TEXT PREPROCESSING AND COMPARATIVE STUDY OF COSINE SIMILARITY AND ITS INTEGRATION WITH TF-IDF

[1]Parag Shah, [2]Pavitra Saxena, [3]Hardik Parnami, [4]Kavita Namdeo, [5]Ritesh Khedekar
[1,2,3]Student and [4]Sr. Asst Professor, [5]Asst Professor
[1,2,3,4,5]Department of Computer Science and Engineering
[1,2,3,4,5]Acropolis Institute of Technology and Research, Indore-453771, Madhya Pradesh

ABSTRACT:

Mammoth pile of text has much in common, text matching will reduce the redundancy and the importance of the related text remains intact. Human collected data is mostly in Natural Language. Natural language pre-processing is widely used in artificial intelligence projects and in text mining for information retrieval systems. The need of text pre-processing made the similarity algorithm much faster using a systematic NLTK libraries. The matching approaches developed a new mathematical formulation such as dynamic programming, vector dot product, term frequency and their logarithmic corresponding in the areas of text similarity algorithms. Other than mathematical approach vector form of text can be processed by layered neural net.

Keywords – text pre-processing, semantic similarity, lexical similarity, Natural Language processing, Natural Language Toolkit, Tokenization, Stemming, lemmatization, stop-words, Levenshtein distance, Cosine similarity, TF, TF-IDF, Word2vec

I. INTRODUCTION

Text matching is extensively used technique for solving semantic problems such as text mining, natural language processing. Text, considered as a sentence to a huge paragraph, consists of words such as alphanumeric character, special symbols which are considered as a natural language.

Natural language processing is widely used in Artificial intelligence, speech recognition, A.I Bots, Information retrieval systems. Natural language processing is used to manipulate natural language (human understood) and gives machine the ability to read. **Natural language processing** (NLP) is powerful technique which can be done using **Natural Language toolkit** (NLTK) library of python.

In the normal human language there are many word which might be irrelevant for machine comprehension, so the text undergoes many filtrations. NLTK features many functions for pre-processing such as tokenization, stop-word elimination, stemming, lemmatization, reg-ex filter and more.

A wide range of text matching algorithm were introduced from late 90's from dynamic programming to machine learning. Text similarity has to determine how 'close' two pieces of text are both in surface closeness [lexical similarity] and meaning [semantic similarity]. These algorithms include Levenshtein distance, Cosine Similarity, Fuzzy Logic, Jaccard Similarity, Euclidean distance etc. All of the mentioned above algorithms are used for measuring similarity between two given sentences. But do we have any winning strategy for which is the best algorithm to use?

No, there are many ways to compute the features that capture the semantics or essence of documents and multiple algorithms to capture dependency structure of documents, so that the focus remains on meanings of documents. Talking about all the algorithms is beyond the scope of this research paper. We will talk about algorithms named as Levenshtein distance, Cosine Similarity, Word2Vec. Machine Learning algorithms and almost all Deep Learning Architectures are incapable of processing strings or plain *text* in their raw form, So Word Embedding is done i.e., the texts converted into numbers and there may be different numerical representations of the same text. Word Embedding can be done in two ways [1] Frequency based Embedding [2] Prediction based Embedding. As TF-IDF is example of frequency based embedding. Word2vec method were prediction based in the sense that they provided probabilities to the words and proved to be like word analogies and word similarities. Unlike most of the previously used neural network architectures for learning word vectors, models don't involve dense matrix multiplications. This makes the training extremely efficient: an optimized single-machine implementation can train on more than 100 billion words in one day.

II. Literature Survey

Every algorithm needs text, a preprocessed text, then how text preprocesses? Using nltk a much useful data can be generated.

Tokenization

Tokenization is a step which splits longer strings of text into smaller pieces, or tokens. Larger chunks of text can be tokenized into sentences; sentences can be tokenized into words. Tokenization is not like split function, but in tokenization tokens are created using different approach such as, sent_tokenize (tokenize sentences from paragraph), word_tokenize (tokenize words from sentences), RegexpTokenizer (tokenize the sentences to word using regular expression) which can be useful for sentences having special symbols, TreebankWordTokenizer (tokenize words on the basis of dictionary meaning) is very efficient over others. All the tokenizer belongs to nltk library of python.

Normalization

Normalization generally refers to a series of related tasks meant to put all text on same level that is text to be processed by stages of filtration such as Stemming, lemmatization and elimination of stop words.

Stemming

Stemming is the process of eliminating affixes (suffixed, prefixes, infixes, circumfixes) from a word in order to obtain a word stem. For example, obtaining "dance" from "dancing".

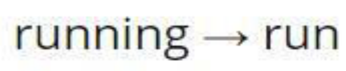

running → run

Figure 1: Stemming example

Lemmatization

Lemmatization is related to stemming, differing in that lemmatization is able to capture canonical forms based on a word's lemma. For example, stemming the word "nice" would fail to return its citation form (another word for lemma) i.e., derives the synonyms of the word.

better → good

Figure 2: Lemmatization example

Elimination of Stop words

Stop words are those words which contribute little to overall meaning the sentences, stop words are generally the most common words in a language. For example, a, the, for, and, of etc.

~~The~~ quick brown fox jumps over ~~the~~ lazy dog.

Figure 3: Eliminate stop word

Levenshtein Distance

The Levenshtein distance between two strings a, b(of length |a| and |b| respectively) is given by $\operatorname{lev}_{a,b}(|a|,|b|)$

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Figure 4: Levenshtein distance formula

where is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise, and $\operatorname{lev}_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b, i and j are 1-based indices. Note that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same. Levenshtein distance is said to be number of single edits that is required in a word to be like the other. For example, the Levenshtein distance between the word “give” and “take” is 3. Also the distance between the word “Danger” and “Dangerous” is 3. It can be observed that the meaning of “Danger” and “Dangerous” is far similar than “give” and “take” but their Levenshtein distance is same.

Cosine Similarity

This approach is used to find the similarity between two sentences irrespective of their sizes. Generally, in text matching approaches is based on matching the maximum number of common words between the documents, but this approach inherits flaws, i.e. as the size of document increases the number of common words increases, the matching deviates from actual topic. So cosine similarity helps in overcoming the flaw. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space.

Cosine similarity derives as:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Figure 5: Cosine similarity formula

The cosine of 0° is 1, and it is less than 1 for any angle in the interval (0, π] radians, it is thus a judgment of orientation(angle).

When plotted on a multi-dimensional space, where each dimension corresponds to a word in the document, the cosine similarity captures the angle of the documents and not the magnitude as computed in Euclidean distance.

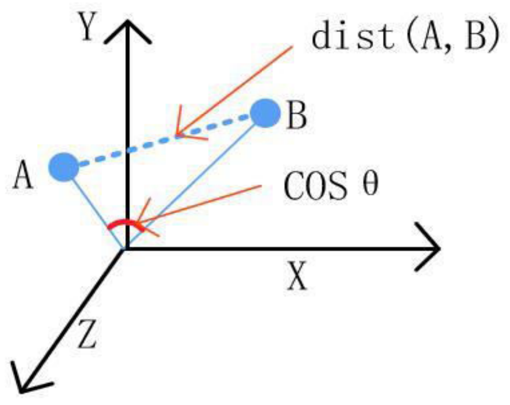


Figure 6: Graphical Representation

The above figure elucidates that the dist (A, B) is the magnitude which is the result of euclidean distance while the angle(θ) between the “A” and “B” is representing cosine angle i.e. cosθ.

Term Frequency (TF)

The number of times a word appears in a document divided by the total number of words in the document. Every document has its own term frequency.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

Figure 7: Term Frequency formula

Inverse Document Frequency (IDF)

The log of the number of documents divided by the number of documents that contain the word ***w***. Inverse data frequency determines the weight of rare words across all documents in the corpus.

$$idf(w) = log(\frac{N}{df_t})$$

Figure 8: Inverse Document Frequency formula

III. Research Methodology

Now consider an example, given below are four general statements:

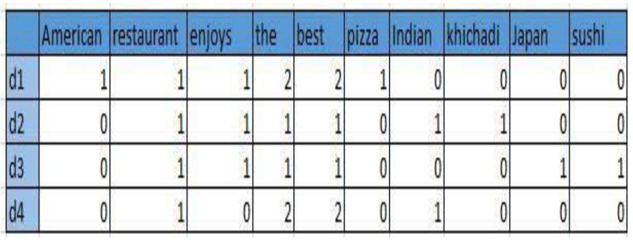
d1: the best American restaurant enjoys the best pizza

d2: Indian restaurant enjoys the best khichadi.

d3: japan restaurant enjoys the best sushi.

d4: the best the best Indian restaurant.

Observing the occurrence in the following table.



	American	restaurant	enjoys	the	best	pizza	Indian	khichadi	Japan	sushi
d1	1	1	1	2	2	1	0	0	0	0
d2	0	1	1	1	1	0	1	1	0	0
d3	0	1	1	1	1	0	0	0	1	1
d4	0	1	0	2	2	0	1	0	0	0

Figure 9: Occurrence of words

Frequencies are:

D1: [1,1,1,2,2,1,0,0,0,0]

D2: [0,1,1,1,1,0,1,1,0,0]

D3: [0,1,1,1,1,0,0,0,1,1]

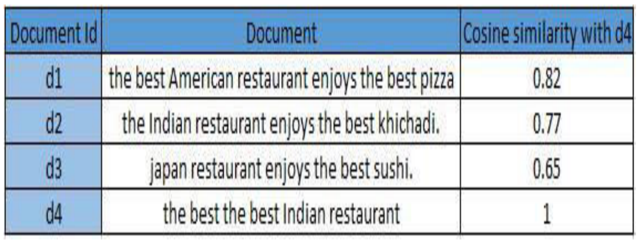
D4: [0,1,0,2,2,0,1,0,0,0]

From Cosine formula:

Similarity=

$$\frac{[1,1,1,2,2,1,0,0,0,0] * [0,1,0,2,2,0,1,0,0,0]^{T}}{\text{Sqrt}(1+1+1+4+4+1+0+0+0+0) * \text{Sqrt}(0+1+0+4+4+0+1+0+0+0)}$$

Cosine similarity of d1: d4 =9/11=0.82. After calculating for each document, the observed results are shown in figure 8.



Document Id	Document	Cosine similarity with d4
d1	the best American restaurant enjoys the best pizza	0.82
d2	the Indian restaurant enjoys the best khichadi.	0.77
d3	japan restaurant enjoys the best sushi.	0.65
d4	the best the best Indian restaurant	1

Figure 10: Cosine Similarity with d4

The similarity of document d2 and d4 should be much similar but the document d1 and d4 are more similar, this is because the common word matching from d1 to d4 is more as compared to d2 (as shown in Figure 11 and Figure 12).

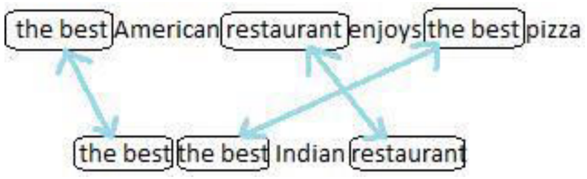


Figure 11: Representing d1 and d4

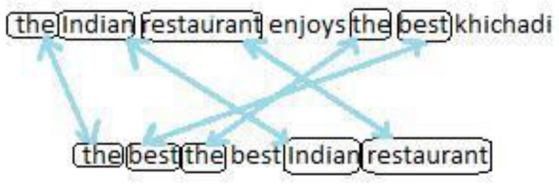
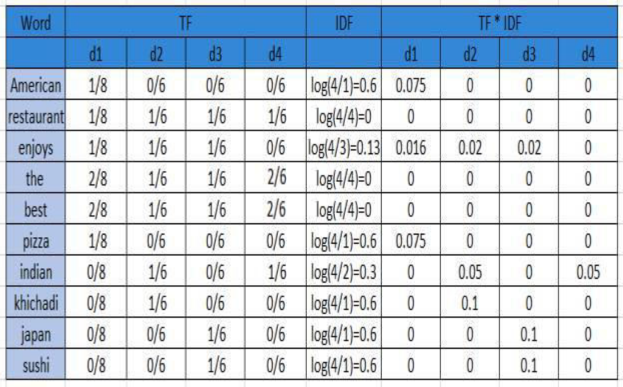


Figure 12: Representing d2 and d4

The above figure illustrates that the most desirable matching is becoming the least desirable due to occurrence of stop words.

To overcome this limitation and to reduce the importance of stop words TF-IDF (Term Frequency-Inverse Document Frequency) can be used.

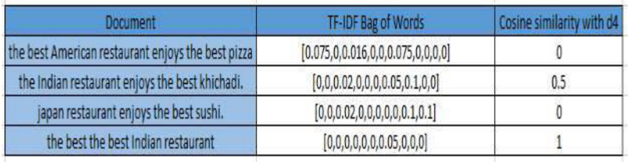
Integrating cosine similarity with TF-IDF, the observation are in the following table:



Word	TF				IDF	TF * IDF			
	d1	d2	d3	d4		d1	d2	d3	d4
American	1/8	0/6	0/6	0/6	log(4/1)=0.6	0.075	0	0	0
restaurant	1/8	1/6	1/6	1/6	log(4/4)=0	0	0	0	0
enjoys	1/8	1/6	1/6	0/6	log(4/3)=0.13	0.016	0.02	0.02	0
the	2/8	1/6	1/6	2/6	log(4/4)=0	0	0	0	0
best	2/8	1/6	1/6	2/6	log(4/4)=0	0	0	0	0
pizza	1/8	0/6	0/6	0/6	log(4/1)=0.6	0.075	0	0	0
indian	0/8	1/6	0/6	1/6	log(4/2)=0.3	0	0.05	0	0.05
khichadi	0/8	1/6	0/6	0/6	log(4/1)=0.6	0	0.1	0	0
japan	0/8	0/6	1/6	0/6	log(4/1)=0.6	0	0	0.1	0
sushi	0/8	0/6	1/6	0/6	log(4/1)=0.6	0	0	0.1	0

Figure 13: Integration of TF-IDF

The final output is as follows:



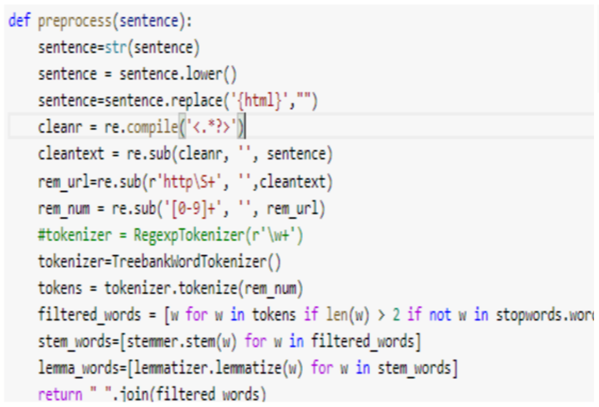
Document	TF-IDF Bag of Words	Cosine similarity with d4
the best American restaurant enjoys the best pizza	[0.075,0,0.016,0,0,0.075,0,0,0,0]	0
the Indian restaurant enjoys the best khichadi.	[0,0,0.02,0,0,0,0.05,0.1,0,0]	0.5
japan restaurant enjoys the best sushi.	[0,0,0.02,0,0,0,0,0,0.1,0.1]	0
the best the best Indian restaurant	[0,0,0,0,0,0,0.05,0,0,0]	1

Figure 14: Final Output

After integrating TF-IDF the final results match the desired results. i.e. d2 is more similar to d4.

IV. Result

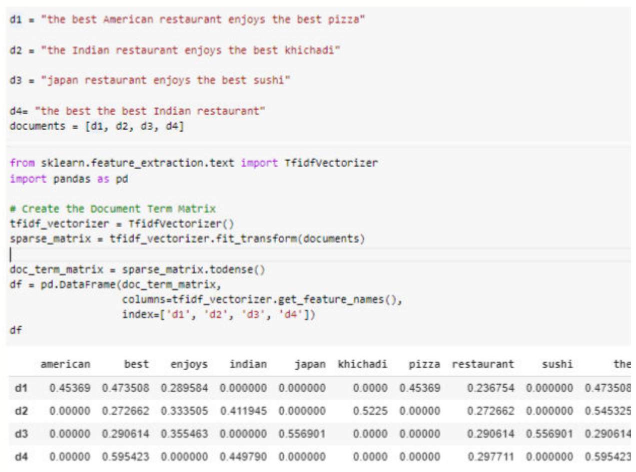
Text pre-processing helps in removing unwanted noise, such as punctuation marks and stop words and make the raw text into refine text which can further useful for similarity algorithm.



```
def preprocess(sentence):
    sentence=str(sentence)
    sentence = sentence.lower()
    sentence=sentence.replace('{html}',"")
    cleanr = re.compile('<.*?>')
    cleantext = re.sub(cleanr, '', sentence)
    rem_url=re.sub(r'http\S+', '',cleantext)
    rem_num = re.sub('[0-9]+', '', rem_url)
    #tokenizer = RegexpTokenizer(r'\w+')
    tokenizer=TreebankWordTokenizer()
    tokens = tokenizer.tokenize(rem_num)
    filtered_words = [w for w in tokens if len(w) > 2 if not w in stopwords.wor
    stem_words=[stemmer.stem(w) for w in filtered_words]
    lemma_words=[lemmatizer.lemmatize(w) for w in stem_words]
    return " ".join(filtered words)
```

Figure 15: Text pre-processing

Cosine similarity is one of the algorithm to find the similarity between the documents but the similarity score suffers by only applying term frequency vector (CountVectorizer). By combining TF-IDF(TfidfVectorizer) the similarity score drastically improves.



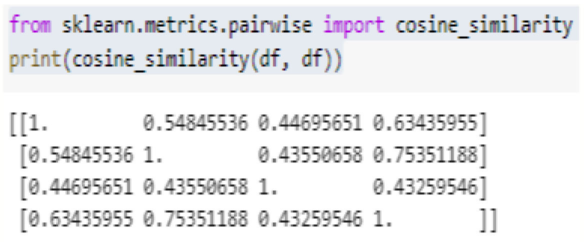
```
d1 = "the best American restaurant enjoys the best pizza"

d2 = "the Indian restaurant enjoys the best khichadi"

d3 = "japan restaurant enjoys the best sushi"

d4= "the best the best Indian restaurant"
documents = [d1, d2, d3, d4]
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

# Create the Document Term Matrix
tfidf_vectorizer = TfidfVectorizer()
sparse_matrix = tfidf_vectorizer.fit_transform(documents)

doc_term_matrix = sparse_matrix.todense()
df = pd.DataFrame(doc_term_matrix,
                  columns=tfidf_vectorizer.get_feature_names(),
                  index=['d1', 'd2', 'd3', 'd4'])
df
```

	american	best	enjoys	indian	japan	khichadi	pizza	restaurant	sushi	the
d1	0.45369	0.473508	0.289584	0.000000	0.000000	0.0000	0.45369	0.236754	0.000000	0.473508
d2	0.00000	0.272662	0.333505	0.411945	0.000000	0.5225	0.00000	0.272662	0.000000	0.545325
d3	0.00000	0.290614	0.355463	0.000000	0.556901	0.0000	0.00000	0.290614	0.556901	0.290614
d4	0.00000	0.595423	0.000000	0.449790	0.000000	0.0000	0.00000	0.297711	0.000000	0.595423

Figure 16: Intermediate process

The final output after calculating TF-IDF will generate the similarity score for d2 and d4 is 75.3% which is true.



```
from sklearn.metrics.pairwise import cosine_similarity
print(cosine_similarity(df, df))
```

```
[[1.         0.54845536 0.44695651 0.63435955]
 [0.54845536 1.         0.43550658 0.75351188]
 [0.44695651 0.43550658 1.         0.43259546]
 [0.63435955 0.75351188 0.43259546 1.        ]]
```

Figure 17: Final Output

V. Conclusion and Future Enhancement

Text data is usually the most generated data from past till now be it in any form. Text matching and Text Similarity is covering a broad spectrum, as artificial intelligence is improving it is attracting new technologies such as chatbot, information retrieval systems, context gathering in which pattern matching is widely used. Chatbot recognizes the received text and then matches the intent with the received text and responds accordingly. But it is easily observable that no single algorithm is fully accurate. Highest accuracy can be achieved only with combinations among this algorithm. For example, [1] Cosine Similarity with TF-IDF [2] Cosine Similarity with Word2Vec [3] Cosine Similarity with BERT Embedding and many more. This algorithm is suitable with subjective text check which can be used in online platforms in recruitment processes. Since mass recruitment process becomes tedious, subjective test could be taken without involvement of humans and result can be generated using text similarity, this would be cost effective and would avoid biasness during recruitment.

VI. References

[1] Tomas Mikolov,Ilya Sutskever,Kai Chen,Greg Corrado,Jeffrey Dean,Distributed Representations of Words and Phrases and their Compositionality Google Inc. Mountain View California 2013

[2] Hakak, Saqib & Kamsin, Amirrudin & Palaiahnakote, Shivakumara & Gilkar, Gulshan & Khan, Wazir & Imran, Muhammad. (2019). Exact String Matching Algorithms: Survey, Issues, and Future Research Directions. IEEE Access. PP. 1-1. 10.1109/ACCESS.2019.2914071.

[3] Albitar, Shereen & Fournier, Sébastien & Espinasse, Bernard. (2014). An Effective TF/IDF-based Text-to-Text Semantic Similarity Measure for Text Classification. 10.1007/978-3-319-11749-2_8.

[4] S. Zhang, Y. Hu and G. Bian, "Research on string similarity algorithm based on Levenshtein Distance," 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Chongqing, 2017, pp. 2247-2251.

[5]https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/

[6]https://www.machinelearningplus.com/nlp/cosine-similarity/