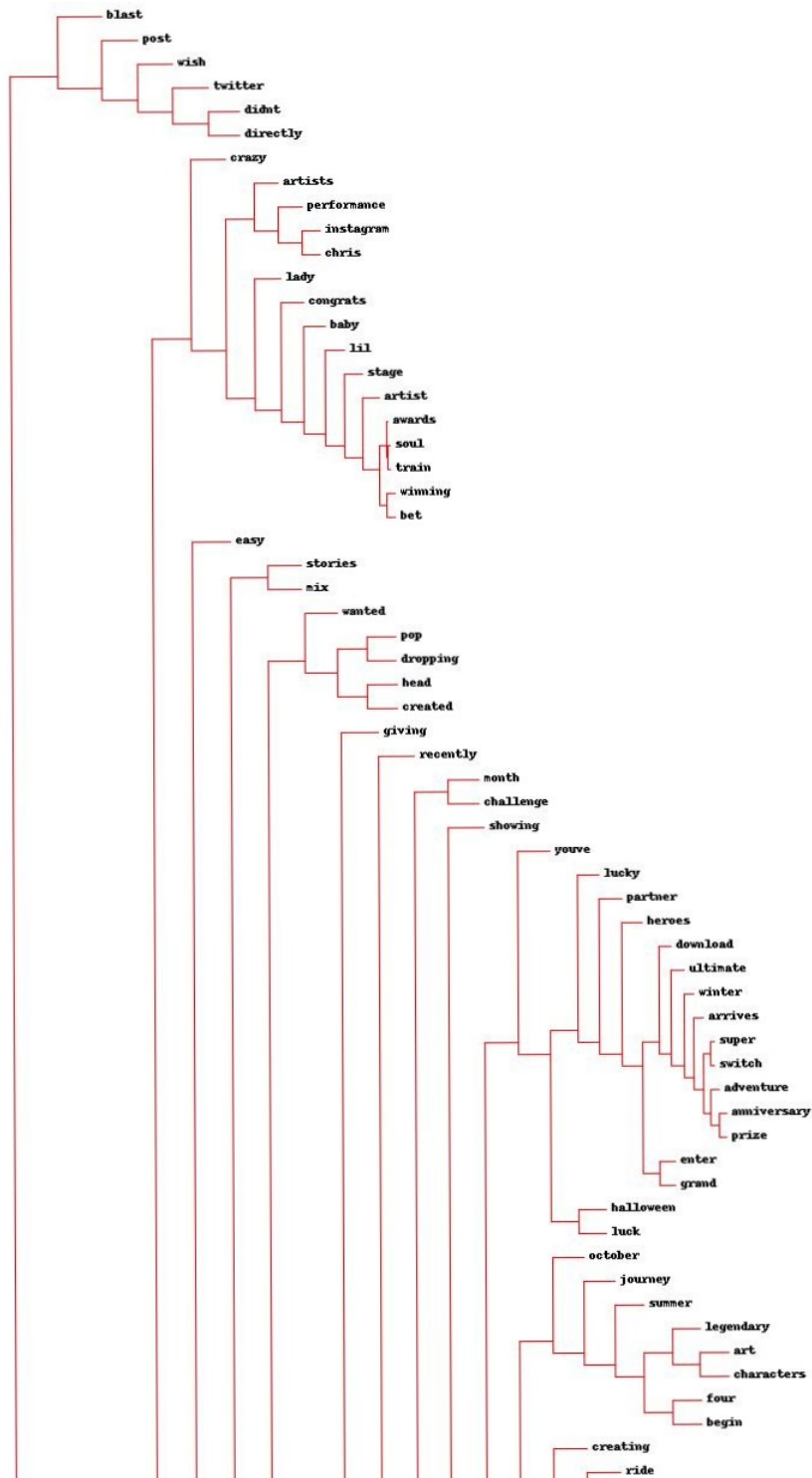


Figure 2: Snippet of dendrogram

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.

```
1 For k=5
2 Iteration 0
3 Iteration 1
4 Iteration 2
5 Iteration 3
6 Iteration 4
7 Iteration 5
8
9 For k=10
10 Iteration 0
11 Iteration 1
12 Iteration 2
13 Iteration 3
14 Iteration 4
15 Iteration 5
16 Iteration 6
17 Iteration 7
18 Iteration 8
19 Iteration 9
20 Iteration 10
21 Iteration 11
22 Iteration 12
23 Iteration 13
24
25 For k=20
26 Iteration 0
27 Iteration 1
28 Iteration 2
29 Iteration 3
30 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.

```
1 Iterations at k=(5): 5
2
3 matthewmercerc
4 NFL
5 Wendys
6 dominos
7 McDonalds
8 BurgerKing
9 PapaJohns
10 pizzahut
11 BaskinRobbins
12
13 BarackObama
14 ladygaga
15 selenagomez
16 ddlovato
17 shakira
18 Oprah
19 NASA
20 elonmusk
21 HillaryClinton
22 pitbull
23 NatGeo
24 ricky_martin
25 WHO
26 Microsoft
27 NintendoAmerica
28 EA
29 Blizzard_Ent
30 bethesda
31 WuTangClan
32 littlecaesars
33 Bungie
34 Treyarch
35 Respawn
36 Charmin
```

```
37
38 justinbieber
39 katyperry
40 rihanna
41 ArianaGrande
42 TheEllenShow
43 YouTube
44 KimKardashian
45 KingJames
46 MileyCyrus
47 JLo
48 BeingSalmanKhan
49 KevinHart4real
50 wizkhalifa
51 KylieJenner
52 KendallJenner
53 aliciakeys
54 coldplay
55 MariahCarey
56 NICKIMINAJ
57 davidguetta
58 JERICHO
59 SavinTheBees
60 Ninja
61 PostMalone
62 hulu
63 netflix
64 adamlevine
65 MLB
66 TheRock
67 WNBA
68 Sony
69 NintendoUK
70 FIFAcorn
71 michaelb4jordan
72 BET
73 BETMusic
74 NathanFillion
75 TroyBakerVA
76 WayneBrady
77 tacobell
78 MrPeanut
79 RoosterTeeth
80 Monstercat
81 PegboardNerds
82 marshmellomusic
83 MerriamWebster
```

```
84
85 espn
86 Google
87 NBA
88 Windows
89 PaulMcCartney
90
91 realDonaldTrump
92 CNN
93 jimmyfallon
94 nytimes
95 TheDailyShow
96 FoxNews
97 NHL
98 RealRonHoward
99 BigMikeJ73
100 AOC
101 ToddHaberKorn
102 Crunchyroll
103 StephenAtHome
104 KenJennings
105 TheTweetOfGod
106 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.

```
1 Iterations at k=(10): 13
2
3 realDonaldTrump
4 CNN
5 nytimes
6 NatGeo
7 TheDailyShow
8 FoxNews
9 AOC
10 ToddHaberKorn
11 TheTweetOfGod
12 TheOnion
13
14 BarackObama
15 ladygaga
16 TheEllenShow
17 YouTube
18 ddlovato
19 Oprah
20 NASA
21 elonmusk
22 HillaryClinton
23 bethesda
24 NathanFillion
25 tacobell
26 StephenAtHome
27 Treyarch
28 marshmellomusic
29
30 KimKardashian
31 shakira
32 KylieJenner
33 KendallJenner
34 pitbull
35 ricky_martin
36 WHO
37 Microsoft
38 NHL
39 PaulMcCartney
40 EA
41 Blizzard_Ent
42 WuTangClan
43 littlecaesars
44 MrPeanut
45 KenJennings
```

```
46 Bungie
47 Respawn
48
49 justinbieber
50 selenagomez
51 BeingSalmanKhan
52 KevinHart4real
53 TheRock
54 NintendoAmerica
55 NintendoUK
56 RoosterTeeth
57
58 NFL
59
60 katyperry
61 rihanna
62 ArianaGrande
63 KingJames
64 MileyCyrus
65 JLo
66 wizkhalifa
67 aliciakeys
68 coldplay
69 MariahCarey
70 NICKIMINAJ
71 davidguetta
72 JERICHO
73 SavinTheBees
74 Ninja
75 PostMalone
76 hulu
77 netflix
78 adamlevine
79 Sony
80 michaelb4jordan
81 BET
82 BETMusic
83 TroyBakerVA
84 WayneBrady
85 Monstercat
86 PegboardNerds
87
88 matthewmercercer
89 Windows
90 RealRonHoward
91 Wendys
92 dominos
```



```
93 McDonalds
94 PapaJohns
95 Charmin
96 MerriamWebster
97 BaskinRobbins
98
99 WNBA
100 BigMikeJ73
101 BurgerKing
102 pizzahut
103
104 jimmyfallon
105 espn
106 NBA
107 Crunchyroll
108
109 Google
110 MLB
111 FIFACom
```

Listing 11: Clustering at k=10

Listing 12 shows the different account clusters at k=20. Most 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

```
1 Iterations at k=(20): 4
2
3
4 KevinHart4real
```

```
5 TheRock
6
7 Google
8 NHL
9 NintendoUK
10 dominos
11
12 NathanFillion
13 TheTweetOfGod
14 Treyarch
15
16 NFL
17
18 Microsoft
19
20 MariahCarey
21 JERICHO
22 Ninja
23 bethesda
24
25 rihanna
26 KingJames
27 espn
28 NICKIMINAJ
29 SavinTheBees
30 BigMikeJ73
31 michaelb4jordan
32
33 wizkhalifa
34 PapaJohns
35 MrPeanut
36 StephenAtHome
37
38 ArianaGrande
39 shakira
40 BeingSalmanKhan
41 matthewmercercer
42 netflix
43 WNBA
44 Sony
45 WuTangClan
46 BaskinRobbins
47
48 TheEllenShow
49 KendallJenner
50 MLB
51 FIFACom
```

```
52 EA
53 BETMusic
54 Bungie
55 Charmin
56
57 pitbull
58 hulu
59 NintendoAmerica
60 PaulMcCartney
61 RealRonHoward
62 Respawn
63
64 NBA
65
66 BarackObama
67 realDonaldTrump
68 ladygaga
69 CNN
70 nytimes
71 Oprah
72 HillaryClinton
73 TheDailyShow
74 FoxNews
75 AOC
76 littlecaesars
77 TheOnion
78
79 jimmyfallon
80 BET
81 ToddHaberKorn
82 Crunchyroll
83 tacobell
84
85 KimKardashian
86 KylieJenner
87 Wendys
88 McDonalds
89 BurgerKing
90 pizzahut
91 RoosterTeeth
92
93 NASA
94 elonmusk
95 MerriamWebster
96
97 Windows
98 Blizzard_Ent
```

```
99 TroyBakerVA
100 WayneBrady
101 PegboardNerds
102 KenJennings
103
104 justinbieber
105 katyperry
106 YouTube
107 selenagomez
108 ddlovato
109 MileyCyrus
110 JLo
111 aliciakeys
112 NatGeo
113 coldplay
114 davidguetta
115 ricky_martin
116 WHO
117 PostMalone
118 adamlevine
119 Monstercat
120 marshmellomusic
```

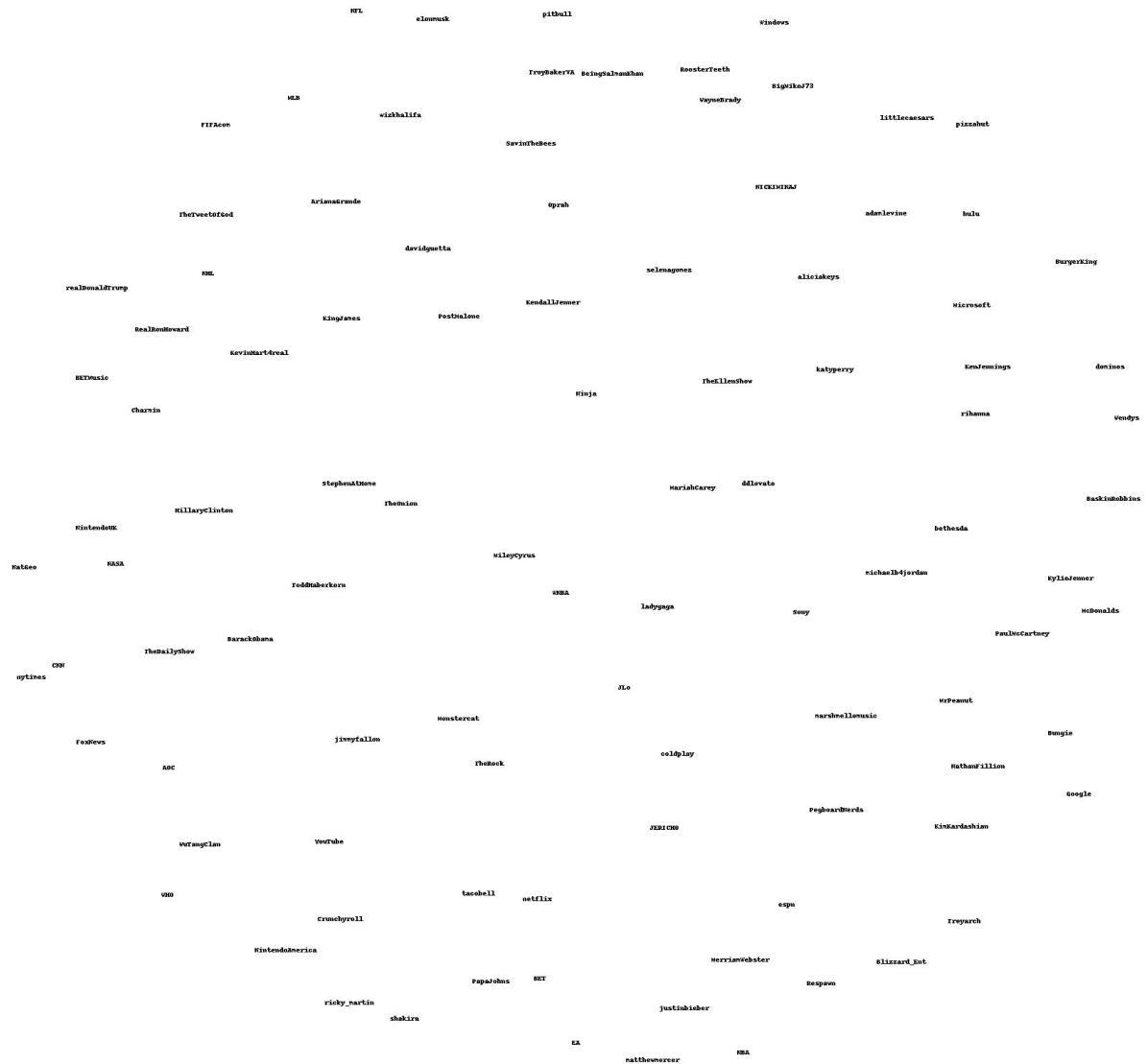
Listing 12: Clustering at k=20

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 its 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 to 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 its extreme size, it had to be scaled down to capture the full, 3-dimensional effect.



- 1 4292.093979274119
- 2 3468.6963378241735
- 3 3443.1605133694075
- 4 3433.8771891736046
- 5 3428.3435921231517
- 6 3424.1729984429485
- 7 3421.120766401213
- 8 3418.7469839225996
- 9 3416.4118497193167
- 10 3414.4440844275505

```
11 3412.440863751611
12 3410.6465210733254
13 3409.0135600255144
14 3407.828801597129
15 3406.614686573917
16 3405.348619533868
17 3403.879972210023
18 3402.4599352017785
19 3400.6049413304436
20 3398.4835025635516
21 3396.4503913590497
22 3394.498279149446
23 3392.6385673344253
24 3390.574933345304
25 3388.2408982400316
26 3385.945677397648
27 3383.644519964651
28 3381.3471770664746
29 3379.071768466391
30 3376.9061201468085
31 3374.86572041902
32 3372.7470693075834
33 3370.679695018536
34 3368.8020144909347
35 3367.078404372282
36 3365.4284726916517
37 3363.5180227427154
38 3361.5044352044524
39 3359.567836902659
40 3357.6994851742597
41 3356.075201494575
42 3354.6152534564762
43 3353.3437071308554
44 3352.335510480809
45 3351.321018381074
46 3350.1888786790323
47 3348.985883309517
48 3347.780406455964
49 3346.5534605142593
50 3345.199031162315
51 3343.751169397896
52 3342.243964528533
53 3340.5999192427143
54 3339.0387078681683
55 3337.450064806221
56 3335.923911769075
57 3334.5058497032032
```

```
58 3333.1150908016352
59 3331.899056124837
60 3330.6460139440615
61 3329.390198500818
62 3328.2085720215514
63 3326.942245522004
64 3325.6903331491635
65 3324.731009673131
66 3323.9414807388216
67 3323.252399237465
68 3322.621871481046
69 3321.966053962298
70 3321.3863407367453
71 3320.7252626481722
72 3320.1258695526835
73 3319.5913971475425
74 3319.042673119391
75 3318.3179305943013
76 3317.530619079285
77 3316.6666483553036
78 3315.6227629071536
79 3314.6336580402417
80 3313.929465632325
81 3313.255518125941
82 3312.461244955732
83 3311.738311092058
84 3310.991010500956
85 3310.235490785789
86 3309.330568837568
87 3308.2826330791786
88 3307.2518796395166
89 3306.6054579567126
90 3305.799861116069
91 3305.253415207178
92 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_factory* 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.

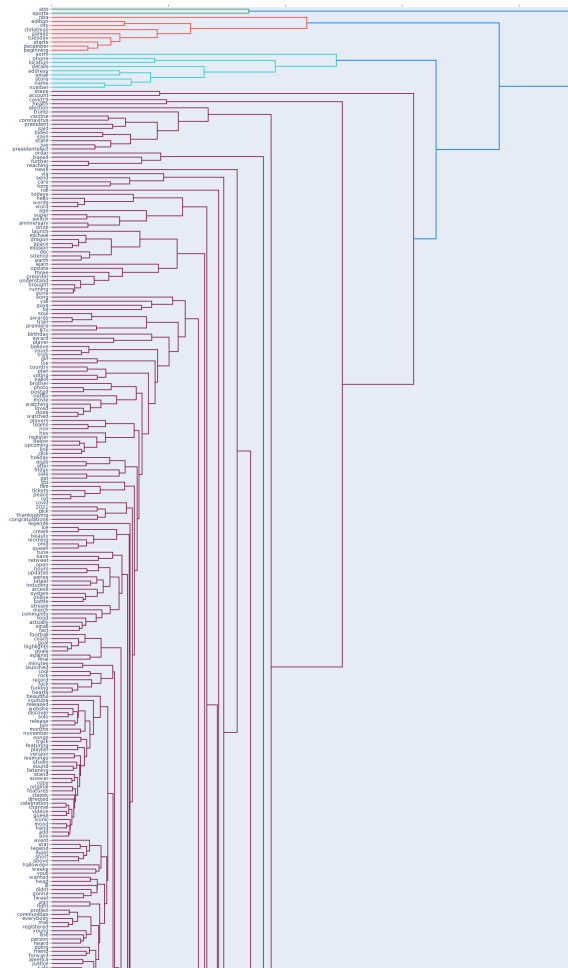


Figure 4: Snippet of dendrogram plotted using Plotly

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.

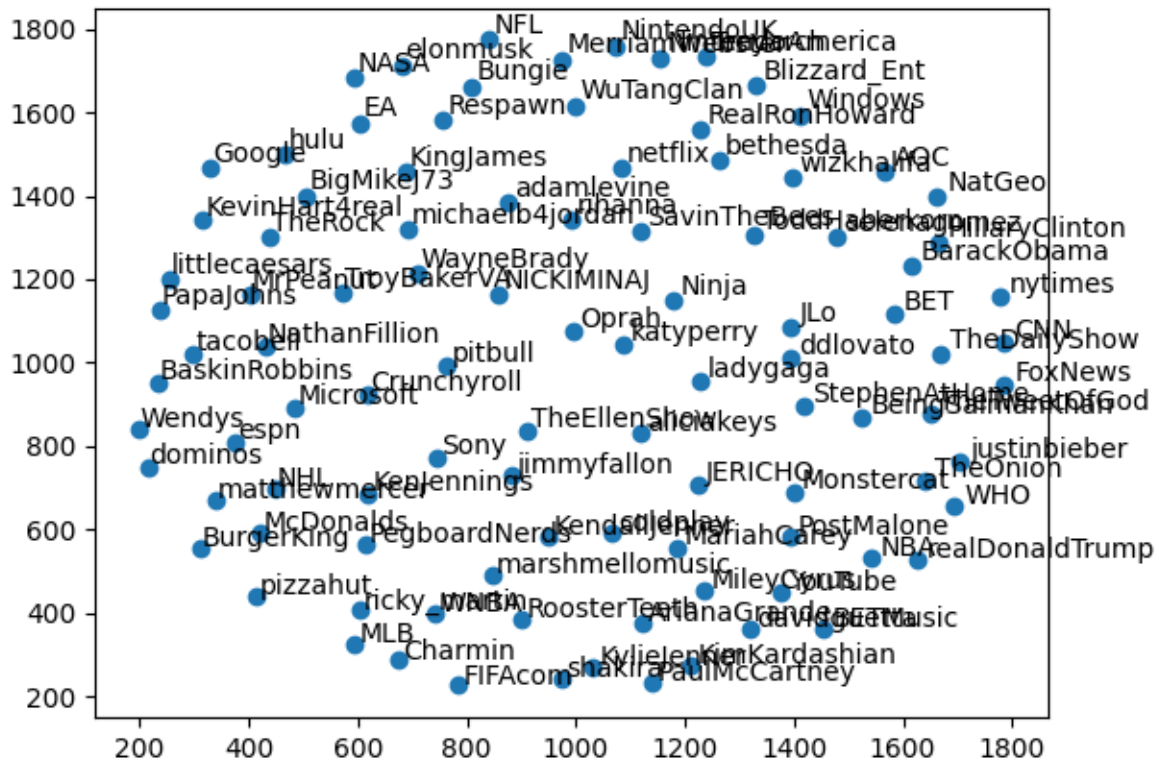


Figure 5: MDS plotted using PyPlot