Practical 5

Text Analytics: Similarities

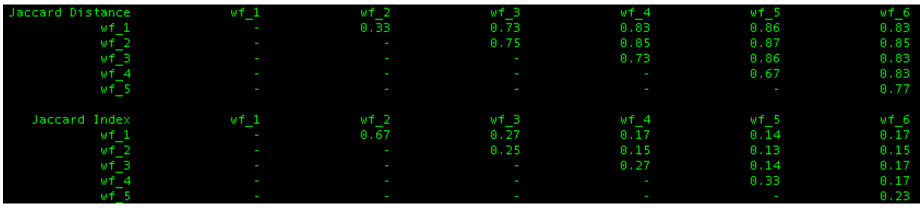
Otto Hermann
otto.hermann@ucdconnect.ie
16203034

COMP47600

University College Dublin
19th October 2017

1a: Jaccard Stuff

- wf_1: the talking heads are my favourite band
- wf_2: the talking heads are my favourite musical group
- wf_3: i enjoy listening to the talking heads
- wf_4: i hate talking heads on fox news
- wf_5: fox news is a gaggle of iditotic talking heads
- wf_6: talking heads annoy me a great deal



Jaccard Distance	wf_1	wf_2	wf_3	wf_4	wf_5	wf_6
wf_1	-	0.33	0.73	0.83	0.86	0.83
wf_2	-	-	0.75	0.85	0.87	0.85
wf_3	-	-	-	0.73	0.86	0.83
wf_4	-	-	-	-	0.67	0.83
wf_5	-	-	-	-	-	0.77

Jaccard Index	wf_1	wf_2	wf_3	wf_4	wf_5	wf_6
wf_1	-	0.67	0.27	0.17	0.14	0.17
wf_2	-	-	0.25	0.15	0.13	0.15
wf_3	-	-	-	0.27	0.14	0.17
wf_4	-	-	-	-	0.33	0.17
wf_5	-	-	-	-	-	0.23

Figure 1: Jaccard Pairwise Distances and Indices

To provide empirical evidence supporting the fact that triangle inequality holds for Jaccard Distance, I enumerate all possible permutations of the word features and compare, for A, B, and C that $d(A, C) \leq d(A, B) + d(B, C)$; the triangle inequality obtains for all 120 permutations. For analytic proof, see here.

```
example_count = 0 # keep track of examples generated
incorrect_count = 0 # keep track of number triangle inequality failures
for first_item in wf_list: # use each item in wf_list
  second_item_list = [w for w in wf_list]
  second_item_list.remove(first_item)
  for second_item in second_item_list: # use every other item in wf_list less the
      first_item
    third_item_list = [s for s in second_item_list]
    third_item_list.remove(second_item)
    for third_item in third_item_list: # use every other item in wf_list less
        first_item, second_item
      a = first_item
      b = second_item
      c = third_item
      jd_ac = jaccard_distance(a, c)
      jd_ab = jaccard_distance(a, b)
      jd_bc = jaccard_distance(b, c)
      example_count += 1
      if jd_ac > (jd_ab + jd_bc): # if ac > ab + bc, then the triangle inequality
          has failed
        incorrect_count += 1
        print("{} > {} + {}".format(jd_ac, jd_ab, jd_bc))
```

1b: Dice Coefficient Stuff

Word Features
wf_1: a, wf_2: b, wf_3: c, wf_4: a b, wf_5: b c, wf_6: a c

Dice Coefficient Core Functions

```python
def qs(string_1, string_2):
  # modify the input strings
  set_1 = set(string_1.split())
  set_2 = set(string_2.split())

  # numerator
  intersection = set_1.intersection(set_2)
  magnitude_of_intersection = len(intersection)
  numerator = 2 * magnitude_of_intersection

  # denominator
  magnitude_of_set_1 = len(set_1)
  magnitude_of_set_2 = len(set_2)
  denominator = magnitude_of_set_1 + magnitude_of_set_2

  # qs_value
  qs_value = float(numerator / denominator)

  # return
  return round(qs_value, 2)


def qs_distance(string_1, string_2):
  return round(float(1 - qs(string_1, string_2)), 2)
```

Triangle inequality does not obtain

Using code similar to that from the Jaccard section, I demonstrated that 18/120 possible permutations did not satisfy the triangle inequality.

```
Empirical demonstration of the triangle inequality failing for the Dice Coefficient
Failure 1: QS("a","b") = 1.0 > 0.66 = 0.33 + 0.33 = QS("a","a b") + QS("a b","b")
Failure 2: QS("a","b c") = 1.0 > 0.83 = 0.33 + 0.5 = QS("a","a b") + QS("a b","b c")
Failure 3: QS("a","c") = 1.0 > 0.66 = 0.33 + 0.33 = QS("a","a c") + QS("a c","c")
Failure 4: QS("a","b c") = 1.0 > 0.83 = 0.33 + 0.5 = QS("a","a c") + QS("a c","b c")
Failure 5: QS("b","a") = 1.0 > 0.66 = 0.33 + 0.33 = QS("b","a b") + QS("a b","a")
Failure 6: QS("b","a c") = 1.0 > 0.83 = 0.33 + 0.5 = QS("b","a b") + QS("a b","a c")
Failure 7: QS("b","c") = 1.0 > 0.66 = 0.33 + 0.33 = QS("b","b c") + QS("b c","c")
Failure 8: QS("b","a c") = 1.0 > 0.83 = 0.33 + 0.5 = QS("b","b c") + QS("b c","a c")
Failure 9: QS("c","b") = 1.0 > 0.66 = 0.33 + 0.33 = QS("c","b c") + QS("b c","b")
Failure 10: QS("c","a b") = 1.0 > 0.83 = 0.33 + 0.5 = QS("c","b c") + QS("b c","a b")
Failure 11: QS("c","a") = 1.0 > 0.66 = 0.33 + 0.33 = QS("c","a c") + QS("a c","a")
Failure 12: QS("c","a b") = 1.0 > 0.83 = 0.33 + 0.5 = QS("c","a c") + QS("a c","a b")
Failure 13: QS("a b","c") = 1.0 > 0.83 = 0.5 + 0.33 = QS("a b","b c") + QS("b c","c")
Failure 14: QS("a b","c") = 1.0 > 0.83 = 0.5 + 0.33 = QS("a b","a c") + QS("a c","c")
Failure 15: QS("b c","a") = 1.0 > 0.83 = 0.5 + 0.33 = QS("b c","a b") + QS("a b","a")
Failure 16: QS("b c","a") = 1.0 > 0.83 = 0.5 + 0.33 = QS("b c","a c") + QS("a c","a")
Failure 17: QS("a c","b") = 1.0 > 0.83 = 0.5 + 0.33 = QS("a c","a b") + QS("a b","b")
Failure 18: QS("a c","b") = 1.0 > 0.83 = 0.5 + 0.33 = QS("a c","b c") + QS("b c","b")
```

Figure 2: Dice Distance Not Satisfying the Triangle Inequality

2a: Cosine Similarity Output

See below for output; discussion will be in 2b.

```
Original: I like the talking heads.
Vector: {'I': 0.89, 'like': 1.58, 'talking': 0.89, 'heads': 0.89, '.': 0.89, 'love': 0, 'enjoy': 0, 'really': 0}
Cosine Similarity: 1.0


Original: I love the talking heads.
Vector: {'I': 0.89, 'like': 0, 'talking': 0.89, 'heads': 0.89, '.': 0.89, 'love': 1.58, 'enjoy': 0, 'really': 0}
Cosine Similarity: 0.56


Original: I enjoy the talking heads.
Vector: {'I': 0.89, 'like': 0, 'talking': 0.89, 'heads': 0.89, '.': 0.89, 'love': 0, 'enjoy': 1.58, 'really': 0}
Cosine Similarity: 0.56


Original: I really like the talking heads.
Vector: {'I': 0.89, 'like': 1.58, 'talking': 0.89, 'heads': 0.89, '.': 0.89, 'love': 0, 'enjoy': 0, 'really': 1.32}
Cosine Similarity: 0.87


Original: I really enjoy the talking heads.
Vector: {'I': 0.89, 'like': 0, 'talking': 0.89, 'heads': 0.89, '.': 0.89, 'love': 0, 'enjoy': 1.58, 'really': 1.32}
Cosine Similarity: 0.49


Original: I really love the talking heads.
Vector: {'I': 0.89, 'like': 0, 'talking': 0.89, 'heads': 0.89, '.': 0.89, 'love': 1.58, 'enjoy': 0, 'really': 1.32}
Cosine Similarity: 0.49
```

Figure 3: Changes in Cosine Similarity

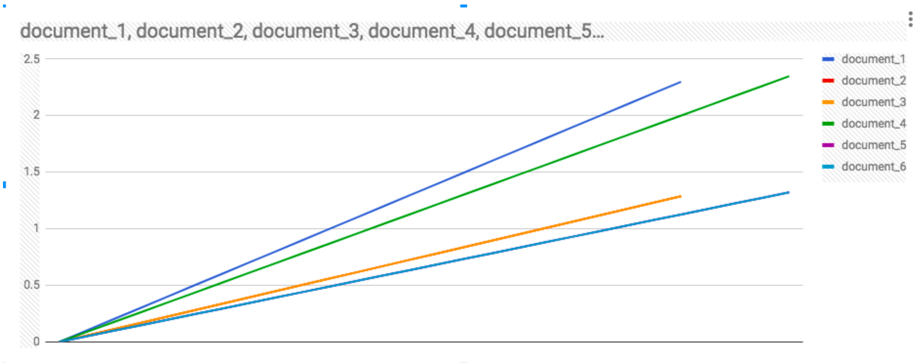


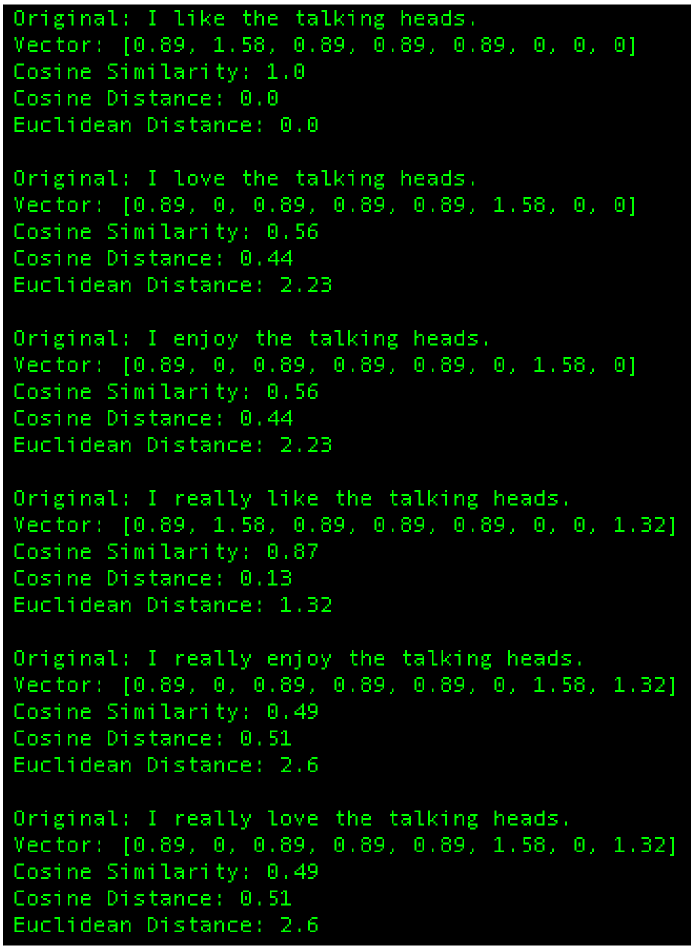
Figure 4: Graph of Cosine Similarities

2b: Cosine Similarity Graph

In my graph the magnitude of each vector is the length and the slope is the cosine between document_1 and the other documents. This representation (and the underlying changes) reflect my intuition that the cosine value measures how similar the documents are to each other in way that relates to the relative importance of each feature; it also has the benefit of showing absolute value of the underlying TF-IDF scores that make the vectors e.g. Document_4 is most similar to Document_1 (Document_1 is a proper subset of Document_4), but Document_4 also has a high TF-IDF score for feature "really", which extends it's magnitude.

2c: Cosine v. Euclidean

I used Cosine Distance and Euclidean Distance packages from SciPy; to generate Cosine Similarity, I subtracted the results of Cosine Distance from 1.
My calculated vales of Cosine Similarity match those calculated by SciPy. There



```
Original: I like the talking heads.
Vector: [0.89, 1.58, 0.89, 0.89, 0.89, 0, 0, 0]
Cosine Similarity: 1.0
Cosine Distance: 0.0
Euclidean Distance: 0.0

Original: I love the talking heads.
Vector: [0.89, 0, 0.89, 0.89, 0.89, 1.58, 0, 0]
Cosine Similarity: 0.56
Cosine Distance: 0.44
Euclidean Distance: 2.23

Original: I enjoy the talking heads.
Vector: [0.89, 0, 0.89, 0.89, 0.89, 0, 1.58, 0]
Cosine Similarity: 0.56
Cosine Distance: 0.44
Euclidean Distance: 2.23

Original: I really like the talking heads.
Vector: [0.89, 1.58, 0.89, 0.89, 0.89, 0, 0, 1.32]
Cosine Similarity: 0.87
Cosine Distance: 0.13
Euclidean Distance: 1.32

Original: I really enjoy the talking heads.
Vector: [0.89, 0, 0.89, 0.89, 0.89, 0, 1.58, 1.32]
Cosine Similarity: 0.49
Cosine Distance: 0.51
Euclidean Distance: 2.6

Original: I really love the talking heads.
Vector: [0.89, 0, 0.89, 0.89, 0.89, 1.58, 0, 1.32]
Cosine Similarity: 0.49
Cosine Distance: 0.51
Euclidean Distance: 2.6
```

Figure 5: Cosine and Euclidean Results

appears to be a generally positive association between the Euclidean Distance and Cosine Distance, which is to be expected; bu,t more importantly, we want these measures to remain distinct: Cosine Similarity/Distance abstracts from the size of the text, whereas Euclidean measures do not. In this regard, Cosine Similarity is measuring similarity of texts based on a normalised input i.e. where absolute frequency of occurrence does not matter; we only care about relative frequency. These techniques have distinct applications and context and intention will suggest which is the more appropriate metric, though there will be some commonality in their respective outputs.

3: Spam

```
normal_tweets = [
  "Robert Webb @arobertwebb This was a top chat with an instinctively great
      interviewer. Big fan of @mrjamesob",
```

```
  "J.K. Rowling @jk_rowling Retweeted Lumos @lumos Violence, coercion, abuse of power.
      Children are trafficked into institutions become vulnerable to modern slavery.
      #antitraffickingday",
  "Janey Godley Retweeted Angry Scotland @AngryScotland Tory MP will miss a
      parliamentary vote on universal credit to run the line at a Champions League
      game instead.",
  "Caroline O. liked Manu Raju @mkraju Conservative blogger Chuck Johnson has been
      asked to turn over docs to Senate Intel over this but he won't cooperate",
  "Robert Web @arobertwebb Retweeted Marcoooos! @marcusbrig The Young Ones On Comic
      Relief THIS...sometimes I need this.
      https://www.youtube.com/watch?v=NhqlrQ64f2Y",
]

spam_list = [
  [0, "Roberto", "Roger"],
  [1, "Webb", "Wes"],
  [2, "@robertowebb", "@rogerailes"],
  [5, "@mugabi", "@marcusaurelius"],
  [16, "https://www.youtube.com/watch?v=ERw-Frq6knI",
      "https://www.youtube.com/watch?v=dTcvmmOkqJI"],
]

def spam_index(integer):
  length = len(list("{0:b}".format(integer - 1))) # size of containters
  spam_indices = list()
  for k in range(integer):
    binary_elements = list("{0:b}".format(k))
    temp_list = [0 for k in range(length - len(binary_elements))]
    for b in binary_elements:
      temp_list.append(int(b))
    spam_indices.append(temp_list)
  return spam_indices

def make_spam(tweet, spam_index, spam_list):
  tweet_list = tweet.split()
  count = 0
  for index in spam_index:
    spam_change = spam_list[count][index + 1]
    tweet_index = spam_list[count][0]
    tweet_list[tweet_index] = spam_change
    count += 1
  return " ".join(tweet_list)
```

Average Levenshtein Score for a Normal Tweet compared to the chosen normal tweet: 132.5. Average Levenshtein Score for a Spam Tweet compared to the chosen normal tweet: 24.0. Ratio of spam_average to normal_average: 0.18.

Spam tweets distinguish themselves from my tweet by having less distance than normal tweets; this is to be expected: spam tweets, as we were asked to construct them, attempt to mimic normal tweets, so by construction they are "close". Each of the normal tweets is relatively quite different, keeping in mind that twitter has a 140 character limit on tweets, so an average distance of 132.5 is non-trivial. This analysis produces similar results for each normal tweet: spam tweets based on a normal tweet are very close to the normal tweet, whereas other normal tweet tend to be further away. Other normal tweets might be clsoer together if they concern the same subject, but as the tweets should be conveying different concepts, intentions, etc. you'd still expect them to have a greater distance than spam tweets.