Document Similarity in Vector Space Model

*A **project report** in partial fulfilment for the degree of*

Bachelors of Technology (B. Tech)

in

Computer Science and Engineering

Submitted By

Akhil Anand Pritam, Ashish Kumar & Nitesh Kumar Upadhyay

12000115012, 12000115028 & 12000115056

under the supervision of

Dr. Maunendra Sankar Desarkar, Assistant Professor

Department of CSE, Indian Institute of Technology, Hyderabad, Telangana

&

Prof. Dinesh K. Pradhan, Assistant Professor

Department of CSE, BCREC, Durgapur, West Bengal

Department of Computer Science and Engineering

Dr. B. C. Roy Engineering College, Durgapur

May, 2019

Declaration

We declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Ashish Kumar (12000115028)
Akhil Anand Pritam (12000115012)
Nitesh Kumar Upadhyay (12000115056)

Date: May 21, 2019

Certificate of Recommendation

This is to certify that the report entitled **“Document Similarity in Vector Space Model"**, completed by **Ashish Kumar, Akhil Anand Pritam** & **Nitesh Kumar Upadhyay** for the partial fulfillment of requirements for the award of degree of **Bachelor of Technology (B. Tech) in Computer Science and Engineering**, is a bonafide research work under the guidance of **Dr. Maunendra Sankar Desarkar & Prof. Dinesh K. Pradhan**. The results embodied in this report have not been submitted to any other University or Institute for the award of any degree or diploma. In our opinion, this report is of the standard required for the partial fulfillment of the requirements for the award of the degree of **Bachelor’s of Technology (B. Tech)**.

Dr. Maunendra Sankar Desarkar
Assistant Professor, CSE Department
Indian Institute of Technology
Hyderabad, Telangana

Prof. Dinesh K. Pradhan
Assistant Professor, CSE Department
Dr. B. C. Roy Engineering College
Fuljhore, Durgapur - 713206, West Bengal

Dr. Chandan Koner
HoD, CSE Department
Dr. B. C. Roy Engineering College
Fuljhore, Durgapur - 713206, West Bengal

Certificate of Approval

This project report entitled **“Document Similarity In Vector Space Model"**, completed by **Ashish Kumar, Akhil Anand Pritam** & **Nitesh Kumar Upadhyay** is approved for the degree of B. Tech in Computer Science & Engineering.

Examiners

____

____

____

Supervisor

____

____

HoD

____

Project Coordinator

____

(Report Format Verified)

Date: May 21, 2019

Place: Durgapur

Acknowledgement

With deep sense of gratitude, we want to thank our project mentors **Dr. Maunendra Sankar Desarkar and Prof. Dinesh K. Pradhan**. Without their continuous dedication, assistance and support in each and every step throughout project, this thesis would have never been accomplished. Their guidance and valuable suggestions throughout helped us in doing all experiments and also, writing this thesis. We would like to thank all for their unwavering support, patience, and positive encouragements throughout duration of this project.

We wish to acknowledge **Dr. Chandan Koner**, HoD, Department of Computer Science and Engineering, for providing us supporting environment and labs to carry on our research work. We would also like to show gratitude and respect to all faculty members of our department for their support.

Last but not the least, we would like to thank our parents for their unconditional support always.

Ashish Kumar
Akhil Anand Pritam
Nitesh Kumar Upadhyay

Abstract

The problem of finding similarity score among all document set having pair of sentences and how Vector space can be used to solve it. A vector space is a mathematical structure formed by a collection of elements called vectors, which may be added together and multiplied ("scaled") by numbers, called scalars in this context. A document is a bag of words or a collection of words or terms. The problem can be easily experienced in the domain of web search or classification, where the aim is to find out documents which are similar in context or content.

Measuring semantic nearness of documents is important for accurate information retrieval, automated text categorization and classification. Inspired by the observation that text documents contain semantically coherent set of ideas/topics, this paper presents the design and experimental evaluation of a method to represent a text document as a set of concepts. Based on this, we propose a method to measure semantic nearness of texts. Our method makes use of Python3.1 with different packages. In order to show the effectiveness of our representation of texts, we compare experimental results of text classification and clustering with the results of classification and clustering with standard techniques.

We identified top words (or phrases) from a collection and represented each document as a vector in the vector space determined by those top words. After that we found similarities between documents in that vector space. Finally, we implemented on the given dataset having pair of sentences and produced output as score similarity among those document set.

Contents

List of Figures

Introduction

The problem was finding documents which were similar and how Vector space can be used to solve it. A vector space is a mathematical structure formed by a collection of elements called vectors, which may be added together and multiplied ("scaled") by numbers, called scalars in this context. A document is a bag of words or a collection of terms. The problem can be easily experienced in the domain of web search or classification, where the aim is to find out documents which are similar in context or content.

A key-step towards achieving the goal of retrieving all and only the most relevant information, is detecting semantic similarity, i.e., judging how similar the contents of the two texts are. There is given an information theoretic definition of similarity. Fundamentally, the similarity of two objects is measured by the number of features they have in common. The similarity of texts is estimated based on the number of words they have in common i.e. Term Frequency (TF) and the relative importance of these in the corpus i.e Inverse Document Frequency (IDF). Unfortunately, owing to the richness (and vagaries) of natural languages such as English, these methods fail to identify the actual semantic relatedness between texts if words from both texts are matched directly. Many approaches have been proposed in the past to measure similarity between words based on their relationships such as synonymy-antonym etc. But these require finding the correct sense of each word in the text. We are facing an ever increasing volume of text documents. The abundant texts flowing over the Internet, huge collections of documents in digital libraries and repositories, and digitized personal information such as blog articles and emails are piling up quickly every day. These have brought challenges for the effective and efficient organization of text documents.

Word Embedding is one of the most popular representation of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc. What are Word Embedding exactly? Loosely speaking, they are vector representations of a particular word. Having said this, what follows is how do we generate them? More importantly, how do they capture the context? Word2Vec is one of the most popular technique to learn Word Embedding using shallow neural network.

Motivation: A document or zone that mentions a query term more often has more to do with that query and therefore should receive a higher score. To motivate this, we recall the notion of a free text query: a query in which the terms of the query are typed free-form into the search interface, without any connecting search operators (such as Boolean operators). This query style, which is extremely popular on the web, views the query as simply as a set of words. A plausible scoring mechanism then is to compute a score that is the sum, over the query terms, of the match scores between each query term and the document. Towards this end, we assign to each term in a document a weight for that term, that depends on the number of occurrences of the term in the document. We would like to compute a score between a query term t and a document d based on the weight of t in d. The simplest approach is to assign the weight to be equal to the number of occurrences of term t in document d.

Objective: Main objective of this project work is to find out importance of particular word or document in the list of documents/Dataset. The main objective is that the richness of natural language such as English, these methods fail to identify the actual semantic relatedness between texts if words from both texts are not matched directly. Many approaches have been proposed in the past to measure similarity between words based on their relationships such as synonymy-antonymy etc. But these require finding the correct sense of each word in the text. So there is a need of different measures of similarity which can be used to overcome these problem.

Literature Survey

Learning Weight [1]: How do they determine the weights gi for weighted zone scoring? These weights could be specied by an expert (or, in principle, the user); but increasingly, these weights are "learned" using training examples that have been judged editorially. This latter methodology falls under a general class of approaches to scoring and ranking in information retrieval, known as machine-learned relevance. They are provided with a set of training examples, each of which is a tuple consisting of a query q and a document d, together with a relevance judgment for d on q. In the simplest form, each relevance judgments is either Relevant or Non-relevant. More sophisticated implementations of the methodology make use of more nuanced judgments. The weights gi are then "learned" from these examples, in order that the learned scores approximate the relevance judgments in the training examples.

Term Frequency and Weighting [1]: For them scoring has hinged on whether or not a query term is present in a zone within a document. They took the next logical step: a document or zone that mentions a query term more often has more to do with that query and therefore should receive a higher score. To motivate this, we recall the notion of a free text query introduced in Section 1.4: a query in which the terms of the query are typed free form into the search interface, without any connecting search operators (such as Boolean operators). This query style, which is extremely popular on the web, views the query as simply a set of words. A plausible scoring mechanism then is to compute a score that is the sum,over the query terms, of the match scores between each query term and the document.

The Vector Space Model for Scoring [1]: We developed the notion of a document vector that captures the relative importance of the terms in a document. The representation of a set of documents as vectors in a common vector space is known as the vector space model and is fundamental to a host of information retrieval operations ranging from scoring documents on a query,document classication and document clustering. We rst develop the basic ideas underlying vector space scoring; a pivotal step in this development is the view of queries as vectors in the same vector space as the document collection.

Word Embedding as a Word2Vec experiment by Tomas Mikolov and colleagues [2]: The word2vec software of Tomas Mikolov and colleagues1 has gained a lot of traction lately, and provides state-of-the-art word embeddings. The learning models behind the software are described in two research papers followed by them which mention in the paper refrenced here. We found the description of the models in these papers to be some what cryptic and hard to follow. While the motivations and presentation may be obvious to the neural-networks language-modeling crowd, we had to struggle quite a bit to figure out the rationale behind the equations.

It's like numbers are language, like all the letters in the language are turned into numbers, and so it's something that everyone understands the same way. You lose the sounds of the letters and whether they click or pop or touch the palate, or go ooh or aah, and anything that can be misread or con you with its music or the pictures it puts in your mind, all of that is gone, along with the accent, and you have a new understanding entirely, a language of numbers, and everything becomes as clear to everyone as the writing on the wall

Vector Space Model

3.1 Introduction to Vector Space Model

Vector Space Model (VSM) [1, 3] represent (embed) words in a continuous vector space where semantically similar words are mapped to nearby points ('are embedded nearby each other'). VSMs have a long, rich history in NLP, but all methods depend in some way or another on the Distributional Hypothesis, which states that words that appear in the same contexts share semantic meaning. The different approaches that leverage this principle can be divided into two categories: count-based methods (e.g. Latent Semantic Analysis), and predictive methods (e.g. neural probabilistic language models).

Count-based methods compute the statistics of how often some word co-occurs with its neighbor words in a large text corpus, and then map these count-statistics down to a small, dense vector for each word. Predictive models directly try to predict a word from its neighbors in terms of learned small, dense embedding vectors (considered parameters of the model).

Word2Vec is a particularly computationally-efficient predictive model for learning Word Embedding from raw text. It comes in two flavors, the Continuous Bag-of-Words model (CBOW) and the Skip-Gram model. Algorithmically, these models are similar, except that CBOW predicts target words (e.g. 'mat') from source context words ('the cat sits on the'), while the skip-gram does the inverse and predicts source context-words from the target words. This inversion might seem like an arbitrary choice, but statistically it has the effect that CBOW smoothes over a lot of the distributional information (by treating an entire context as one observation).

The representation of a set of documents as vectors in a common vector space is known as the vector space model and is fundamental to a host of information retrieval operations ranging from scoring documents on a query, document classication and document clustering. We rst develop the basic ideas underlying vector space scoring; a pivotal step in this development is the view of queries as vectors in the same vector space as the document collection.

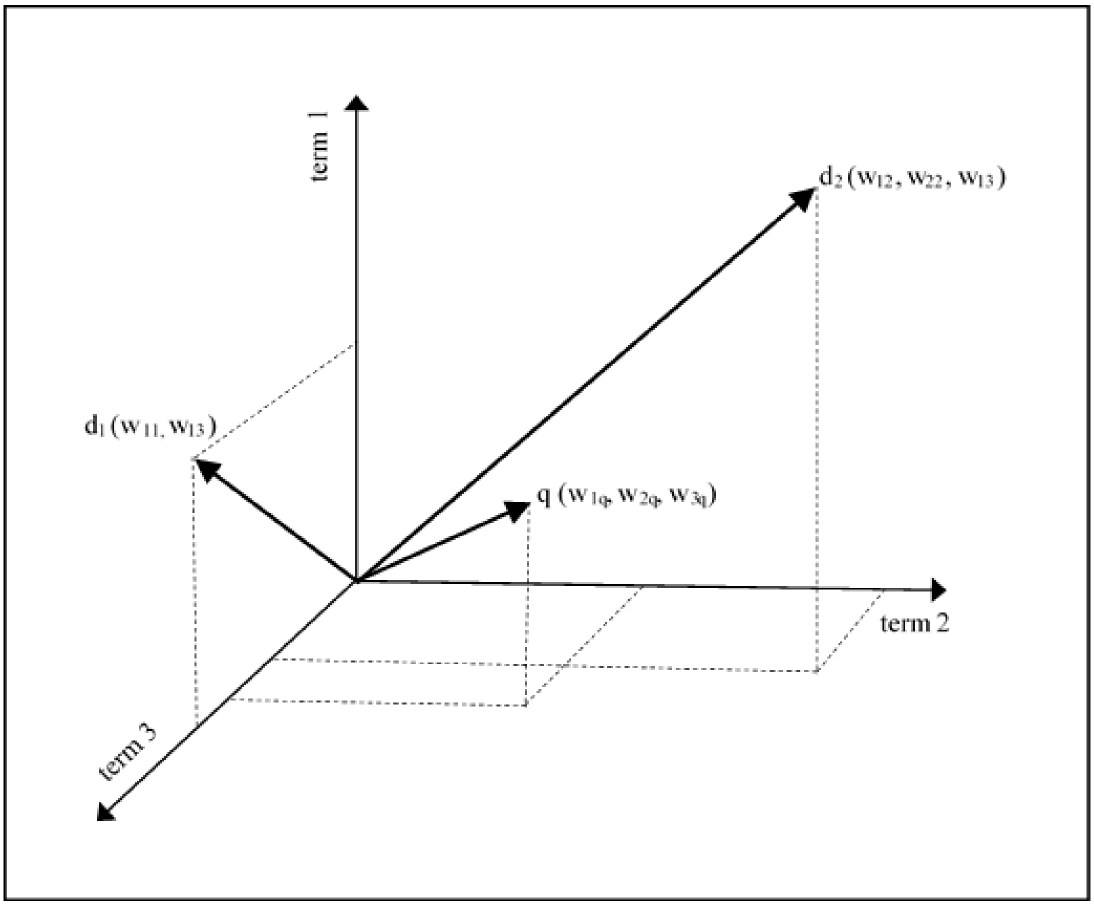

Figure 3.1: **Vector Space Model**

3.1.1 Dot Product

We denote by $\vec{V}(d)$ the vector derived from document d, with one component in the vector for each dictionary term. Unless otherwise specied, the reader may assume that the components are computed using the Tf-Idf weighting scheme, although the particular weighting scheme is immaterial to the discussion that follows. The set of documents in a collection then may be viewed as a set of vectors in a vector space, in which there is one axis for each term. This representation loses the relative ordering of the terms in each document;as an example: the documents "Mary is quicker than John" and "John is quicker than Mary" are identical in such a bag of words representation.

3.2 Cosine Similarity

Cosine similarity is a metric used to measure how similar the documents are irrespective of their size. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space. The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance (due to the size of the document), chances are they may still be oriented closer together. The smaller the angle, higher the cosine similarity.
Cosine similarity is a metric used to determine how similar the documents are irrespective of their size.

Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space. In this context, the two vectors I am talking about are arrays containing the word counts of two documents.
As a similarity metric, how does cosine similarity differ from the number of common words?
When plotted on a multi-dimensional space, where each dimension corresponds to a word in the document, the cosine similarity captures the orientation (the angle) of the documents and not the magnitude. If you want the magnitude, compute the Euclidean distance instead.

The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance because of the size (like, the word ‘cricket’ appeared

50 times in one document and 10 times in another) they could still have a smaller angle between them. Smaller the angle, higher the similarity.

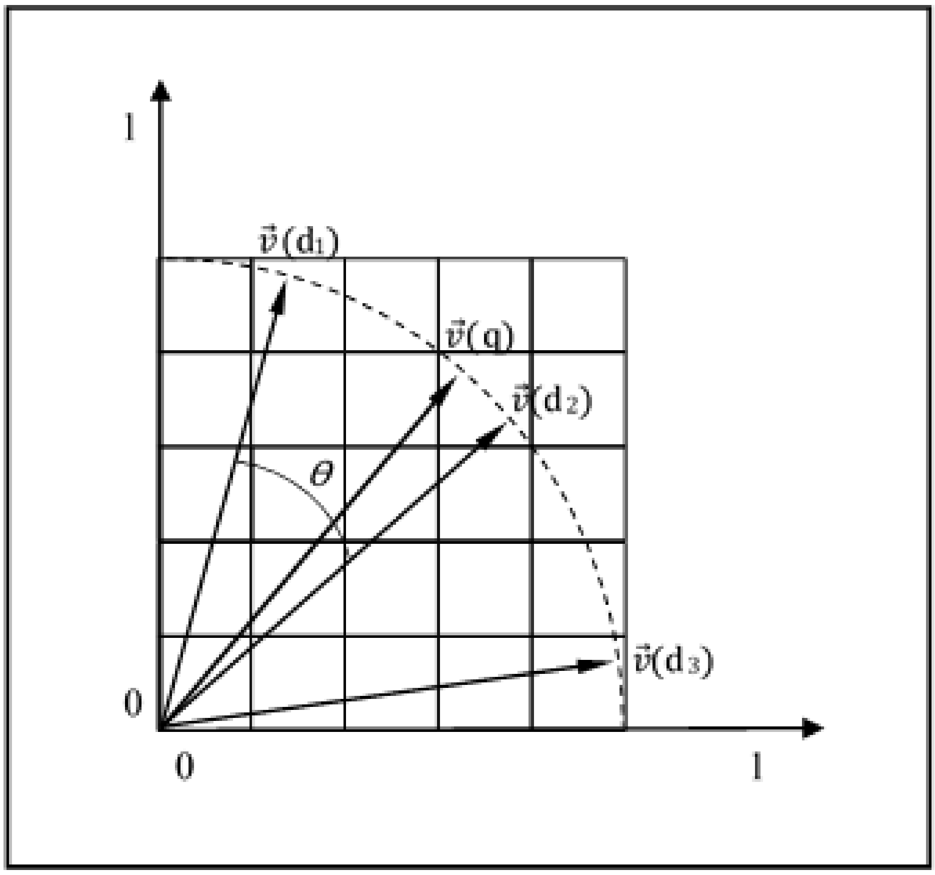

Figure 3.2: **Cosine Similarity**

To compensate for the effect of document length, the standard way of quantifying the similarity between two documents d1 and d2 is to compute the cosine similarity of their vector representations $\vec{V}(d1)$ and $\vec{V}(d2)$.

$$sim(d1, d2) = \frac{\vec{V}(d1).\vec{V}(d2)}{|\vec{V}(d1)||\vec{V}(d2)|}$$

TF Approach

Term Frequency [1, 4]: Scoring has hinged on whether or not a query term is present in a zone within a document. We take the next logical step: a document or zone that mentions a query term more often has more to do with that query and therefore should receive a higher score. To motivate this, we recall the notion of a free text query. A query in which the terms of the query are typed free form into the search interface, without any connecting search operators (such as Boolean operators). This query style, which is extremely popular on the web, views the query as simply a set of words. A plausible scoring mechanism then is to compute a score that is the sum,over the query m terms, of the match scores between each query term and the document. Towards this end, we assign to each term in a document a weight for that term, that depends on the number of occurrences of the term in the document.

We would like to compute a score between a query term t and a document d, based on the weight of t in d. The simplest approach is to assign the weight to be equal to the number of occurrences of term t in document d. This weighting scheme is referred to as term frequency and is denoted tf(t,d) with the subscripts denoting the term and the document in order. It simply means the number of particular terms in the document divided by total number of terms in that document. And it is called term frequency of a particular word in a document.

$$Tf(word, doc) = \frac{(freq(word, doc))}{float(word\_count(doc))} \tag{4.1}$$

Now we have to measure similarity score between pair of sentences, for which we have two ways mainly.

1. Tfvectorizer

2. Take the union of terms present in s1 and s2. For each term in the union, get the term frequency and get the vector accordingly, for s1 and separately s2.
The output arised for Document set is plotted and shown below in fig 4.1.

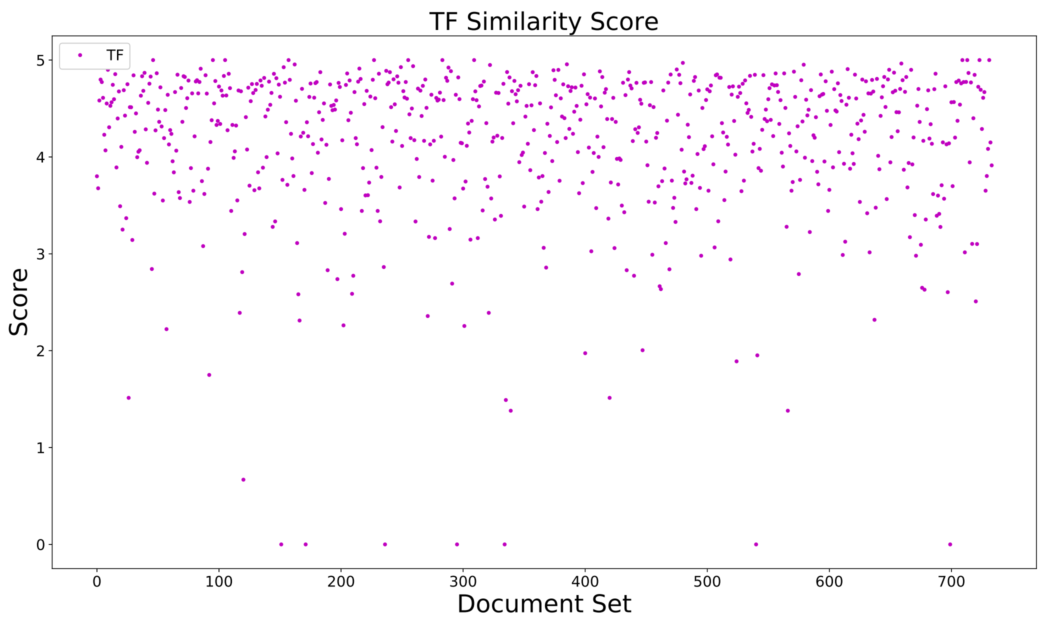

Figure 4.1: **TF Document set similarity Score arised**

Now we are plotting the comparison graph of the Actual value and arised similar score in TF approach, which is shown in fig 4.2.

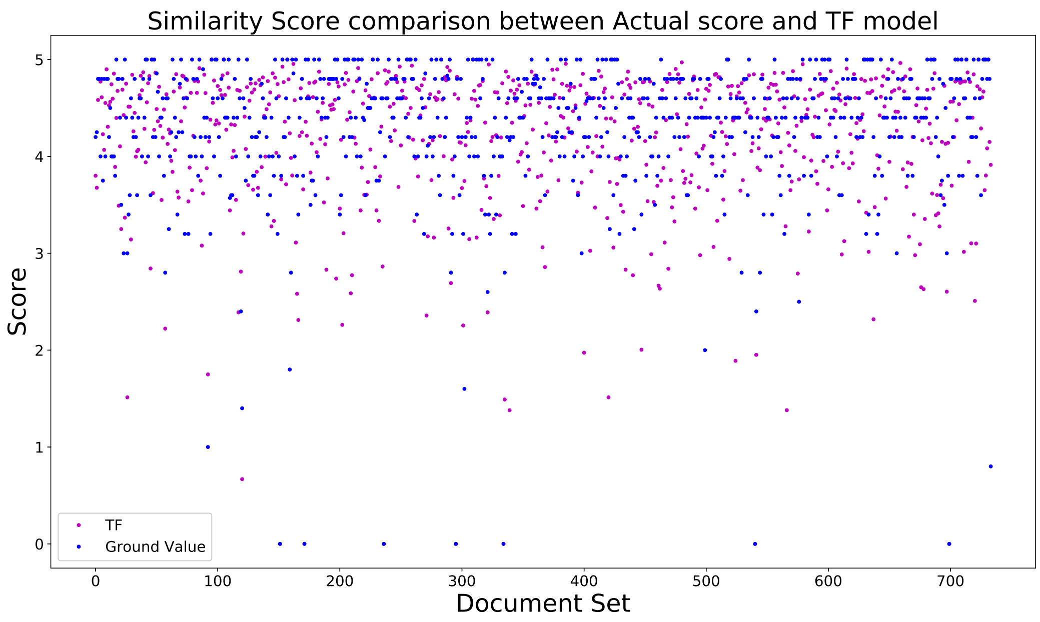

Figure 4.2: **Comparison among TF Document set similarity Score arised and Actual Score**

The code for calculating the score by TF Approach can be found in Appendix II.
Here is the evaluation matrix of document set for Actual score and TF output score.

Table 3.1 : Similarity Score Comparision between Actual Score and TF Score

Document Set	1	2	3	4	5	6	7	8	9	10
Actual Score	4.2	4.25	4.8	4.8	4	4.8	3.75	4.8	4	4.8
TF	3.80	3.68	4.58	4.80	4.77	4.61	4.23	4.07	4.55	4.90

TF-IDF Model

TF-IDF [1, 5] stands for term frequency-inverse document frequency, and the TF-IDF weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the TF-IDF weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.
One of the simplest ranking functions is computed by summing the TF-IDF for each query term; many more sophisticated ranking functions are variants of this simple model.TF-IDF can be successfully used for stop-words filtering in various subject fields including text summarization and classification.
How to Compute:
Typically, the TF-IDF weight is composed by two terms: the first computes the normalized Term Frequency (TF), aka. the number of times a word appears in a document, divided by the total number of words in that document; the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

Term Frequency: We would like to compute a score between a query term t and a document d, based on the weight of t in d. The simplest approach is to assign the weight to be equal to the number of occurrences of term t in document d. This weighting scheme is referred to as term frequency and is denoted tf(t,d) with the subscripts denoting the term and the document in order.

Bag of Words: For a document d, the set of weights determined by the TF weights above (or indeed any weighting function that maps the number of occurrences of t in d to a positive real value) may be viewed as a quantitative digest of that document. In this view of a document, known in the literature as the bag of words model, the exact ordering of the terms in a document is ignored but the number of occurrences of each term is material (in contrast to Boolean retrieval). We only retain information on the number of occurrences of each term. Thus, the document "Mary is quicker than John" is, in this view, identical to the document "John is quicker than Mary". Nevertheless, it seems intuitive that two documents with similar bag of words representations are similar in content. Before doing so we rst study the question: are all words in a document equally important? Clearly not; we looked at the idea of stop words –words that we decide not to index at all, and therefore do not contribute in any way to retrieval and scoring.

Document Frequency: Instead, it is more commonplace to use for this purpose the document frequency DF(t), dened to be the number of documents in the collection that contain a term t. This is because in trying to discriminate between documents for the purpose of scoring it is better to use a document-level statistic (such as the number of documents containing a term) than to use a collection-wide statistic for the term. The reason to prefer df to cf is ,that collection frequency (cf) and document frequency (DF) can behave rather differently.

Inverse Document Frequency: Raw term frequency as above suffers from a critical problem: all terms are considered equally important when it comes to assessing relevancy on a query. In fact certain terms have little or no discriminating power in determining relevance. For instance, a collection of documents on the auto industry is likely to have the term auto in almost every document. To this end, we introduce a mechanism for attenuating the effect of terms that occur too often in the collection to be meaningful for relevance determination. An immediate idea is to scale down the term weights of terms with high collection frequency, dened to be the total number of occurrences of a term in the collection. The idea would be to reduce the tf weight of a term by a factor that grows with its collection frequency. How is the document frequency (DF) of a term used to scale its weight? Denoting as usual the total number of documents in a collection by N, we dene the inverse document frequency (IDF) of a term t as follows:

$$\text{IDF(t)} = \log \text{(N / DF(t))}$$

Thus the IDF of a rare term is high, whereas the IDF of a frequent term is likely to be low.

TF-IDF weighting: We now combine the denitions of term frequency and inverse document frequency, to produce a composite weight for each term in each document. The TF-IDF weighting scheme assigns to term t a weight in document d given by:

$$\text{TF-IDF (t,d)} = \text{TF(t,d)} \times \text{IDF(t)}$$

In other words, TF-IDF(t, d) assigns to term t a weight in document d that is:
1. highest when t occurs many times within a small number of documents (thus lending high is criminating power to those document).
2. lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal).
3. lowest when the term occurs in virtually all documents.

Document Vector: At this point, we may view each document as a vector with one component corresponding to each term in the dictionary, together with a weight for each component that is given by TF-IDF(t, d). For dictionary terms that do not occur in a document, this weight is zero. This vector form will prove to be crucial to scoring and ranking; we will develop these ideas further. As a rst step, we introduce the overlap score measure: the score of a document d is the sum, over all query terms, of the number of times each of the query terms occurs in d. We can rene this idea so that we add up not the number of occurrences of each query term t in d, but instead the TF-IDF weight of each term in d.

$$Score(q, d) = \sum (TF - IDF)(t, d) \qquad where, t \in q \tag{5.1}$$

We will be getting Similarity scores between pair of documents by applying similarity mechanism on TF-IDF Score. And the Output are plotted which are shown as fig. 5.1 . Now the Plotting we have made is Comparison between Gold Score and output Score, and the Similarity Score comparison of Document Set in TF-IDF method shown as fig. 5.2 .

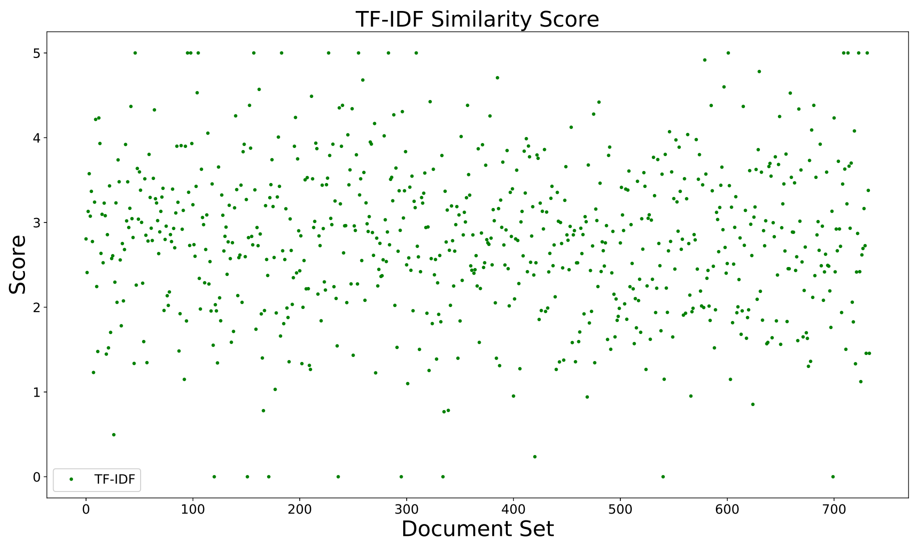

Figure 5.1: **TF-IDF Document set similarity Score**

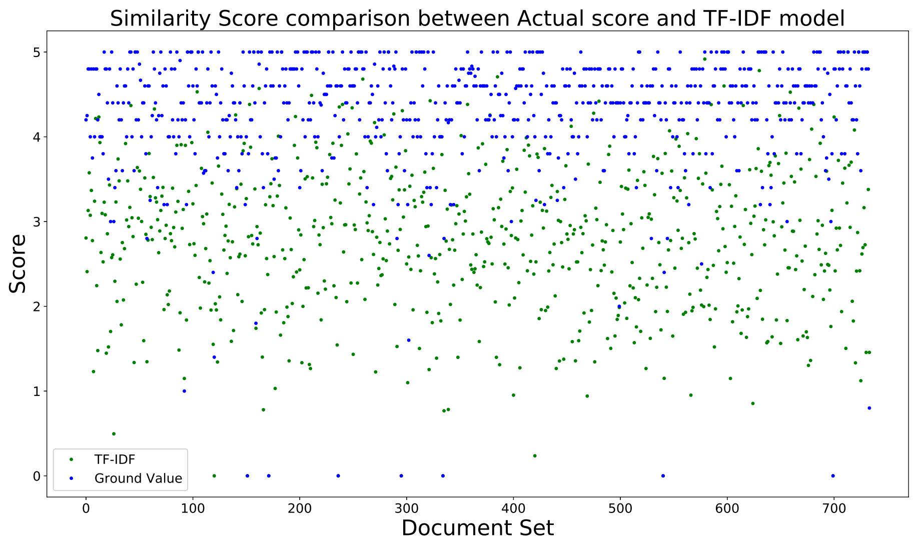

Figure 5.2: **TF-IDF Document set similarity Score in comparison to Actual Score**

The code for calculating the score by TF-IDF Similarity can be found in Appendix III.

The Evaluation Matrix of Sample 10 Document Set with their TF-IDF score and Actual score is shown in Table 4.1. In this matrix, it shows the TF-IDF Score differs from actual score since it measures the similarity according to importance of words not by their semantic meaning.

Table 4.1 : Similarity Score Comparision between Actual Score and TF-IDF Score

Document Set	1	2	3	4	5	6	7	8	9	10
Actual Score	4.2	4.25	4.8	4.8	4	4.8	3.75	4.8	4	4.8
TF-IDF	2.81	2.41	3.13	3.58	3.07	3.37	2.78	1.23	3.24	4.22

Word Embedding

6.1 Introduction to Word Embedding

Word Embedding [1, 6] is one of the most popular representation of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc. What is Word Embedding exactly? Loosely speaking, they are vector representations of a particular word. Having said this, what follows is how do we generate them? More importantly, how do they capture the context? Word2Vec is one of the most popular technique to learn word embedding using shallow neural network. It was developed by Tomas Mikolov in 2013 at Google. Let's tackle this part by part. Why do we need them?

Consider the following similar sentences: Have a good day and Have a great day. They hardly have different meaning. If we construct an exhaustive vocabulary (let's call it V), it would have V = Have, a, good, great, day. Now, let us create a one-hot encoded vector for each of these words in V. Length of our one-hot encoded vector would be equal to the size of V (=5). We would have a vector of zeros except for the element at the index representing the corresponding word in the vocabulary. That particular element would be one. The encodings below would explain this better. Have = [1,0,0,0,0]'; a=[0,1,0,0,0]' ; good=[0,0,1,0,0]' ; great=[0,0,0,1,0]' ; day=[0,0,0,0,1]' (' represents transpose)

If we try to visualize these encodings, we can think of a 5 dimensional space, where each word occupies one of the dimensions and has nothing to do with the rest (no projection along the other dimensions). This means 'good' and 'great' are as different as 'day' and 'have', which is not true. Our objective is to have words with similar context occupy close spatial positions. Mathematically, the cosine of the angle between such vectors should be close

to 1, i.e. angle close to 0. Here comes the idea of generating distributed representations. Intuitively, we introduce some dependence of one word on the other words. The words in context of this word would get a greater share of this dependence. In one hot encoding representations, all the words are independent of each other.
Humans can deal with text format quite intuitively but provided we have millions of documents being generated in a single day, we cannot have humans performing the above the three tasks. It is neither scalable nor effective. So, how do we make computers of today perform clustering, classification etc on a text data since we know that they are generally inefficient at handling and processing strings or texts for any fruitful outputs? Sure, a computer can match two strings and tell you whether they are same or not. But how do we make computers tell you about football or Ronaldo when you search for Messi? How do you make a computer understand that “Apple” in “Apple is a tasty fruit” is a fruit that can be eaten and not a company? The answer to the above questions lie in creating a representation for words that capture their meanings, semantic relationships and the different types of contexts they are used in. And all of these are implemented by using Word Embedding or numerical representations of texts so that computers may handle them. In very simplistic terms, Word Embedding are the texts converted into numbers and there may be different numerical representations of the same text. But before we dive into the details of Word Embedding, the following question should be asked – Why do we need Word Embedding? As it turns out, many Machine Learning algorithms and almost all Deep Learning Architectures are incapable of processing strings or plain text in their raw form. They require numbers as inputs to perform any sort of job, be it classification, regression etc. in broad terms. And with the huge amount of data that is present in the text format, it is imperative to extract knowledge out of it and build applications. Some real world applications of text applications are – sentiment analysis of reviews by Amazon etc., document or news classification or clustering by Google etc. Let us now define Word Embedding formally. A Word Embedding format generally tries to map a word using a dictionary to a vector. Let us break this sentence down into finer details to have a clear view.

Take a look at this example – sentence=” Word Embedding are Word converted into numbers ” A word in this sentence may be “Embeddings” or “numbers ” etc. A dictionary may be the list of all unique words in the sentence. So, a dictionary may look like –

[‘Word’,’Embeddings’,’are’,’Converted’,’into’,’numbers’] A vector representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else. The vector representation of “numbers” in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0]. This is just a very simple method to represent a word in the vector form. Let us look at different types of Word Embedding or Word Vectors and their advantages and disadvantages over the rest.

6.2 Word Embedding using Word2Vec

Word2Vec is a method to construct such an embedding.Word2Vec is an efficient solution to these problems, which leverages the context of the target words. Essentially, we want to use the surrounding words to represent the target words with a Neural Network whose hidden layer encodes the word representation. It can be obtained using two methods (both involving Neural Networks): *Skip Gram* and *Common Bag Of Words (CBOW).*

CBOW Model: This method takes the context of each word as the input and tries to predict the word corresponding to the context. Consider our example: Have a great day. Let the input to the Neural Network be the word, great. Notice that here we are trying to predict a target word (day) using a single context input word great. More specifically, we use the one hot encoding of the input word and measure the output error compared to one hot encoding of the target word (day). In the process of predicting the target word, we learn the vector representation of the target word.
The input or the context word is a one hot encoded vector of size V. The hidden layer contains N neurons and the output is again a V length vector with the elements being the softmax values.

Let’s get the terms in the picture right: - Wvn is the weight matrix that maps the input x to the hidden layer (V*N dimensional matrix) -W‘nv is the weight matrix that maps the hidden layer outputs to the final output layer (N*V dimensional matrix) The hidden layer neurons just copy the weighted sum of inputs to the next layer. There is no activation like sigmoid, tanh or ReLU. The only non-linearity is the softmax calculations in the output layer. But, the model(fig. 4.1) used a single context word to predict the target. We can use multiple context words to do the same.

The actual Architecture we can look deeper as:

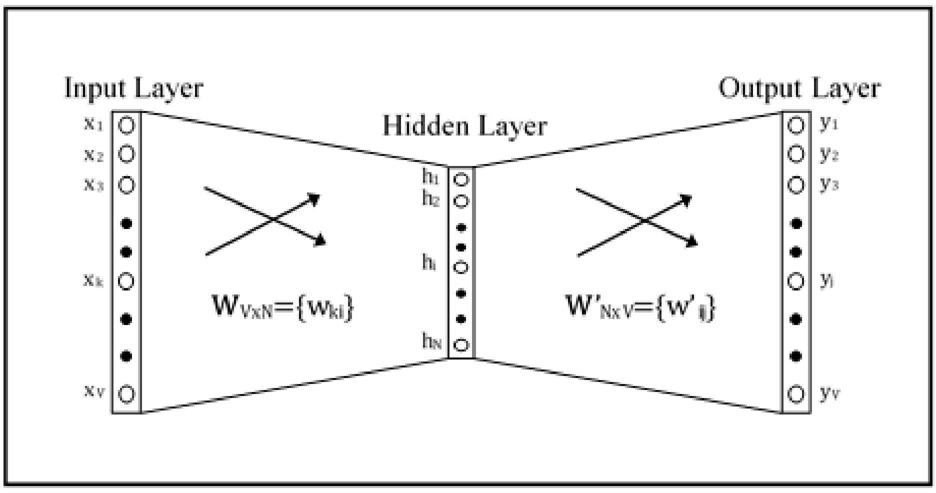

Figure 6.1: **CBOW Model with one word in the context.**

The model(fig. 4.2) takes C context words. When W_{VxN} is used to calculate hidden layer inputs, we take an average over all these C context word inputs.

So, we have seen how word representations are generated using the context words. But there's one more way we can do the same. We can use the target word (whose representation we want to generate) to predict the context and in the process, we produce the representations. Another variant, called Skip Gram model does this.

Advantages: Being probabilistic is nature, it is supposed to perform superior to deterministic methods(generally).

It is low on memory. It does not need to have huge RAM requirements like that of co-occurrence matrix where it needs to store three huge matrices.

*Disadvantages:*CBOW takes the average of the context of a word (as seen above in calculation of hidden activation). For example, Apple can be both a fruit and a company but CBOW takes an average of both the contexts and places it in between a cluster for fruits and companies. Training a CBOW from scratch can take forever if not properly optimized.

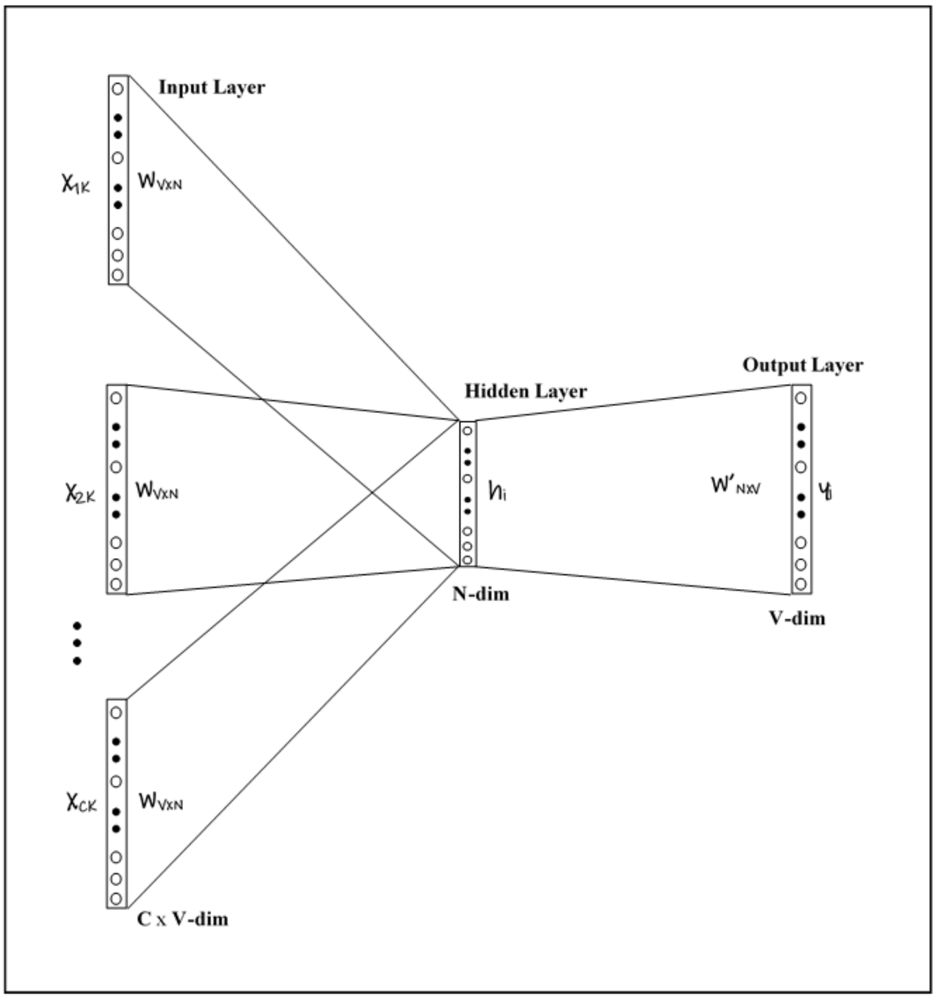

Figure 6.2: **CBOW Model with C word in the context.**

Skip-Gram Model: This looks like multiple-context CBOW model just got flipped. To some extent that is true. We input the target word into the network. The model outputs C probability distributions. What does this mean? For each context position, we get C probability distributions of V probabilities, one for each word. Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit. Skip-gram with negative sub-sampling outperforms every other method generally.

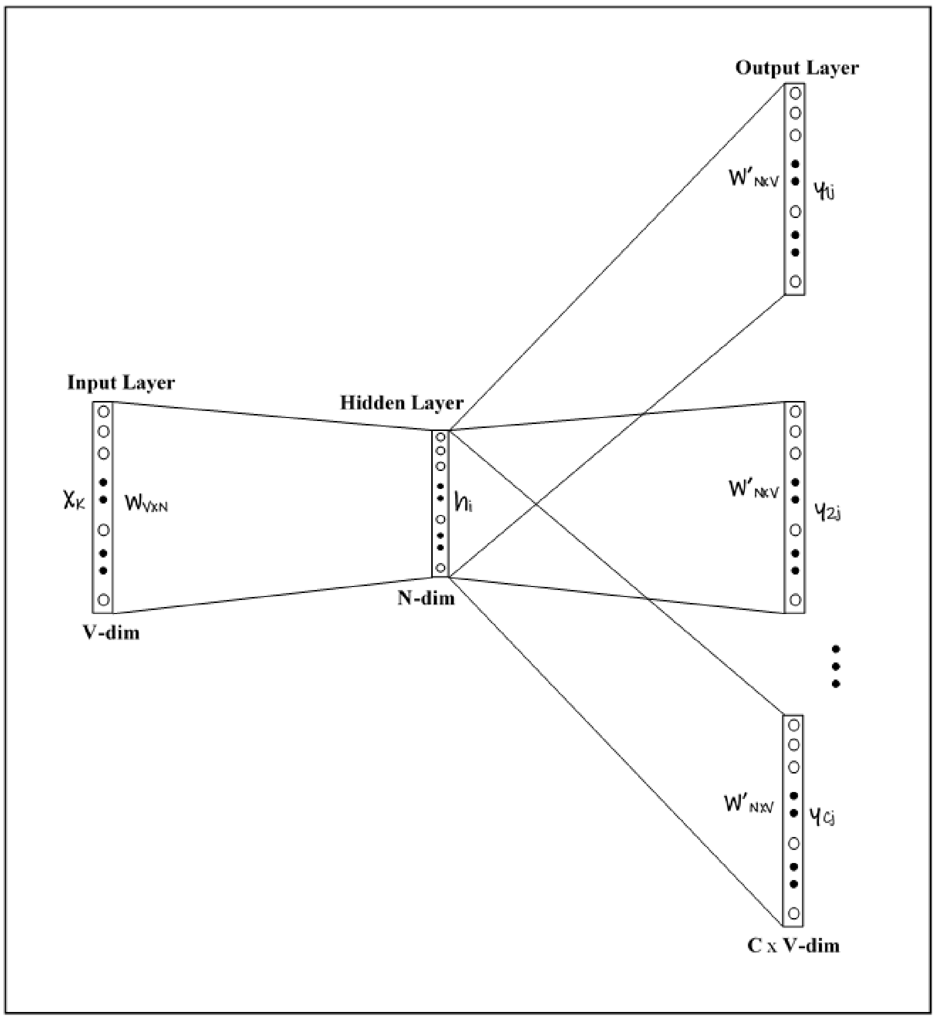

Figure 6.3: **Skip-Gram Model.**

The codes for calculating the score by Word2Vec Similarity can be found in Appendix IV.

The Evaluation Matrix shown in Table 5.1 shows that the score obtained by Word2Vec Model is almost similar to the Actual score, since it uses Word Embedding method with semantic meaning as-well.

Table 5.1 : Similarity Score Comparision between Actual Score and Word2Vec Score

Document Set	1	2	3	4	5	6	7	8	9	10
Actual Score	4.2	4.25	4.8	4.8	4	4.8	3.75	4.8	4	4.8
Word2Vec	4.70	3.91	4.50	4.36	4.27	4.07	3.82	3.68	4.42	4.66

Again here we came with output as a Score of Document set similarity using Word2Vec.The plotting of output as:

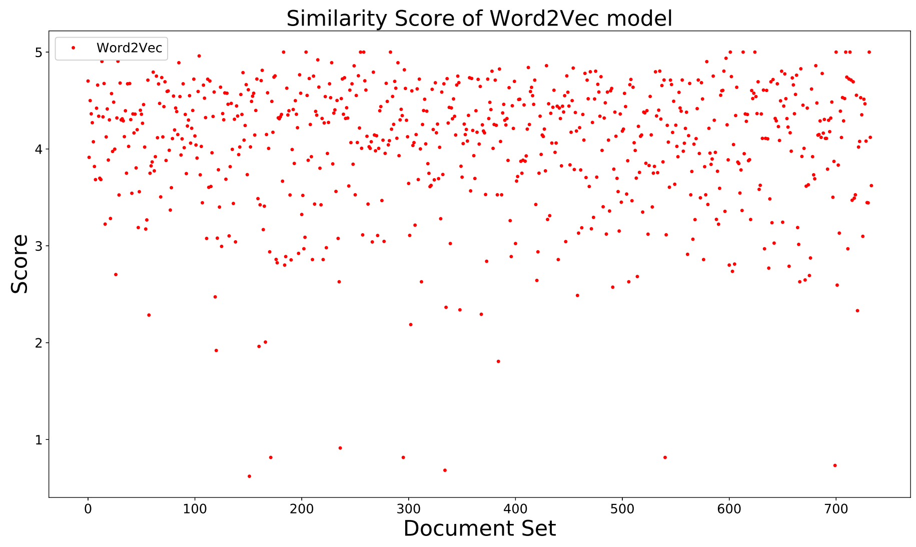

Figure 6.4: **Document Set Similarity in Word2Vec.**

And in comparison to Gold Score and the output arised is plotted as:

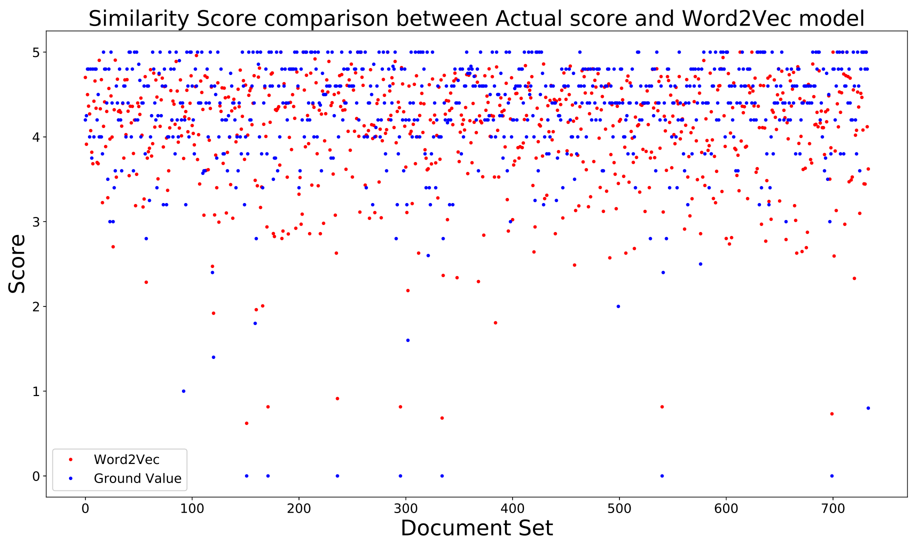

Figure 6.5: **Document Set Similarity Score in Word2Vec in comparison to Actual Score.**

6.3 Training Model in Word2Vec/Training our own Word Vectors

We will be training our own word2vec on a custom corpus. For training the model we will be using gensim and the steps are illustrated as below.
word2Vec requires that a format of list of list for training where every document is contained in a list and every list contains list of tokens of that documents. I won't be covering the pre-preprocessing part here. So let's take an example list of list to train our word2vec model.
sentence=[['Neeraj','Boy'],['Sarwan','is'],['good','boy']]

Training word2vec on 3 sentences
model = gensim.models.Word2Vec(sentence,min_count=1,size=300,workers=4)
Let us try to understand the parameters of this model:
sentence - list of list of our corpus

min_count - the threshold value for the words. Word with frequency greater than this only are going to be included into the model.
size - the number of dimensions in which we wish to represent our word. This is the size of the word vector.
workers - used for parallelization.

Using the model
The new trained model can be used similar to the pre-trained ones.

Printing Similarity Index
print(model.similarity('woman', 'man'))

6.4 Google Pre-trained Model

We are going to use Google's pre-trained model. It contains word vectors for a vocabulary of 3 million words and phrases trained on around 100 billion words from a Google News dataset. The vector length is 300 features.
Sample way Of showing how to use This Google's pre-trained model:
from gensim.models import Word2Vec

Loading the downloaded model
Loading this model using gensim is a piece of cake; you just need to pass in the path to the model file.
model = Word2Vec.load_word2vec_format('GoogleNews-vectors-negative300.bin', binary=True, norm_only=True)

The Model is loaded. It can be used to perform all of the tasks mentioned above.
Getting word vectors of a word
dog = model['dog']

Performing King Queen magic
print(model.most_similar(positive=['woman', 'king'], negative=['man']))

Picking odd one out

print(model.doesn't_match("breakfast cereal dinner lunch".split()))

Printing Similarity Index

print(model.similarity('woman', 'man'))

Note: If the code is running on 32-bit Python, then a memory error may arise. This is because gensim allocates a big matrix to hold all of the word vectors, and if we do the math that's a big matrix.

3 million words * 300 features * 4bytes/feature = 3.35GB

More about Google Pre-trained Model

Does it include stop words? Some stop words like "a", "and", "of" are excluded, but others like "the", "also", "should" are included.

Does it include misspellings of words? -Yes. For instance, it includes both "mispelled" and "misspelled"–the latter is the correct one.

Does it include commonly paired words? -Yes. For instance, it includes "Soviet_Union" and "New_York".

Does it include numbers? -Not directly; e.g., you won't find "100". But it does include entries like "###MHz_DDR2_SDRAM" where I'm assuming the '' are intended to match any digit.

Experimental Analysis

7.1 Introduction to Dataset

The dataset used in our experiment is from github[7].
The Sample Dataset can be found in Appendix I.
The dataset consist of 734 documents with their ground score in gold score format i.e, score out of 5. Dataset used here is a tab separated data(.tsv) file. Each document consists of pair of sentences, S1 & S2 and for those pairs, we have to find the similarity score among them. The Dataset set we were provided consists of actual score with Gold value. So we are showing here the plotted graph of document set with their score value.

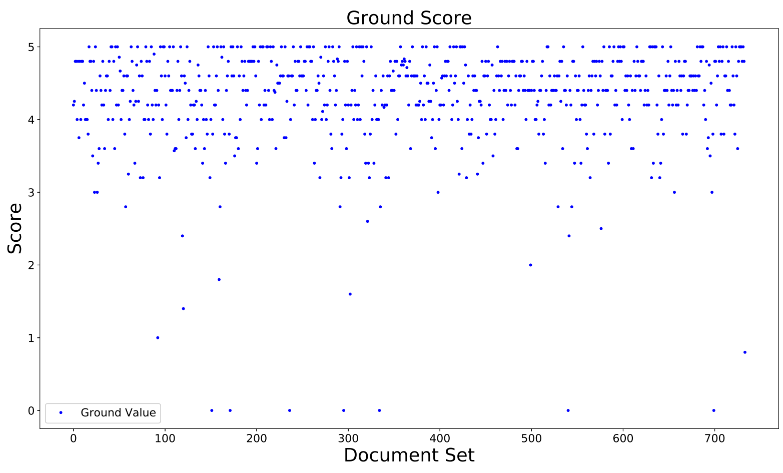

Figure 7.1: **Plotting of Actual score of Dataset.**

7.2 Result Comparison

Finally we came to our result comparison where we have made a comparison among TF approach, TF-IDF method and Word2Vec method. As the evaluation matrix shows the score generated among these three methods is somehow different. And we can see the result generated from Word2Vec method is quite similar with actual score provided.
Since Word2Vec approach uses trained model for which it considers context meaning with semantic nature. TF approach uses the concept of unique meaning of words not semantically similar words, so it shows somehow different score from actual score provided. And TF-IDF model uses the concept of TF and IDF in which importance of the word in that document is considered but it does not considers the meaning of words, also it does not checks for context meaning of terms.

So by comparing we can say that Word2Vec approach yields better result among these three approaches.

Table 6.1 : Similarity Score Comparison between Actual Score, TF Score, TF-IDF Score and Word2Vec Score

Document Set	1	2	3	4	5	6	7	8	9	10
Actual Score	4.2	4.25	4.8	4.8	4	4.8	3.75	4.8	4	4.8
TF	3.80	3.68	4.58	4.80	4.77	4.61	4.23	4.07	4.55	4.90
TF-IDF	2.81	2.41	3.13	3.58	3.07	3.37	2.78	1.23	3.24	4.22
Word2Vec	4.70	3.91	4.50	4.36	4.27	4.07	3.82	3.68	4.42	4.66

Fig. 7.2 shows the comparison of the result of the three experimental approaches: Fig.

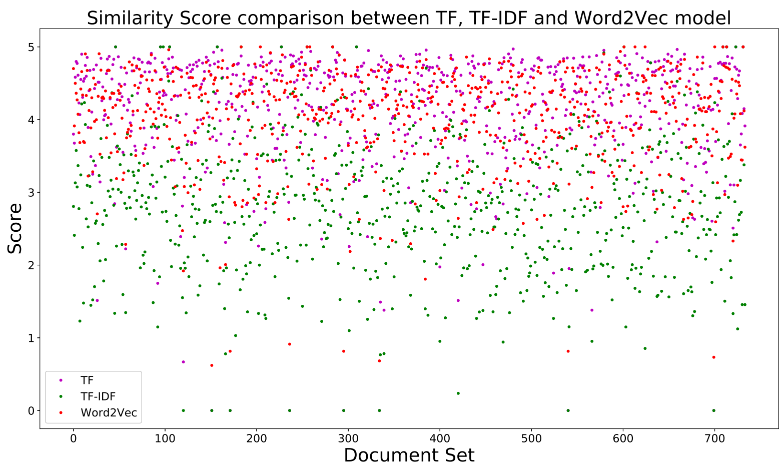

Figure 7.2: **Plot Comparison of TF, TF-IDF and Word2Vec Score.**

7.3 shows the comparison of the actual score and the experimental scores obtained.

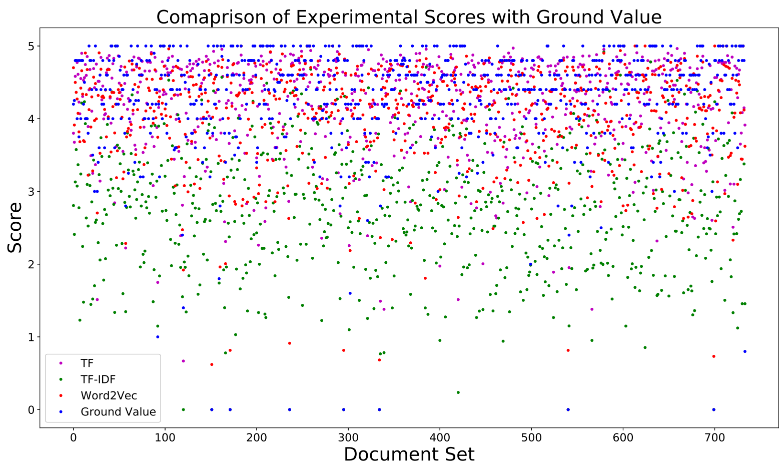

Figure 7.3: **Plot Comparison of TF, TF-IDF & Word2Vec Score with Actual Score.**

Summary & Conclusion

Elaborating and computing a similarity measure that captures well the semantic relationships between texts and/or documents is an important topic given the ubiquity of textual data and the multiple usages it allows: products recommendation, search, sentiment analysis, etc.

After experimental analysis of the three methods viz. TF Approach, TF-IDF Model and Word2Vec Model on the Dataset, we found differences in similarity score among the methods we used. We compared the similarity scores obtained from each method with the actual ground similarity score.

We found that, TF Approach gave us a score which was somewhat close to the actual score. The score was calculated based on the frequency of each term in the document. The score obtained from TF-IDF Model was far different from the actual score as it emphasizes on the importance of the term. So, the score differs very much from the actual score. Coming to the Word2Vec Model, since this method uses semantic similarity approach it gave the most accurate score than other two experimental findings.

Appendices

Appendix I

Sample Dataset

4.200 In Nigeria, Chevron has been accused by the All-Ijaw indigenous people of instigating violence against them and actually paying Nigerian soldiers to shoot protesters at the Warri naval base. In Nigeria, the whole ijaw indigenous showed Chevron to encourage the violence against them and of up to pay Nigerian soldiers to shoot the demonstrators at the naval base from Warri.

4.250 I know that in France they have had whole herd slaughter and this does not seem to be the best way forward. I know that in France, the principle of slaughter of whole herd has been implemented and that this is not the best way to combat this phenomenon.

4.800 Unfortunately, the ultimate objective of a European Constitution would be precisely the opposite, and so, of course, we cannot vote for it. Unfortunately the final objective of a European Constitution would be exactly the opposite and obviously we cannot approve it. 4.800 The right of a government arbitrarily to set aside its own constitution is the defining characteristic of a tyranny. The right for a government to draw aside its constitution arbitrarily is the definition characteristic of a tyranny.

4.000 The House had also fought, however, for the reduction of the funds available for innovative measures to be offset by means of resources from the flexibility instrument, a demand which is recorded in a declaration on the financial perspective in the Interinstitutional Agreement. This Parliament has also fought for this reduction in the funds available for innovative actions should be compensated for by the use of the framework of

flexibility, defined in a statement on the financial perspective.
4.800 The right of a government arbitrarily to set aside its own constitution is the defining characteristic of a tyranny. The right for a government to dismiss arbitrarily its constitution is the definition of a characteristic tyranny.
3.750 After all, it is by no means certain that the proposed definition of equitable price is better than any other, because the various definitions that are currently in use in the Member States are all perfectly satisfactory. Indeed, it is not absolutely certain that the definition of fair price which is better than another, because the definitions used in the Member States in all amply sufficient.
4.800 One could indeed wish for more and for improvement, but I honestly believe that we have made a good start. They can in fact wish to more and better, but I think that it is a good start.
4.000 It must genuinely be a centre for coordinating the network of national agencies which, in turn, must activate and coordinate a network of centres of excellence for food safety at the level of the individual regions. It should be the real coordination of the network of national agencies which, in turn, will speed up and a network of centres of excellence of food at regional level.
4.800 On my own behalf and on behalf of my colleagues in the Committee on Fisheries, I would ask you, Madam President, to send Parliament' s condolences to the families of the victims and to the local authorities in both Brittany and in Marín, Galicia, from where the majority of the victims came. Madam President, I would ask you, on behalf of my colleagues in the Committee on Fisheries and in my own name, to send a message of condolence on the part of Parliament to the families of the victims and the local authorities of Brittany as Marín, city of Galicia, where originating from most of the victims.

Appendix II

TF Similarity

```
import string
from numpy import dot
from numpy.linalg import norm
import numpy as np
import nltk
import math
from scipy import spatial

def freq(word, doc):
    return doc.count(word)
def word_count(doc):
    count=0
    return len(doc)
def tf(word, doc):
    return (freq(word, doc) / float(word_count(doc)))
def intersection(lst1, lst2):
    lst3 = [value for value in lst1 if value in lst2]
    return lst3
def union(lst1, lst2):
        lst3=lst1+lst2
        lst3=list(set(lst3))
        return lst3

f=open('SMTeuroparl.train.tsv',encoding='utf-8')
txt=f.readlines()
tfscore=[]
out=open('TF_Similarity_Output.tsv','w+',encoding='utf-8')
for i in txt:
        i=i.split('\t')
        source_doc=i[1]
        target_doc=i[2]
        source_doc=[c for c in source_doc if c not in
        string.punctuation]
        source_doc=''.join(source_doc)
        target_doc=[c for c in target_doc if c not in
        string.punctuation]
        target_doc=''.join(target_doc)
        #saving sentence as a list
        source=source_doc.split()
        #saving sentence as a list
        target=target_doc.split()
        source_target=union(source, target)
        s1=[]
        s2=[]
        for i in source_target:
                s1.append(tf(i,source_doc))
                s2.append(tf(i,target_doc))
        print(len(s1),len(s2))
        csim=round((1 - spatial.distance.cosine(s1,s2))*5,4)
        out.write('gold score: '+str(csim)+'\t'+target_doc)
out.close()
```

Appendix III

TF-IDF Similarity

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
import string
import numpy as np
from scipy import spatial

f=open('./2012/SMTeuroparl.train.tsv',encoding='utf-8')
txt=f.readlines()
tfscore=[]
out=open('TFIDF_Similarity_Output.tsv','w+',encoding='utf-8')
for i in txt:
    i=i.split('\t')
    source_doc=i[1]
    target_doc=i[2]
    source_doc=[c for c in source_doc if c not in
    string.punctuation]
    source_doc=''.join(source_doc)
    target_doc=[c for c in target_doc if c not in
    string.punctuation]
    target_doc=''.join(target_doc)
    texts=[source_doc,target_doc]
    tfidf = TfidfVectorizer()
    features = tfidf.fit_transform(texts)
    scores=np.array(pd.DataFrame(features.todense(),columns
    =tfidf.get_feature_names()))
    csim=round((1 - spatial.distance.cosine(scores[0],
    scores[1]))*5,4)
    out.write('gold score: '+str(csim)+'\t'+target_doc)
out.close()
```

Appendix IV

DocSim

```
import numpy as np
class DocSim(object):
    def __init__(self, w2v_model , stopwords=[]):
        self.w2v_model = w2v_model
        self.stopwords = stopwords

    def vectorize(self, doc):
        """Identify the vector values for each word in the given
        document"""
        doc = doc.lower()
        words = [w for w in doc.split(" ") if w not in
        self.stopwords]
        word_vecs = []
        for word in words:
            try:
                vec = self.w2v_model[word]
                word_vecs.append(vec)
            # Ignore, if the word doesn't exist in the vocabulary
            except KeyError:
```

```
            pass

        # Assuming that document vector is the mean of all the
        # word vectors
        # PS: There are other & better ways to do it.
        vector = np.mean(word_vecs, axis=0)
        return vector

    def _cosine_sim(self, vecA, vecB):
        #Find the cosine similarity distance between two vectors.
        csim = np.dot(vecA, vecB) / (np.linalg.norm(vecA) *
                   np.linalg.norm(vecB))
        if np.isnan(np.sum(csim)):
            return 0
        return csim

    def calculate_similarity(self, source_doc, target_docs=[],
    threshold=0):
        """Calculates & returns similarity scores between given
        source document & all the target documents."""
        if isinstance(target_docs, str):
            target_docs = [target_docs]
        source_vec = self.vectorize(source_doc)
        results = []
        #i=0
        for doc in target_docs:
            target_vec = self.vectorize(doc)
            sim_score = self._cosine_sim(source_vec, target_vec)
            gold_score=sim_score*5
            if sim_score > threshold:
                results.append({
                    'gold score' : gold_score,
                    'doc' : doc
                })
            #i+=1
            # Sort results by score in desc order
            results.sort(key=lambda k : k['gold score'] ,
            reverse=True)
        results=str(results).strip('[{').strip('}]').split('}, {')
        return results
```

Word2Vec Similarity

```
from gensim.models.keyedvectors import KeyedVectors
from DocSim import DocSim
import string

# Using the pre-trained word2vec model trained using Google
# news corpus of 3 billion running words.

googlenews_model_path = 'F:\DP Sir\Python Code\Document Similarity
# using Word2Vec\data\GoogleNews-vectors-negative300.bin'
stopwords_path = "F:\DP Sir\Python Code\Document Similarity
# using Word2Vec\data\stopwords_en.txt"
#loading Google pre-trained w2v model
model = KeyedVectors.load_word2vec_format(googlenews_model_path,
        binary=True)
with open(stopwords_path, 'r') as fh:
    stopwords = fh.read().split(",")
ds = DocSim(model,stopwords=stopwords)  #removing stopwords
```

```
f=open('./2012/SMTeuroparl.train.tsv',encoding='utf-8')
txt=f.readlines()

sim_scores=[]

for i in txt:
    i=i.split('\t')
    source_doc=i[1]
    target_doc=i[2]
    #removing punctuation marks
    source_doc=[c for c in source_doc if c not in
    string.punctuation]
    source_doc=''.join(source_doc)
    target_doc=[c for c in target_doc if c not in
    string.punctuation]
    target_doc=''.join(target_doc)
    #appending similarity score of s1 & s2 for each
    document in a list
    sim_scores.append(ds.calculate_similarity(source_doc,target_doc))
print(sim_scores)
print('done')
out=open('Word2Vec_output.tsv','w+',encoding='utf-8')
for i in sim_scores:
        out.write(str(i)+'\n')
out.close()
```

Bibliography

[1] MS Windows NT kernel description. https://nlp.stanford.edu/IR-book/pdf/irbookprint.pdf. Accessed: 2010-09-30.

[2] MS Windows NT kernel description. https://arxiv.org/pdf/1402.3722.pdf. Accessed: 2010-09-30.

[3] MS Windows NT kernel description. https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa. Accessed: 2010-09-30.

[4] MS Windows NT kernel description. https://hackernoon.com/finding-the-most-important-sentences-using-nlp-tf-idf-3065028897a3. Accessed: 2010-09-30.

[5] MS Windows NT kernel description. https://medium.freecodecamp.org/how-to-process-textual-data-using-tf-idf-in-python-cd2bbc0a94a3. Accessed: 2010-09-30.

[6] MS Windows NT kernel description. https://arxiv.org/pdf/1411.2738.pdf. Accessed: 2010-09-30.

[7] MS Windows NT kernel description. https://github.com/brmson/dataset-sts/blob/master/data/sts/semeval-sts/2012/SMTeuroparl.train.tsv. Accessed: 2010-09-30.