**Plagiarism Checker**

Dhanush Bommavaram

**Problem Statement:**

Blogs are the richest source of information. However, much of the text information is redundant. Tech Mahindra takes up the job of designing a blog retrieval and summarization system where the goal is to retrieve the K least redundant blogs while returning the top-K search results.

You are a software developer at Tech Mahindra. Your Team is assigned the task of designing the blog search tool. To find the top K results, the team leader decides to use the plagiarism score as a metric for evaluation. You are being asked to design a prototype for this tool using basic text processing techniques like edit distance and n-gram model considering the processing speed. Write a program for designing the plagiarism check tool.

**Proposed Solution:**

Place all the blogs as .txt files. Place all the file names in the blogNames list in the blog\_ranker.py python file.

We take each file and calculate it’s plagiarism score with all the other files(we store text from all the files in a single list). We rank the files based on plagiarism score. The blog with the least plagiarism score is ranked the highest.

The plagiarism score is nothing but similarity score. It is calculated by combining Jaccard similarity and edit distance.

We use a 1 vs all approach for checking plagiarism.

Plagiarism scores vary between 0 and 1.

Jaccard similarity:

Jaccard Similarity is a common proximity measurement used to compute the similarity between two objects, such as two text documents. The Jaccard similarity is calculated by dividing the number of observations in both sets by the number of observations in either set. In other words, the Jaccard similarity can be computed as the size of the intersection divided by the size of the union of two sets.

Edit distance:

Minimum number of operations required to convert one string to another string. You have the following three operations permitted on a word:

1. Insert a character.
2. Delete a character.
3. Replace a character.

**Code documentation:**

The list blogNames contains all the names of blogs.

We take input from the above named files. We normalise it by converting all the characters into lowercase and removing all punctuation. We store all the words in a list separated by whitespace. To make computation faster we store the starting index of each file in the allBlogsArray so that we can exclude that file when calculating similarity of that particular file.

blogNames=["t.txt","x.txt","t1.txt"]

allBlogsArray=[]

blogStartIndex=[]

s2TrigramStartIndex=[]

s2len=[]

textInBlogArray=[]

s2Trigrams=list()

for blog in blogNames:

    file=io.open(blog, mode="r", encoding="utf-8")

    textInBlog=file.read()

    textInBlog=textInBlog.lower()

    textInBlog = textInBlog.translate(str.maketrans('', '', string.punctuation))

    textArray = textInBlog.split()

    textInBlogArray.append(textArray)

    blogStartIndex.append(len(allBlogsArray))

    allBlogsArray+=textArray

    s2TrigramStartIndex.append(len(s2Trigrams))

    temtri=trigrams(textArray)

    s2Trigrams+=temtri

    s2len.append(len(temtri))

    file.close()

trigrams function returns a list of all trigrams, bigrams, and unigrams.

def trigrams(textArray):

    trigram = []

    for i in range(len(textArray)-3+1):

        trigram.append(textArray[i:i+3])

    for i in range(len(textArray)-2+1):

        trigram.append(textArray[i:i+2])

    for i in range(len(textArray)):

        trigram.append(textArray[i])

    return trigram

The following function returns the Jaccard Similarity by taking trigrams for lists s1 and s2. List s1 is the list of trigrams for the files whose score is being calculated and s2 is the list that contains trigrams of all the files.

def jaccardSimilarity(s1Trigrams,s2Trigrams,s2TrigramIndex,s2len):

    s2Trigrams=s2Trigrams[:s2TrigramIndex] + s2Trigrams[s2TrigramIndex+s2len:]

    s1Trigrams = set(map(tuple, [ngram for ngram in s1Trigrams]))

    s2Trigrams = set(map(tuple, [ngram for ngram in s2Trigrams]))

    intersectionlen = len(s1Trigrams.intersection(s2Trigrams))

    unionlen = len(s1Trigrams.union(s2Trigrams))

    # intersectionlen = len(intersection(s2Trigrams,s1Trigrams))

    # unionlen = len(s2Trigrams)

    return (intersectionlen)/(unionlen)

The following functions returns a score after combining Jaccard and Edit distance scores.

def similarity(s1, s2, s2Trigrams, exclude\_index, exclude\_len,s2TrigramIndex,s2len):

    s1Trigrams = trigrams(s1)

    jaccard = jaccardSimilarity(s1Trigrams,s2Trigrams,s2TrigramIndex,s2len)

    s2 = s2[:exclude\_index] + s2[exclude\_index+exclude\_len:]

    distance = [[float("inf") for j in range(len(s2)+1)] for i in range(len(s1)+1)]

    for i in range(len(s1) + 1):

        distance[i][len(s2)] = len(s1)-i

    for j in range(len(s2) + 1):

        distance[len(s1)][j] = len(s2)-j

    for i in range(len(s1)-1,-1,-1):

        for j in range(len(s2)-1,-1,-1):

            if s1[i] == s2[j]:

                distance[i][j]=distance[i+1][j+1]

            else:

                distance[i][j] = 1+min(distance[i+1][j], distance[i][j+1], distance[i+1][j+1])

    editDistanceScore=(1-distance[0][0]/max(len(s1),len(s2)))

    # print(jaccard,editDistanceScore)

    alpha=0.5

    similarityScore = alpha\*jaccard + (1-alpha)\*editDistanceScore

    return similarityScore

blogScores contains blog names along with their similarity score. We calculate similarity for each file and sort then based on these values and print all the names based on scores.

blogScores=[[0 for j in range(2)] for i in range(len(textInBlogArray))]

for j,i in enumerate(textInBlogArray):

    similarityScore=similarity(i,allBlogsArray,s2Trigrams,blogStartIndex[j],len(i),s2TrigramStartIndex[j],s2len[j])

    blogScores[j][0]=blogNames[j]

    blogScores[j][1]=similarityScore

blogScores.sort(key = lambda x: x[1])

print(blogScores)