# Assignment 1 - Beyond word count

Anirudh Narasimhamurthy(u0941400) and Soumya Smruti Mishra(u0926085)

September 8, 2015

## 1 Word Transition counts

In this part of the assignment we extend the basic word count example to calculate count of word pairs or 2-grams or bigrams. The data structure used to compute the number of times word2 follows word1 in the given file was implemented using **tuples** in Python.

The input file had a lot of punctuations and leading and trailing spaces. If we consider the input file as such and pass it to the sc.textFile() method, the count of the most commonly occurring word pairs has variations of the same word pair with differences in punctuation. A brief summary of our approach to find the most common word pairs in the document is provided below:

- Use the textFile() method to create a RDD of input file as lines.
- Map the lines to individual words by using an appropriate lambda function inside map()
- Create tuples of word pairs by applying flatMap() over the mapped wordsRDD
- Filter the tuples or word-pairs with word length less than 5 characters using the filter function
- Apply map() and reduceByKey() over the filteredRDD to obtain the appropriate word pair counts.
- Apply the takeOrdered() function on the resulting RDD from the previous step and provide the appropriate parameter values to obtain the ten most frequently occurring word pairs.

The results after running our program on the input file are tabulated in Table 1:

Clearly the results in the top 10 contain a variation of the same word pair in 3 different places owing to the punctuation. So we also tweaked our code to remove all the punctuation, leading and trailing spaces and convert the entire text to lowercase. The remove punctuation was done immediately after loading the document using sc.textFile and so the filtered/cleaned up text document was then passed to the rest of the functions.

On running the input doc through the punctuation remover, the results of the 10 frequently occurring word pairs are tabulated in Table 2 :

**Ten most frequently occurring word pairs (len(word) > 4)**

| Word Pair | Count |
|---|---|
| **(u'Prince', u'Andrew')** | 631 |
| **(u'United', u'States')** | 229 |
| **(u'Prince', u'Andrew,')** | 163 |
| **(u'Prince', u'Vasili')** | 140 |
| **(u'Prince', u'Andrew.')** | 97 |
| **(u'Project', u'Gutenberg-tm')** | 86 |
| **(u'Prince', u"Andrew's")** | 76 |
| **(u'United', u'States,')** | 68 |
| **(u'takes', u'place')** | 67 |
| **(u'Project', u'Gutenberg')** | 66 |

Table 1: Top 10 most frequently occurring word pairs for input document

| Word Pair | Count |
|---|---|
| **(u'prince', u'andrew')** | 907 |
| **(u'united', u'states')** | 392 |
| **(u'prince', u'vasili')** | 178 |
| **(u'project', u'gutenbergtm')** | 104 |
| **(u'project', u'gutenberg')** | 99 |
| **(u'sherlock', u'holmes')** | 99 |
| **(u'mademoiselle', u'bourienne'):** | 91 |
| **(u'takes', u'place')** | 90 |
| **(u'prince', u'andrews'):** | 79 |
| **(u'marya', u'dmitrievna')** | 76 |

Table 2: Top 10 most frequently occurring word pairs for after removing punctuation and trailing spaces in input document

## 2 Text similarity

**Part A : Jaccard Distance/Similarity**

For computing the Jaccard distance across all pairs of documents, we first need to load all the files in the directory and we used sc.wholeTextFiles() method for this.

wholeTextFiles() method considers the newline as a character and not a separate line. Hence we had to use mapValues() to make sure newline was converted to a space and then we do a split based on space to get the list of words in the document.

- We also applied a filter to make sure we were considering words which had a length of at least 4 characters.
- We then do a cartesian product of the key value pairs and this will give us all possible combinations of one document with other. The (key,value) pair in this case would be (filename, words in file as a list). Cartesian product would give us 451 x 451 combinations.
- Similar to the wordcount example, we now map a file and its jaccard similarity with another file as a tuple. Jaccard similarity function takes in the input list and then constructs a set of words from the two files and then computes the similarity as per the

standard expression (len(no of common words in two documents)/len(total no of words in two documents))

- We then do a reduction by key where key is the filename and construct a list of length 451 which consists of jaccard similarity values of one document with respect to all the other documents in the directory.
- We do this for all the files and store the values in a list.

The diagnol entries of the resultant matrix/list would be 1 and the matrix would be a symmetric matrix since similarity between ai and ak is same as ak and ai.

We were able to run our code on all the 451 documents and obtain the Jaccard Similarity matrix. But plotting using matplotlib was taking or running for a really long time. Since matplotlib expects the input to be in a numpy array, we had to filter and transfer the results from the RDD into individual numpy arrays.

Observation: Jaccard similarity says the documents are not very similar and most of the plot lies in the red region which depicts the lower range of similarity.

A plot of Jaccard similarity between the different documents is shown below :

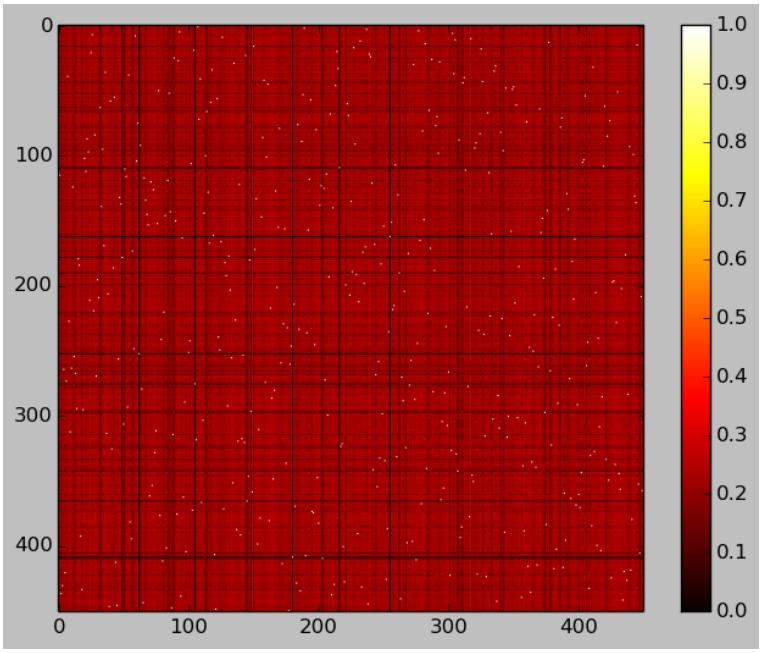


Figure 1: Jaccard Matrix plotted as an Image

## Part B : Cosine Distance/Similarity

To compute the Cosine Distance, we first found the 1000 most common words across the set of all documents based on the word count. After loading the files and getting the words in each document via map() functions, we then had a flatMap() function to get a list of all words across all the files.

- The resulting RDD was passed to the wordcount function similar to that in Question 1 and we then sorted the count in descending order and took the top 1000 words from it.

The most common 1000 words were used as the basis for building up the individual feature vectors where individual entries in the the feature vector represents the number of times the word in the common list appears in the individual document.

- If the word in the common list did not occur in the individual documents, then its corresponding value in the vector was set as 0.
- We again used the **cartesian** function to obtain all pair of documents similarity and processed and stored the results in a similar fashion to the Jaccard similarity results.

Observation: Cosine similarity says the documents are similar, since we used only 1000 top words to build the matrix and most of the plot lies in the yellow region which depicts the higher range of similarity.

A plot of cosine similarity between the different documents is shown below :

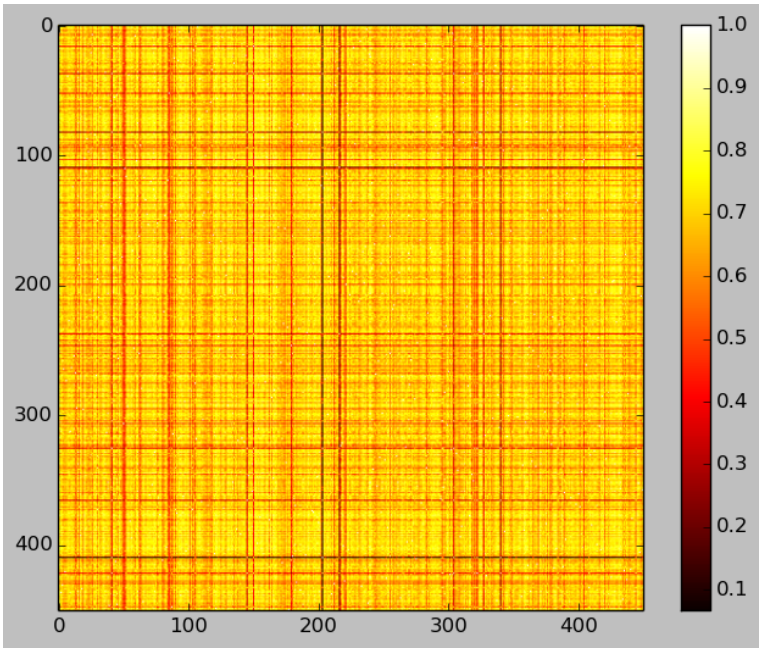


Figure 2: Cosine Matrix plotted as an Image

Text file containing the similarity matrices has also been added to the zipped project folder submitted with this assignment. Named:-

Jaccard Similarity Matrix.csv/jaccard.txt
Cosine Similarity Matrix.csv/cosine.txt