Text Analytics Practical 5

Question 1

Question 1 a)

I modified the *Jaccard Index* formula to do *Jaccard Distance* and computed all pairwise distances between my six entities. Below is the code and some of those results.

```
def JaccardDistance(str1, str2):
    set1 = set(str1.split())
    set2 = set(str2.split())
    ans = float(len(set1 | set2) - len(set1 & set2)) / float(len(set1 | set2))
    return round(ans, 2)
```

Entity 1

```
Entities 1 and 2: 0.71
Entities 1 and 3: 1.0
Entities 1 and 4: 1.0
Entities 1 and 5: 0.88
Entities 1 and 6: 1.0… (and so on…)
```

```
# Lets take two examples to show triangle inequality holds for measure
point1 = JaccardDistance(entities[0], entities[1])
point2 = JaccardDistance(entities[1], entities[4])
# Code omitted => I selected six points...

# Show Triangle Inequality Holds for Measure
print(point1, "<=", point2 + point3)
print(point2, "<=", round(point1 + point3, 2))
print(point3, "<=", point2 + point1)
print(point4, "<=", point5 + point6)
print(point5, "<=", round(point4 + point6, 2))
print(point6, "<=", point4 + point5)
```

```
0.71 <= 1.88
1.0 <= 1.59
0.88 <= 1.71

1.0 <= 1.9
0.9 <= 2.0
1.0 <= 1.9
```

It can clearly be seen empirically that the property of triangle inequality holds for measure.

Question 1 b)

I implemented the difference function for the *Dice Coefficient* and discovered with the same points as taken in the previous question that the property of triangle inequality may not always hold for this measure. For points 1, 2 and 3 (the same points as in question 1 a) the following were the results.

```
0.44 <= 0.22
0.0 <= 0.66
0.22 <= 0.44
```

It can clearly be seen that the first line does not hold for triangle inequality.

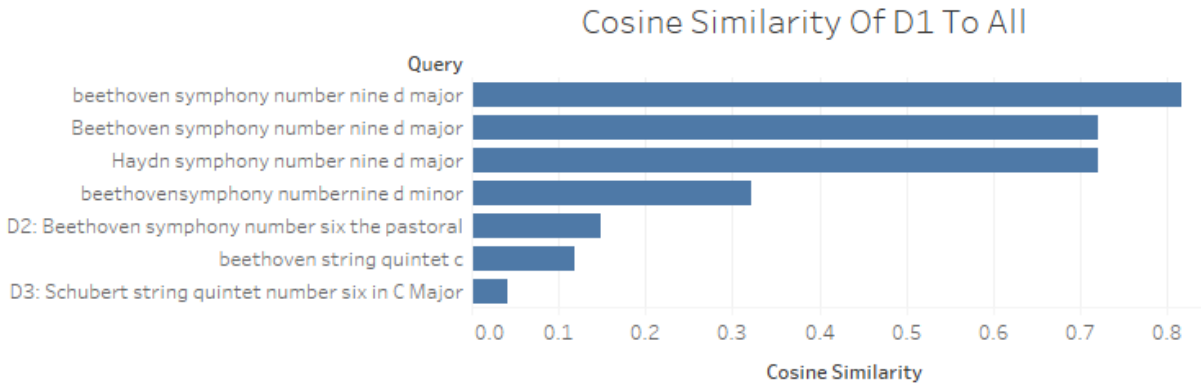
Question 2)

Question 2 a)

I chose three documents and produced five variants on one of them (D1: 'Beethoven symphony number nine in D Major') to see how the cosine similarity changes, below are the results.

```
Cosine to query 1 is: 0.817
Cosine to query 2 is: 0.72
Cosine to query 3 is: 0.72
Cosine to query 4 is: 0.119
Cosine to query 5 is: 0.322
Cosine to d2 is: 0.148
Cosine to d3 is: 0.042
```

Question 2 b)



The document which produced the closest score was the first query. The only difference in these two was the use of different scalers for each component of the vector. For example, the query has binary values only but the score is calculated against a TF-IDF matrix of non-binary values (mostly). Considering this it would be almost impossible for the query to have a perfect match with the matrix which this verifies.

Query two simply used a capital 'B', which affected the score slightly, however query 3 had an entirely different word which produced the same result. Taking this into account it appears that normalisation is exceedingly important, otherwise 'Beethoven' and 'beethoven' will give different results, which is undesirable.

Overall my intuitions are confirmed in that the more the documents are dissimilar the worse the cosine score.

Question 2 c)

For computing cosine similarity I used the package *sklearn.feature_extraction.text.TfidfVectorizer* and *sklearn.metrics.pairwise.cosine_similarity*. The original results for comparing the three documents were:

```
Cosine of doc 1-2 is: 0.15
Cosine of doc 1-3 is: 0.04
Cosine of doc 3-2 is: 0.06
```

When using the library, the results were:

```
d1-d2: 0.79
d2-d3: 0.51
d1-d3: 0.51
```

This code used to get these results was:

```
# Compute the TF-IDF Scores
vect = TfidfVectorizer(stop_words=['of', 'and', 'on', 'in', 'the'])
tfidf = vect.fit_transform(documents)

# Compute the Cosine Similarity Scores
matrix_ans = cosine_similarity(tfidf * tfidf.T)
```

They do not correspond, but are somewhat the same order/ratio, this is likely because the normalisation process in the library is different from the method shown in the lecture slides.

For computing Euclidean distance I used the library *scipy.spatial.distance*. The results were as follows:

```
d1-d2: 0.86
d2-d3: 1.11
d1-d3: 1.20
```

For Euclidean distance the smaller the value the closer the two documents are in term of similarity, taking this into account, the results are quite uniform across all three of the examples above. The most notable aspect of using this (Euclidean) method is that (in contrast to cosine similarity) you must take the complete set of words as your blueprint for the query vector when calculating it with the matrix columns, even if most of the values are 0. To illustrate this, I included the corresponding line in the *Cosine Similarity* function below commented out (the difference is using AND or OR)

```
def get_euclidean(text1, text2):
    vec1 = vector_dict[text1]
    vec2 = vector_dict[text2]
    intersection = set(vec1.keys()) | set(vec2.keys())
    # This above line in Cosine Similarity
    # intersection = set(vec1.keys()) & set(vec2.keys())
    vec1_list = list()
    vec2_list = list()
    for item in intersection:
        try:
            vec1_list.append(vec1[item])
        except:
            vec1_list.append(0.0)
        try:
            vec2_list.append(vec2[item])
        except:
            vec2_list.append(0.0)
    vec1_ans = tuple(vec1_list)
    vec2_ans = tuple(vec2_list)
    return distance.euclidean(vec1_ans, vec2_ans)
```

Question 3)

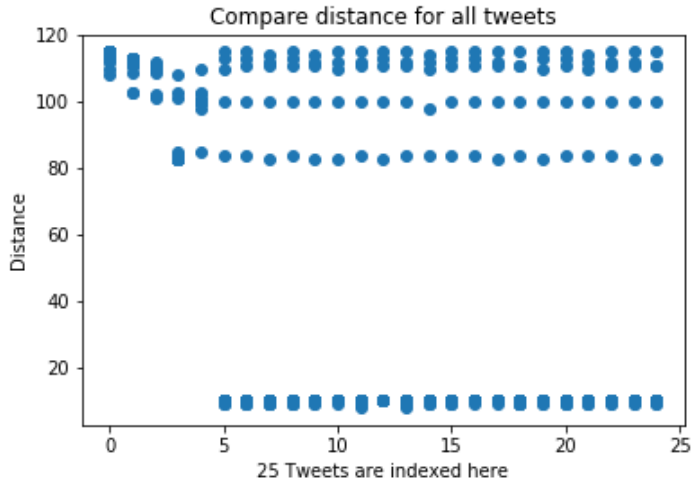
After having collected 5 tweets at random I chose one about pizza and generated 20 random spam tweets using the technique below which essentially assigns a random username and URL to append to the start and end of each tweet respectively. Each tweet also ends in similar hashtags, this is in accordance with the typical techniques of spammers.

```
# Generate spam tweets
for i in range(20):
    tweet = spam_tweet
    spam_at_start = "@twitterUser" + ''.join([random.choice(string.ascii_letters +
string.digits) for n in range(5)])
    spam_at_end = "http://RandSpam" + ''.join([random.choice(string.ascii_letters
+ string.digits) for n in range(5)]) + "#Pizza#ScrewHealthyEating"
    spam_tweets.append(spam_at_start + " " + spam_tweet + spam_at_end)
```

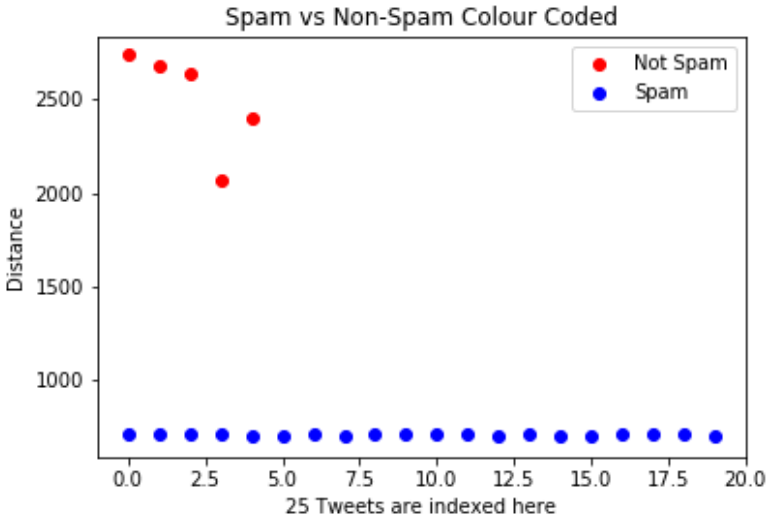
An example of the spam tweets is as follows:

```
'@twitterUservywuw Hungry for some pizza? Have Dominos and get a coupon
here!:http://RandSpamJq4gZ#Pizza#ScrewHealthyEating'
```

In approaching this part of the question, I decided to use the mindset that I didn’t know if/where the spam tweets were in the dataset, to mimic a real-life scenario. To do comparisons between these tweets I first attempted to cluster the data by observing the distance of every tweet to every other tweet.



Here there are clear clusters in the data which could be categorised. The x−axis is simply each tweet indexed for clarity of viewing, the y−axis represents 24 distances to each of the other tweets. As can clearly be seen, 20 of the tweets bear striking similarity scores to others. This method would be good for identifying spam tweets, but even with this toy example the code to compute this took a very long time, so perhaps other methods would be more feasible to scale this.



Shown above is the total distance between each tweet (indexed on the x axis) and every other tweet. I colour coded the spam/non−spam as instructed and used the total distance to all other tweets as my y−scale metric. Obviously, this approach would not be feasible on a large scale to identify spam tweets, something closer to the clustering shown in the previous plot above would be more suitable. However, the spam tweets are obvious from the plot above and because of the colour coding it can been seen that this method to isolate them worked effectively.