LATE DAYS USED: 1

Ashwin Mahesh
CS 473
11/12/19

Project 2 Report

Part 1 – Parse the Corpus

The first thing that I did to process this data was parse the XML. To do this, I created a function called XMLParse that reads an inputted file, breaks it up into separate documents, then into individual tags stored in a dictionary. Special fields within the tags such as NEWID were stored in the dictionary as well, however this functionality was added in as I made more progress with the project. I used a stack to know whether or not to create a new document from reading. If an open tag was the next token <*>, and the length of the stack was 0, then a new document was created. I stored all of this parsed data within a class called XMLDoc to easily access it.

Next, I removed all of the documents that did not contain a 'BODY' section from the corpus. The next thing I did was stemming and stopping the BODY section for each of these documents. I used the NLTK package, using the built in stopwords and PorterStemmer. I then made a list of all of the unique words within the corpus.

The next thing I did was calculate the TFIDF. I created a class called TFIDF that took in all of the documents, and the unique word list, and did the calculations for TF, IDF, and TFIDF. Initially I did it using normal arrays, however this took way too long. As a result, I reformatted everything to use NumPy arrays, and this significantly shortened the run time (From ~180 seconds to ~15 seconds). I then used the TFIDF values and NumPy arrays to calculate and store the Cosine Similarities between each of the documents.

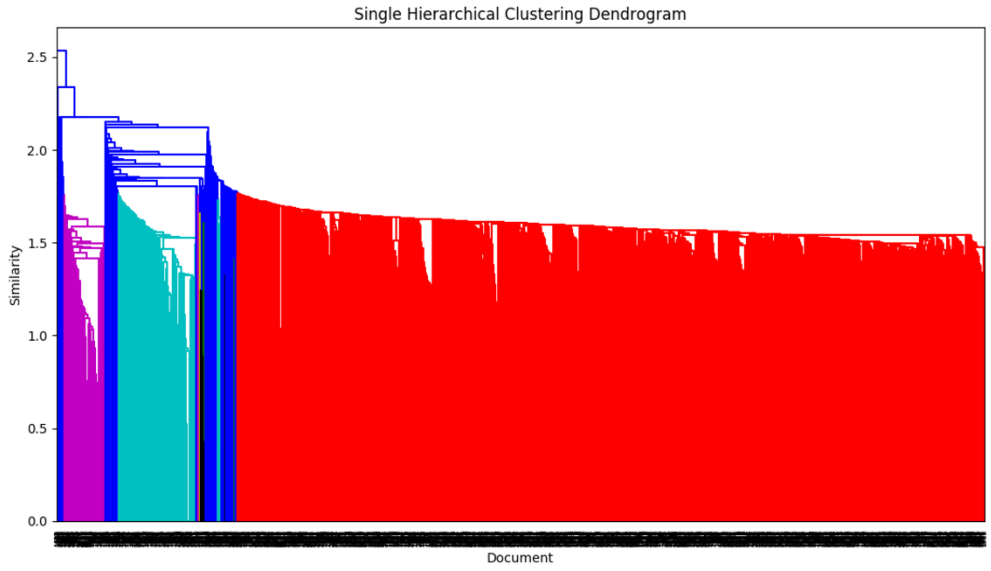
```
#Uses NLTK To Porter Stem and Remove Stopwords
def removeStopwords(text):
  porterStemmer = PorterStemmer()
  stopWords = set(stopwords.words('english'))
  tokenedText = word_tokenize(text)
  newBody=''
  for word in tokenedText:
    if word not in stopWords:
      newBody+=porterStemmer.stem(word)
      newBody+=' '
  return newBody
```

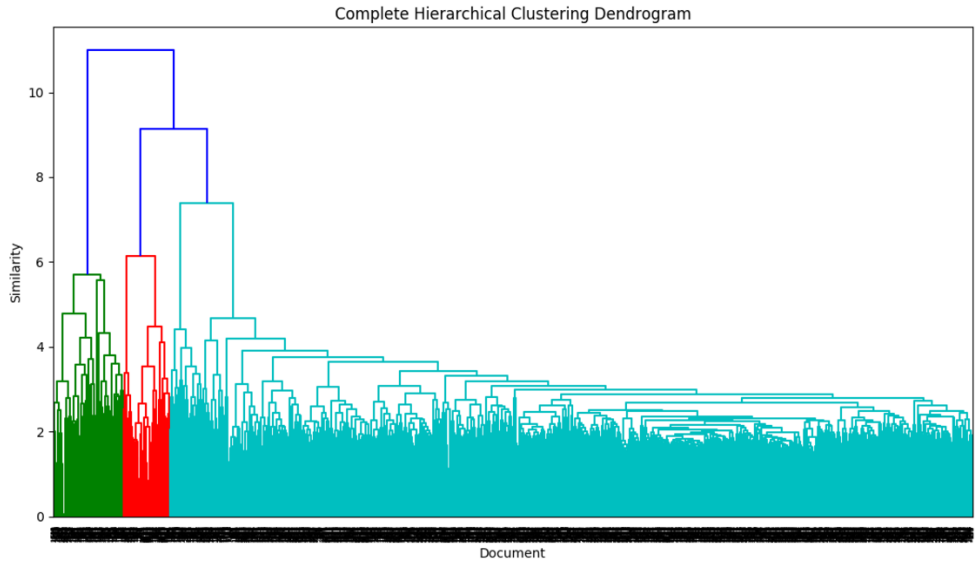
```
def calculateIDF(self, showTime=False):
  startTime = time.time()
  counts = np.zeros(len(self.allTFs[0]))
  for tf in self.allTFs:
    containedIndexes = np.where(tf>0)
    for index in containedIndexes:
      counts[index]+=1

  for i in range(0, len(counts)):
    self.idf[i] = math.log(self.docCount / (counts[i]), 2)
  if showTime:
    print('IDF Function Runtime: ' + str(round(time.time() - startTime, 3)) + ' seconds')
  return self

#Calculates TFIDF Based on Inputted TF and IDF
def calculateTFIDF(self, showTime=False):
  startTime = time.time()
  self.tfidf = np.multiply(self.allTFs, self.idf)
  if showTime:
    print('TFIDF Function Runtime: ' + str(round(time.time() - startTime, 3)) + ' seconds')
  return self

#Calculates the similarity between each of the vectors using the cosine similarity formula
def calculateCosineSimilarity(self):
  dotProd = np.dot(self.tfidf, self.tfidf.T)
  squaredMatrix = np.diag(dotProd)
  inversedSquare = 1/squaredMatrix
  inversedSquare[np.isinf(inversedSquare)] = 0
  inversedUnsquare = np.sqrt(inversedSquare)
  self.similarityMatrix = dotProd * inversedUnsquare
  self.similarityMatrix = self.similarityMatrix.T * inversedUnsquare
```

```
def XMLParse(filePath,  limit=0, ignoreFirstLine=True):
  file = open(filePath)
  if ignoreFirstLine:
    file.readline()
  fileText = file.read()

  index = 0

  tags = []
  output=[]
  currOutputIndex=-1
  token,index=getNextToken(fileText, index)
  currTextVal=''

  hasLimit = False
  if limit>0:
    hasLimit = True

  while(token!='' and index!=-1):
    if(isTag(token)):
      tag = getTagName(token)
      if(isOpenTag(token)):
        tags.append(tag)
        if(len(tags) == 1):
          if(hasLimit and currOutputIndex>=limit):
            break
          output.append(XMLDoc({}))
          currOutputIndex+=1
          specialTags = extractParamsFromTag(token)
          if specialTags!=False:
            for additionalInfo in specialTags:
              output[currOutputIndex].setField(additionalInfo, specialTags[additionalInfo])
      else:
        output[currOutputIndex].setField(tags[len(tags)-1], currTextVal)
        tags.pop()
        currTextVal=''
```

Part 2 – Clustering

From the pictures attached below, we can see that there are significantly more clusters in single hierarchal clustering than in the complete hierarchal clustering. It is so dense that you can barely see individual clusters using the single method. Additionally, there are a lot more clusters at the higher level in single than in complete. At this size, we are able to count the number of high level clusters for complete with just the bare eye, however there are too many high level clusters in single to be able to do this. It appears that the similarity of the vast majority of clusters in single and complete have the around the same cosine similarity value. Until we get to the higher levels, where complete clusters have significantly higher similarities with the cluster they are combining with





Part 3 – Evaluation

1. The way my evaluation metric works is this. First, remove any clusters that contain more than 50% of the documents. For every cluster within a clustering linkage, find the total counts of each of the topics given the documents at this cluster. Take the count of the highest topic, divide it by the count of the 2nd highest topic, then multiply this number by the log2(total number of the documents in the cluster)+1. This is the score for this one cluster. The final score for the clustering method is the sum of the scores of all of the clusters, divided by the total number of documents as a normalizing factor so that this number doesn't grow too high with more documents.

```
#Calculates the similarity at each cluster by using the topics
#Needs input from getTopicCountByCluster Function
#Maximum Topic Count/ 2nd highest topic Count * [log2(Total Number of documents in cluster)+1]
def calculateSimilarityAtOneCluster(cluster, numDocsInCluster):
  maxKey, maxCount, secondMaxKey, secondMaxCount = getMaxAndSecondMaxTopics(cluster)
  secondMaxCount = 1 if secondMaxCount==0 else secondMaxCount
  return maxCount * (math.log(numDocsInCluster, 2) +1) / secondMaxCount

#Sums up the similarities for all the clusters then divides by some normalizing factor?
def evaluate(clusteringMethod):
  clustersWithDocuments, docCount = getDocsByCluster(clusteringMethod)
  removedHigherLevelClusters = cleanupClusters(clustersWithDocuments, 0.5, docCount)

  totalSimilarity=0
  for clusterID in removedHigherLevelClusters:
    topicForCluster = getTopicCountByCluster(removedHigherLevelClusters[clusterID])
    totalSimilarity+=calculateSimilarityAtOneCluster(topicForCluster, len(removedHigherLevelClusters[clusterID]))

  return totalSimilarity/docCount
```

2. **Mathematical Definition**

$$\boldsymbol{score} = \left(\sum_{c \epsilon C} \frac{argmax(topic.count)}{arg2ndMax(topic.count)} * [log_2(c.doc\_count) + 1]\right)/(C.doc\_count)$$

3. **Explanation**

   In terms of data cleaning, I removed all of the clusters that contained more than 50% of the documents because I felt that they would skew the data. Since we were told that the we needed to compute a value based on the topics within each cluster, I knew that the topic with the highest count at each cluster needed to be involved. If a cluster was good, it would contain a higher number of one topic, and a lot less of every other topic. A good indicator of how diverse a cluster is the second highest value. Since it is on the denominator of my metric, if it is low, then the max count topic will be worth more. If it is high, then the max count topic will be worth less. This value is also multiplied by the log of the document count of each cluster in order to give clusters with more documents more weight. This way, a cluster with a good ratio and more documents will mean more. Everything is divided by the total document count at the end as a normalizing factor.

4. **Linkage Scores**

```
Single Linkage Score: 6.215099299071471
Complete Linkage Score: 7.1878839237341445
```

While using the whole given corpus, these are the scores that were calculated using my evaluation metric.

Single Linkage Score = 6.2151
Complete Linkage Score = 7.1879

5. **Discuss Differences**

Based on this calculation, for close to 3000 documents, Complete Linkage scored higher. In terms of run time, they were both about the same.

```
Creating and cutting single link clusters...
Time: 47.715 seconds
Creating list of single link clusters each document is contained in...
Time: 0.162 seconds
Writing single link clusters to file...
Time: 1.302 seconds
Creating and cutting complete link clusters...
Time: 45.508 seconds
```

I tested again using just 500 documents this time, and complete linkage again had the higher score. However, once it gets lower than this, the scores are nearly identical. At 100 documents, they are exactly the same.

I played with one other parameter, and that was the cut off proportion of documents each cluster could have. When I ran it with 3000 and a cutoff of 40%, then Single Linkage clustering won with a higher score.

```
Evaluating Single and Complete Link Clustering...
Single Linkage Score: 5.919956285612209
Complete Linkage Score: 5.9149040829740755
Time: 1.334 seconds
```