**Information Retrieval**

**CS 5615**

**Programming Project**

Reg No: 179318T

MSF.Fayaza

**Brief description of algorithms**

**Jaccard coefficient**

Jaccard coefficient compares members for two sets to see which members are shared and which are distinct. It’s a measure of similarity for the two sets of data, with a range from 0% to 100%. The higher the percentage, the more similar the two populations. Simply known as, intersection over union. Let A and B be two sets; then the J.C. is Equals 1 when A and B have the same elements and zero when they are disjoint.

Measuring the Jaccard similarity coefficient between two data sets is the result of division between the number of features that are common to all divided by the number of properties.

**Levenshtein distance**

Levenshtein distance (LD) is a measure of the similarity between two strings, which we will refer to as the source string (s) and the target string (t). The distance is the number of deletions, insertions, or substitutions required to transform s into t. For example,

* If s is "test" and t is "test", then LD(s,t) = 0, because no transformations are needed. The strings are already identical.
* If s is "test" and t is "tent", then LD(s,t) = 1, because one substitution (change "s" to "n") is sufficient to transform s into t.

The greater the Levenshtein distance, the more different the strings are.

**Jaro distance**

The Jaro algorithm is commonly used for name matching in data linkage systems. It accounts for insertions, deletions and transpositions. The algorithm calculates the number m of common characters (agreeing characters that are within half the length of the longer string) and the number of transpositions t. A similarity dj{\displaystyle d\_{j}} of two given strings s1{\displaystyle s\_{1}} and s2{\displaystyle s\_{2}} is

{\displaystyle d\_{j}=\left\{{\begin{array}{l l}0&{\text{if }}m=0\\{\frac {1}{3}}\left({\frac {m}{|s\_{1}|}}+{\frac {m}{|s\_{2}|}}+{\frac {m-t}{m}}\right)&{\text{otherwise}}\end{array}}\right.}Where: {\displaystyle m}m is the number of matchingcharacters; {\displaystyle t}t is half the number of transpositions.

**Cosine similarity with TF-IDF**

A commonly used term weighting method is tf-idf, which assigns a high weight to a term, if it occurs frequently in the document but rarely in the whole document collection. Contrarily, a term that occurs in nearly all documents has hardly any discriminative power and is given low weights, which is usually true for stopwords like determiners, but also for collection-specific terms.

For calculating the tf-idf weight of a term in a particular document, it is necessary to know two things:

* How often does it occur in the document (term frequency = tf),
* How many documents of the collection does it appear (document frequency = df).

Take the inverse of the document frequency (= idf) and you have both components to calculate the weight by multiplying tf by idf. In practice, the inverse document frequency is calculated as the logarithm (base 10) of the quotient of the total number of documents (N) and the document frequency in order to scale the values.

A common way to compute the similarity of two documents is using the cosine similarity measure. The inner product of the two vectors (sum of the pairwise multiplied elements) is divided by the product of their vector lengths. This has the effect that the vectors are normalized to unit length and only the angle, more precisely the cosine of the angle, between the vectors accounts for their similarity.

**Brief description of the implementation of each algorithm**

**Jaccard coefficient**

File name: JaccardC.java

To calculate jaccard coefficient initially I tokenize the sentences and computed as the number of shared terms over the number of all unique terms in both strings (Line 39).

Using whitespace tokenizes the sentences and using arraylist store the tokens. I used the HashSet to store the unique and shared tokens/words and from that calculate the jaccard coefficient. Threshold value for the similarity is 0.5. If similarity between two sentences is grater or equal to threshold it is consider as true and otherwise it is false. Threshold value used determines the similarity. The threshold value is defined running multiple tests on training dataset.

**Levenshtein distance**

Filename: Levenshtein .java

Calculated the minimum number of edited needed to transform one sentence to another (Line 35). Threshold value for the similarity is 30 if the number of character needs to transform more than 30 then it’s considered as two sentences are not similar. Threshold value used determines the similarity. The threshold value is defined running multiple tests on training dataset.

**Jaro distance**

Filename: JaroWinkler.java

First identify the matching correctors in the sentences (line: 78, line 104). To identify the matching correctors, characters in the small sentence is compere with compare with characters in the long sentence (line 90). Matching correctors in the sentences save in separate arrays and number of transpositions needed is calculated (line 117). Using this values jaro distance calculated (line 53). The threshold for the Jaro distance is 0.64. Threshold value used determines the similarity. The threshold value is defined running multiple tests on training dataset.

**Cosine similarity with TF-IDF**

Filename: CosineDistance.java

First tokenize the sentence using whitespaces (Line 49). After that, calculated the idf values for terms in the documents. (Line 110). After that calculate the tf\* idf value (Line 169). After that calculate the normalized value (Line 192). Using those values calculated the cosine similarity (Line 201). The threshold for the Cosine similarity is 0.51. Threshold value used determines the similarity. The threshold value is defined running multiple tests on training dataset.

**Evaluation matrices implementation**

For the implementation single sentence consider as a document. Using different algorithms similarity between the documents calculated and using threshold values whether documents are similar or not similar is identified. If true documents are similar and false documents are not similar. The result from the algorithms are compared with given labels and TP(RelevantRetrieved), TN(NotRelevantNotRetrieved) ,FN(RelevantNotRetrieved), FP(NotRelevantRetrieved) are calculated (File Name: ReadFile.java Line 92). Using those values accuracy, precision, recall, and F1 score calculated.

Accuracy: (TP + TN)/ (TP + TN + FP + FN) (Measures.java Line 84).

Precision: TP/ (TP + FP) (Measures.java Line 95).

Recall: TP/ (TP + FN) (Measures.java Line 104).

F1 score: = (Measures.java Line 113).

**Evaluation results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Accuracy | Precision | Recall | F1 score |
| Jaccard coefficient | 0.8 | 1.0 | 0.5 | 0.67 |
| Levenshtein distance | 0.6 | 0.5 | 0.33 | 0.4 |
| Jaro distance | 0.73 | 0.75 | 0.5 | 0.6 |
| Cosine similarity with TF-IDF | 0.8 | 1.0 | 0.5 | 0.67 |

Above table shows the performance of each algorithm and presented the similarity coefficient results of the performance measurement using Accuracy, Precision, Recall, and F-measure. Here Jaccard coefficient and Cosine similarity with TF-IDF performed well then others.

**Spell checker implementation**

File name: FileSpellChecker.java

Spell checker used is Jazzy Spell Checker. Jazzy is an open source Spell Checker.

First Downloadjazzy-core-0.5.2.jar and added to the project. After that dictionary text created the file contains the list of English words (<http://introcs.cs.princeton.edu/java/data/words.utf-8.txt>).

Call spell checker to test (File name: ReadFile.java line 46).

First tokenize the sentence and get the terms. (File name: FileSpellChecker.java line 82). And check the spell of each token and get the misspell words (File name: FileSpellChecker.java line 83). Get the suggestions for misspell word (File name: FileSpellChecker.java line 89). Select the best suggestion and replace with it (File name: FileSpellChecker.java line 72, 73).

**Evaluation results after Spell checker use**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Accuracy | Precision | Recall | F1 score |
| Jaccard coefficient | 0.8 | 1.0 | 0.5 | 0.67 |
| Levenshtein distance | 0.6 | 0.5 | 0.33 | 0.4 |
| Jaro distance | 0.67 | 0.6 | 0.5 | 0.55 |
| Cosine similarity with TF-IDF | 0.8 | 1.0 | 0.5 | 0.67 |

After the usage of spell checker also Jaccard coefficient and Cosine similarity with TF-IDF performed well then others. Also all the algorithms have same Accuracy, Precision, Recall, and F1 score after the usage of spell checker also. So for this dataset spell checker doesn’t improve the performances.

**Stemmer**

File name: Stemmer.java

Implementing the Porter Stemming Algorithm. The Stemmer class transforms a word into its root form. After the using stemmer write the new file with root terms of document/sentence (File name: Stemmer.java, Line 93).

**Evaluation results after Stemmer**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Accuracy | Precision | Recall | F1 score |
| Jaccard coefficient | 0.8 | 1.0 | 0.5 | 0.67 |
| Levenshtein distance | 0.53 | 0.44 | 0.67 | 0.53 |
| Jaro distance | 0.6 | 0.5 | 0.83 | 0.62 |
| Cosine similarity with TF-IDF | 0.87 | 1.0 | 0.67 | 0.8 |

After using stemmer Cosine similarity with TF-IDF has high accuracy, Cosine similarity with TF-IDF and Jaccard coefficient have high precision, and Jaro distance have high recall and Cosine similarity with TF-IDF has high F1 score. And Cosine similarity with TF-IDF perfomce well then others.

**Lemmatizer**

File name: StanfordLemmatizer.java

I used the Stanford CoreNLP library for lemmatizer. It gives the base form of terms/words. . CoreNLP is designed to be highly flexible and extensible. With a single option can change which tools should be enabled and which should be disabled.

**Evaluation results after using Lemmatizer**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Accuracy | Precision | Recall | F1 score |
| Jaccard coefficient | 0.8 | 1.0 | 0.5 | 0.67 |
| Levenshtein distance | 0.6 | 0.5 | 0.33 | 0.4 |
| Jaro distance | 0.67 | 0.6 | 0.5 | 0.55 |
| Cosine similarity with TF-IDF | 0.8 | 1.0 | 0.5 | 0.67 |

Here Jaccard coefficient and Cosine similarity with TF-IDF performed well then others. Also all the algorithms have same Accuracy, Precision, Recall, and F1 score after the usage of lemmatizer also. So for this dataset lemmatizer doesn’t improve the performances.

**Reference**

1. Courtney Corley and Rada Mihalcea. Measuring the Semantic Similarity of Texts
2. Suphakit Niwattanakul\*, Jatsada Singthongchai, Ekkachai Naenudorn and Supachanun Wanapu. Using of Jaccard Coefficient for Keywords Similarity
3. Wael H. Gomaa, Aly A. Fahmy. A Survey of Text Similarity Approaches
4. Peter Christen. A Comparison of Personal Name Matching: Techniques and Practical Issues
5. Yuhua Li, Zuhair Bandar, David McLean and James O’Shea. A Method for Measuring Sentence Similarity and its Application to Conversational Agents