

## ML 24/25-09 Semantic Similarity Analysis of Textual Data

Senthil Arumugam Ramasamy  
[senthil.ramasamy@stud.fra-uas.de](mailto:senthil.ramasamy@stud.fra-uas.de)

Neethu Ninan  
[neethu.ninan@stud.fra-uas.de](mailto:neethu.ninan@stud.fra-uas.de)

**Abstract**—*In the era of big data and natural language processing, accurately analyzing and comparing textual data is paramount. This research paper presents a comprehensive framework for semantic analysis of textual data; words, phrases and/or documents, using open AI embedding techniques and Cosine similarity algorithm. The framework is implemented as a software tool that preprocesses text, generates embeddings and calculates similarity scores for phrases and documents, supporting various preprocessing options, including tokenization, normalization and context-based adjustments. This means a robust model for contextually relevant similarity measurements. Visualization tools are included to map scalar values against similarity scores and display a clear view of data distribution and similarity metrics. A secondary plot displays the number of possible comparisons between documents and their corresponding similarity score which displays the semantic analysis between documents or phrases based on a pre-defined threshold, highlighting its relevance. The approach indicates improved accuracy in similarity analysis over traditional methods, and contributes to the field of natural language processing with several potential applications including automated content categorization like resume filtering or filtering admission for students based on the requirements. The software tool is made available as an open-source project, encouraging further research and development for potential use cases.*

**Keywords**—*Embedding, Cosine Similarity, Scalar Values*

### I. INTRODUCTION

As per the statistics, billions of brief text messages are sent on social media every day; of which nearly every tweet is between one and thirty words long. Appropriate information retrieval methods are required in order to access this stream of really brief text fragments. There are varieties of methods available to analyze the meaning of a text. Lexical-Based Similarity (LBS) analysis is the conventional one that relies on calculating the separation between two chains in order to identify their similarities. Character-based and term-based distance measurements are the two categories into which LBS measurements fall. Character-based metrics is used to address typographical problems. Nevertheless, these metrics fail to capture the resemblance with term arrangement problems. Term-based similarity metrics attempt to address this problem. Rather than relying on just character matching, semantic similarity determines how similar two sequences are based on their importance. It is regarded as a possible component of tasks related to Natural Language Processing (NLP), including machine translation, entailment, text summarization, and word sense disambiguation. In order to explore the underlying contextual meaning, it is typically necessary to compute the similarity. Finally, the most expensive one is hybrid-based similarity method; here both lexical-based and semantic-based similarity metrics are combined. It requires a lot of computational stages and hence comparatively slow way of analysis. Despite being costly to implement, it is used in applications where very high accuracy is required [1].

Semantic similarity is a key component of Natural Language Processing (NLP) and one of the core tasks for many NLP applications and related fields. Semantic similarity, as opposed to the lexicographical similarity or statistical similarity mentioned above is a metric that is defined over a collection of documents or phrases. It is based on the concept of distance between objects on how similar their meanings or semantic content are. Also, similarity between the documents is based on the direct and indirect relationships among them, which can be measured and identified by the presence of semantic relations among them. In the realm of Natural Language Processing (NLP), estimating the semantic similarity between text data is one of the most difficult and unresolved research challenges. It is challenging to create rule-based techniques for calculating semantic similarity metrics due to the flexibility of natural language. Numerous semantic similarity techniques have been put out over time to address this problem [2]. To be precise, semantic similarity, which quantifies how closely two pieces of text align in meaning, is a fundamental concept in natural language processing (NLP) with applications in information retrieval, document clustering, and recommendation systems. By utilizing OpenAI's powerful embeddings, it is possible to turn text into dense vector representations that capture its semantic essence, allowing to compute similarity using functions like Euclidean, Jaccard or Cosine functions. In this paper, we have designed a systematic approach to investigate and measure the semantic relationships between textual data at different levels, ranging from individual words and phrases to entire documents.

### II. METHODS

#### A. Literature Review:

In the study proposed by Majumder et al [2], semantic analysis of textual data aims to extract valuable insights from text by understanding the underlying structure and meaning of words, phrases, and texts. This process has become more important for applications such as retrieving information, document classification, sentiment analysis, and natural language understanding. Many approaches and frameworks have evolved throughout time to improve semantic analysis. The emergence of transformer-based models such as GPT has further speed up progress in this field. However, this sector continues to benefit greatly from a variety of modern and old methodologies. This section is designed to give a brief idea about the existing methods where both traditional and distributional are covered and also, a smooth transition to OpenAI's GPT model, later a theoretical estimation about Distant Metrics.

#### 1. Classical Methods for Semantic Analysis

One fundamental method that uses statistical calculations to identify connections between words and documents is called Latent Semantic Analysis (LSA). LSA is a natural language processing technique, which is originally developed for Information Retrieval, that generates a set of concepts associated with a set of documents

and terms by analyzing the links between the documents and terms. It facilitates the discovery of the data's underlying latent semantic structure. Here, a large dataset is analyzed to choose some relevant documents on the basis of given query. Singular Value Decomposition (SVD) is used to break down a term-document matrix, lowering the dimensionality and emphasizing the latent structure in the data [3]. Its dependence on a linear translation, however, restricts its capacity to represent intricate contextual connections among words. In contrast to LSA, Explicit Semantic Analysis (ESA) creates high-dimensional representations of text by using structured knowledge sources such as Wikipedia [4]. ESA can effectively determine semantic relatedness by mapping text to a concept space. While ESA benefits from leveraging external expertise, its effectiveness mostly depends on the depth of the underlying knowledge base.

## 2. Distributional Semantic Models

With word embeddings, a significant breakthrough in capturing the semantic links between words was achieved. In order to produce dense vector representations, Word2Vec and GloVe [5][6] evaluate the context of words in large datasets. Although these models are successful in capturing semantic similarity, they are limited by their inability to accurately represent context-dependent interpretations and polysemy.

## 3. Transformer-based Semantic Models

Since the introduction of transformer systems, semantic analysis has changed. By considering the context in which words appear, models such as BERT (Bidirectional Encoder Representations from Transformers), Open AI's GPT (Generative Pretrained Transformer), RoBERTa (Robustly Optimized BERT Pretraining Approach); provide contextual embeddings that improve on previous methods [8]. Sentence embeddings are designed to give full phrases or documents a meaningful vector space representation. To generate sentence embeddings that score well on similarity and clustering tests, a transformer-based architecture known as Sentence-BERT (SBERT) [7] was introduced. Another method, known as DeCLUTR (Deep Contrastive Learning for Unsupervised Textual Representations) creates unsupervised sentence representations through contrastive learning that exhibit remarkable performance in several downstream tasks. In comparison to traditional word embeddings, these models provide context-aware representations.

Transformer-based models such as SBERT, have demonstrated exceptional performance in a variety of natural language processing tasks, including text summarization, sentiment analysis, and question answering. An efficient method for producing superior sentence embeddings is contrastive learning. DeCLUTR (Deep Contrastive Learning for Unsupervised Textual Representations) is the most prominent one in that which creates unsupervised sentence representations through contrastive learning that exhibit remarkable performance in several downstream tasks. In comparison to traditional word embeddings, these models provide context-aware representations. DeCLUTR performs well on problems involving similarity by using contrastive learning to unsupervised textual representations. When labelled data is hard to come by or unavailable, this method is especially helpful [13].

## B. Evolution to OpenAI's GPT Model for Semantic Analysis:

In our project we have utilized the Open AI's GPT model to enhance semantic analysis procedure in a flexible and robust way. This transition from distributional and classical approaches to transformer-based designs such as GPT is due to the following

reasons and this, in turn represents a paradigm change in semantic analysis.

- LSA and ESA cannot capture contextual meaning because they rely on statistical connections instead of true semantic understanding.
- Word Embeddings (Word2Vec, GloVe) generate static embeddings that guarantee that words are vectorially represented consistently across contexts. Accurate analysis of polysemy is impossible due to this limitation.
- Because Sentence Embeddings (SBERT, DeCLUTR) are context-aware, they require extensive training and fine-tuning optimizations, which can be resource-intensive.

The meaning of words, sentences, and texts is accurately captured via dynamic, context-sensitive embeddings created using an OpenAI model based on GPT. With just a basic preprocessor module, these embeddings can be used straight away without a lot of fine-tuning or training. Additionally, because GPT embeddings are built using the complete context of the text, they effectively capture polysemous meanings. The substantial pre-training on text enables the model to increase its generalization across other domains, providing strength and versatility. Numerous methods have been used to approach semantic analysis, each with its own advantages and disadvantages. Conventional methods such as LSA and ESA are simple and intuitive, but they lack the contextual sensitivity of embedding-based methods. Word and phrase embeddings provide more meaningful representations, but they struggle with polysemy. Transformer-based models solve many of these problems, despite the fact that they may require a lot of resources. Importantly, Open AI's GPT is not free, it is charged based on the API calls per number of tokens. So, in order to use the embedding model appropriately, we need to ensure the token is optimized such that smaller text requires only fewer tokens.

An embedding is a list of floating-point numbers that is vectorized. To be precise, it is a series of numbers that represent the ideas in content, like code or natural language. Machine learning models and other algorithms can hence easily comprehend the connections between content and carry out tasks such as retrieval or clustering based on these embeddings. They represent a variety of input formats that machine learning models can interpret, including text, photos, and audio. Tokenizing text into tokens is the first step an AI model takes after receiving text input. After that, each token is transformed into its matching embedding. They power various retrieval augmented generation (RAG) developer tools and applications such as knowledge retrieval in ChatGPT and the Assistants API. Two vectors' distance from one another indicates how related they are. Large distances indicate low relatedness, while small distances indicate high relatedness. [9][10]. In the paper, embedding model is designed in such a way to receive the preprocessed data as shown in the block diagram above in the introduction part.

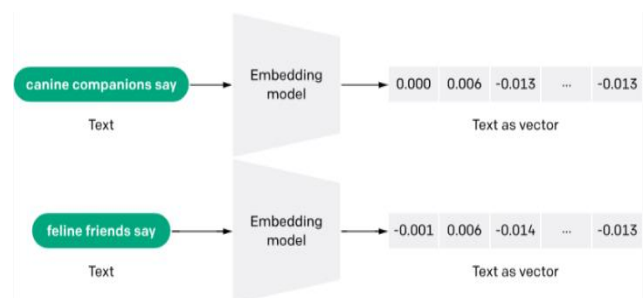


Fig 1: Conversion of Textual Data into Embeddings

In the above Fig 1, it is shown how different words/phrases are mapped into a high dimensional vector space. An embedding model turns each text into vectors. For instance, the short phrase "anatine amigos" is transformed into a single, large vector (e.g., in Fig 1; 1536 dimensions as text-embedding-ada-002 is used), with each dimension capturing a distinctive aspect or feature of the text's meaning. The goal is to represent the text's semantic meaning in a multi-dimensional space, where similar phrases have closer vectors. This process is crucial for semantic similarity analysis, where the closeness of the vectors indicates semantic similarity. It should be noted that, instead of being chosen at random, these vectors are intended to encode the text's meaning such that related sentences will have similar vectors.

Consider the JSON response from the OpenAI API for generating embeddings shown below:

```
{
  "object": "list",
  "data": [
    {
      "object": "embedding",
      "index": 0,
      "embedding": [
        -0.006929283495992422,
        -0.005336422007530928,
        -4.547132266452536e-05,
        -0.024047505110502243
      ]
    }
  ],
  "model": "text-embedding-3-small",
  "usage": {
    "prompt_tokens": 5,
    "total_tokens": 5
  }
}
```

In the above shown JSON response by using text-embedding-3-small, it is given that the "prompt\_tokens" is 5. This shows how many tokens were used in the given input text after the model processed it. Tokens are nothing but pieces of characters or words. For instance in the above Fig 1; "canine companions say" "canine, companions and say" are three tokens that could be separated from the phrase "canine companions say." Nevertheless, the model may further split it if it employs a method called sub-word tokenisation, such as:

```
"canine" → ["ca", "nine"]
"companions" → ["comp", "anions"]
"say" → ["say"]
```

Similarly, "total\_tokens" is also 5. This indicates how many tokens the model has handled overall. When it comes to embeddings, "total\_tokens" and "prompt\_tokens" are typically equal because here we are not producing more text, instead

only a vector representation of the input text is being encoded. These two concepts are very important in terms of cost calculation and management, as OpenAI generate bills according to how many tokens it processes. and also for performance tracking. Therefore, if divided into smaller parts, the phrase "Anatine amigos" may have a total token count of 5.

OpenAI provides two robust third-generation embedding models, which are indicated by the model ID ending with 3.

MODEL	- PAGES PER DOLLAR	PERFORMANCE ON MTEB EVAL	MAX INPUT
text-embedding-3-small	62,500	62.3%	8191
text-embedding-3-large	9,615	64.6%	8191
text-embedding-ada-002	12,500	61.0%	8191

Fig 2: III Generation Models vs II Generation Model

From the Fig 2, the embedding model is not free. There is always a token limit per users. An incredibly effective embedding model, text-embedding-3-small, is a major improvement over the text-embedding-ada-002 model, which was launched in December 2022. Additionally, text-embedding-3-small is far more effective than our text-embedding-ada-002 model from the previous generation. Also, text-embedding-3-small's price has been lowered from \$0.0001 per 1,000 tokens to \$0.00002, a 5X reduction from text-embedding-ada-002. Apart from this the new, next-generation larger embedding model, text-embedding-3-large, can produce embeddings up to 3072 dimensions in size. The price of text-embedding-3-large is set at \$0.00013 per 1,000 tokens. However, while choosing embedding model, the size of the model should be selected appropriately. Larger embeddings (such as text-embedding-3-large, which has dimensions of 3072). This gives text representations that are more precise and thorough and also improved performance on challenging NLP tasks such as text retrieval and document similarity. But, this requires more compute, memory, and storage expenses. While embeddings that are smaller (such as text-embedding-3-small or reduced versions of larger embeddings) is quicker and works at a lower cost for generating embeddings. Additionally, reduce the amount of memory and storage. This can be avoided by passing a "dimensions" parameter to the model, which can remove some numbers at the end of vectors still maintain the accuracy [9]. This dimensions API parameter allows developers to specify the desired embedding size, hence optimized usage.

When working with natural language and code, embeddings are helpful since they are easily absorbed and compared by various machine learning models and algorithms, such as search or clustering. Semantically related embeddings are likewise numerically similar. In the below figure, when it comes to "canine companions say," for instance, the embedding vector will resemble "woof" more than "meow." Assume, a dimension is represented by each box with floating-point integers, and each dimension is associated with a characteristic or quality that may or may not be understandable by humans. While more complex data models may contain tens of thousands of dimensions, large language model text embeddings usually have a few thousand. Due to the similarities and variations in the meaning of the two words, certain of the dimensions of the two vectors in the example above are comparable, while other dimensions are different [11]. However, the text-embedding-3-large approach provides superior contextual management for more precise similarity evaluations and enhances semantic understanding by efficiently recording intricate relationships between words, phrases, and documents. Because it produces high-quality, high-dimensional embeddings, similarity detection may be done with more granularity. Text classification, clustering, semantic search, recommendation systems, and sentiment analysis are just a few of the NLP activities that it can be used for. Across several areas, the paradigm is extremely scalable and appropriate for both brief phrases and lengthy publications. Because of its deeper language comprehension and enhanced contextual awareness, it is especially effective at recognising associations that go beyond mere keyword matching. Furthermore, when compared to previous models, it performs better when evaluating semantic similarity and easily

interfaces with a variety of frameworks and APIs to create reliable NLP applications. That is, unlike other similarity methods a semantic based approach is more accurate in giving the contextual meaning rather than simply word matching, at the same time can be implemented with less complexity.

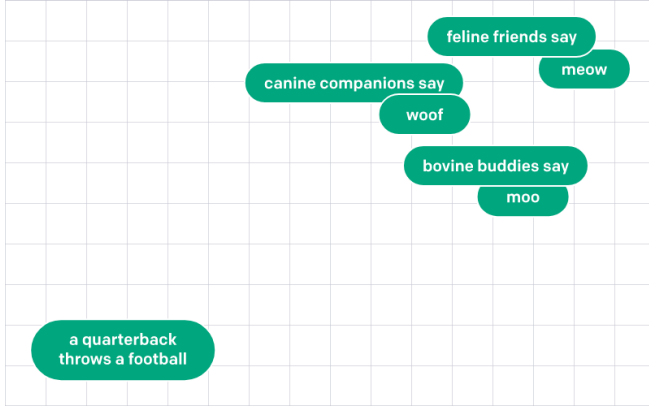


Fig 3: Sample Representation of texts in vector space

The above illustrated Fig 3, shows the spatial proximity of comparable vectors and contrasts them with significantly dissimilar vectors. The depth and variation of similarity can be estimated by using a suitable distant function. However, it is demonstrated by assuming words instead of vectors for a better understanding.

### C. Similarity Analysis- A theoretical estimation about Distant Metrics

The degree to which two objects are similar is measured by their similarity. In the context of data mining, a similarity measure is a distance whose dimensions correspond to the attributes of the objects. Two items are quite similar if they are close together, as well as a low degree of similarity if they are far apart. Mathematical formulas called distance functions are used to quantify how similar or dissimilar two vectors are. The Manhattan distance, Euclidean distance, cosine similarity, and dot product are typical examples. In order to ascertain the degree of relationship between two pieces of data, these metrics are essential.

A distance function or metric is a function  $d(x,y)$  that uses a non-negative real number to quantify the distance between a set's elements. Under that particular measure, the items are equal if the distance is 0. Thus, distance functions give us a mechanism to quantify the proximity of two elements, which can be vectors, matrices, or any other kind of object. Distance functions are frequently employed in optimization problems as cost or error functions that need to be minimized. The commonly used distance functions are Dot product, Minkowski Distance, Euclidean Distance and Cosine Distance. We are comparing the generated embeddings can then be compared using cosine similarity, which yields a textual similarity metric. For tasks like text clustering, semantic search, and document retrieval, this is essential. Cosine similarity is a statistic that quantifies the degree of similarity between two vectors. To determine this similarity, the cosine of the angle between the vectors, which ranges from -1 to 1 is measured. With high cosine similarity signifying that the texts are similar, these vectors frequently represent text data in the context of natural language processing.

The cosine similarity between two vectors, A and B, can be expressed mathematically as: [12]

Cosine Similarity function is given by ,

$$\cos \theta = \frac{A \cdot B}{\|A\| \|B\|}$$

Where:

- A and B are the vectors and A.B is the dot product
- $\|A\|$  and  $\|B\|$  are the magnitude of vectors A and B.

Cosine similarity is used in our implementation mainly because of two reasons:

a) *Normalization*: The vectors are automatically normalised by cosine similarity, which yields a more reliable similarity metric that depends only on the angle between the vectors rather than their lengths.

b) *Magnitude Sensitivity*: It is independent of vector magnitude, in contrast to the dot product and Euclidean distance. Because of this, it is especially helpful for comparing text embeddings, where the direction of the vectors is more important than their length.

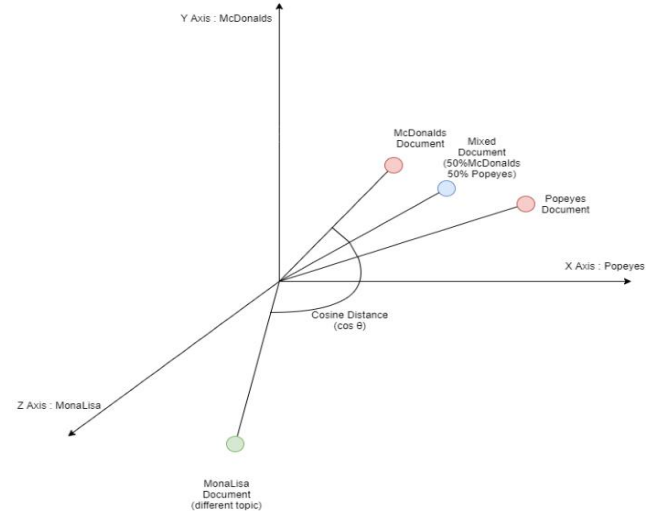


Fig 4: Sample Cosine similarity results

As shown in the Fig 4, the cosine similarity calculated for the four documents. To better illustrate the mechanism, we look at three documents namely Popeys, McDonald's, and a combination of the two, with a document on a different subject (MonaLisa, for example). The last document pertains to a distinct subject but the first two are near to one another because they both deal with the same topic. Here, a three-dimensional space is taken but in reality, the vector space is high dimensional and the texts will be vectors.

### III. IMPLEMENTATION

Based on the literature review and the concepts of OpenAI embedding in semantic analysis, the application is designed to support methods for analyzing data using both phrases and documents. It was understood that when performing analysis using various datasets, such as documents or phrases, the analysis results may vary due to the contextual meaning of the sentences within the documents. During the initial study, several samples were tested with OpenAI Embedding APIs using the NuGet package available in .NET. Additionally, an attempt was made to utilize another tool



known as the Hugging Face API. However, it was discovered that due to the restrictions in the availability of its APIs within the NuGet package, the research was directed towards focusing solely on the OpenAI Embedding technique. This technique was chosen for its availability and its capability to create embeddings that support contextual analysis for different types of datasets, including phrases/words and document comparisons.

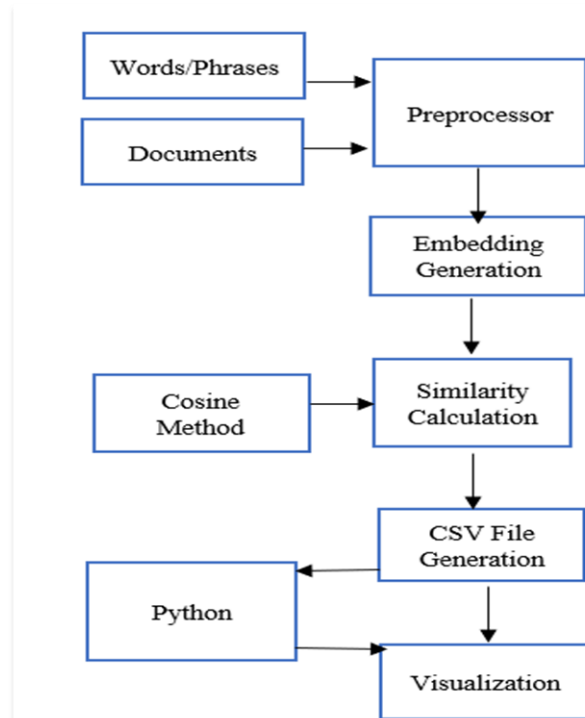


Fig 5: Simplified Block Diagram

As shown in Fig 5, the embedding module we used is the OpenAI's GPT(Generative Pre-training Transformer) based model in order generate embeddings based on respective context. To maintain maximum system accuracy, we have also incorporated a preprocessing module and the raw inputs are first fed to this preprocessor, which minimizes or compresses the words or tokens, still maintaining the context. Tokens are a common term used in the field of NLP. In its simplest form, a token is defined as the pieces of texts. So, by including a preprocessor module, an optimized token usage is guaranteed, since GPT is not free. Once, the embedding is generated, the system performs similarity analysis to determine how different textual elements relate to one another in terms of meaning, context, and domain. The calculated score is later written into a CSV file to generate the visualization for better understanding.

A preprocessor is designed in such a way that it performs initial operations on raw data, including conversion of upper case into lower case, removal of html/url tags if any, stemming/lemmatization focuses on converting words back to its root form), stop word and special character removal and normalization (eg: can't to cannot). We have opted for Cosine method for similarity calculation rather than any other existing metric function, as the proposed system does not need any normalization. Also, an external Python module was utilized to successfully generate the visualization of the similarity scores. Python was chosen for this purpose due to its simplicity and availability of strong visualisation libraries that are not directly available in C#, such as Matplotlib, Seaborn and Plotly. It provides convenience in generating plots as it provides various customization tools for optimizing the plot with minimal initial setup, making it a suitable choice for visualizing the results effectively. The system

also provides a modular structure, making it suitable for future modifications.

To achieve better comparison results, the implementation process was categorized into six main categories:

- Defining Datasets Based on Specific Domains and Processing Datasets
- Designing a Pre-Processing Interface to Process Documents
- Creating Embeddings for the Input Documents
- Calculating Similarity Using the Cosine Similarity Algorithm Based on Programmatically Generated Embeddings
- Generating the Output of Similarity Scores as a CSV File
- Utilizing the Output Data from the CSV File to Generate Meaningful Results Showing Semantic Analysis Between Documents

## 1. Creation of Datasets

During the initial research, it was decided that meaningful analysis could be achieved by classifying datasets by domains. Sets of 5 to 10 words from each domain were identified. For example, "Electricity" and "Energy" are two different words, but contextually, they are related to the *Power Sector* domain. It was intended to verify whether comparison using the OpenAI Embedding technique would provide meaningful results by considering contextual relevance. Similarly, around fifty to sixty different words from the same and different domains were selected to utilize for comparison. The data was also intended to remain editable by developers, enabling future analysis with modified datasets. Initially, phrase comparison datasets were formatted as JSON files, which could be modified by developers as required.

The application was also designed to support the comparison of documents to show contextual relevance between them. For any comparison to be performed, source documents and target documents were required. For example, *JobRequirement.txt* was treated as the source document, while *Job Profile A* and *Job Profile B* were treated as the target documents.

When the source and target documents were compared, meaningful results could be obtained. It was also intended to allow the application administrator of any organization utilizing the application to manage a predefined threshold for comparison.

Furthermore, additional meaningful document comparisons were designed by enabling users of the application to add more documents either directly through a file manager or by integrating a user-specific UI to upload documents. Though such integration requires additional development efforts, it was ensured that the application supports document comparison dynamically, making it more usable and researchable.

## 2. Pre-Processing Interface to Process the Documents:

With the focus on processing any documents or datasets, we first need the data to be loaded into the program, we are achieving this by defining the specific path inside the solution by defining the data folder with the two different folder names.

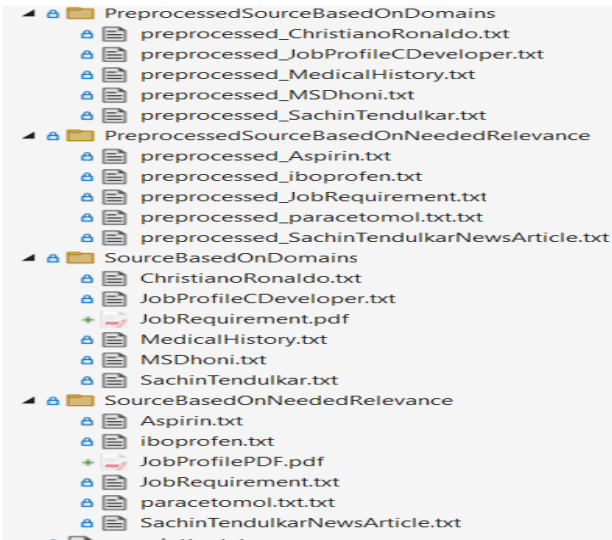


Fig 6: Folder Structure -Raw Documents and Preprocessed Documents

As illustrated in Fig 6, all documents that require comparison are stored in a folder named "SourceBasedOnDomains." This folder is intended to contain all the essential documents based on the specified criteria. Another folder named "Source Based on Relevance" is utilized to hold documents intended for comparison. If this application is integrated with a user interface or user-facing application in the future, users will be able to upload, for instance, their motivation letters or resumes for university admission. The application will then compare this uploads with predefined documents such as "Admission Requirements for Specific Courses in a University" or "Job Requirements." To ensure effective categorization of documents, the "SourceBasedOnDomains" folder must be predetermined according to the required criteria.

The interface "IPreprocessor" has been designed to include all methods necessary for performing the basic functionalities described in the introduction. These methods are responsible for reading raw input datasets/documents/phrases and generating processed documents within a newly created folder, with the folder name generated programmatically. As depicted in Fig. 6, two new folders have been successfully created by appending the word "Preprocessed" to the original folder names, resulting in "PreProcessedSourceBasedOnDomains" and "PreProcessedSourceBasedOnRelevance."

The following asynchronous method calls are utilized to process the text files in the specified folders:

```
await ProcessTextFilesInFolderAsync(textPreprocessor,
sourceDomainsFolder, outputDomainsFolder);
```

```
await ProcessTextFilesInFolderAsync(textPreprocessor,
sourceRelevanceFolder, outputRelevanceFolder);
```

Once the documents are loaded into the program and the processed documents are generated, the method definition provided below is applied:

```
String PreprocessText (string text, TextDataType type);
```

Subsequent methods, which will be discussed later in this paper, can then be invoked to generate embeddings and calculate similarity scores.

### 3. Create Embedding's for the input documents:

The interface CalculateEmbeddingAsync was created to accept phrases or documents as text. Additionally, it was designed to identify the category of the domain to determine its relevance to the context in which it is created. Therefore, the interface was structured to accept *text1* and *text2* for processing, where the source document or phrase is provided in *text1*, and the target document or phrase is provided in *text2*, along with their corresponding filenames.

```
Task<double> CalculateEmbeddingAsync (string text1, string
text2, string fileName1, string fileName2);
```

This interface was intended to be utilized within two different services; SemanticSimilarityForDocumentsWithInputDataDynamic.cs and SemanticSimilarityPhrasesWithInputDataSet.cs. The actual implementation within these services was designed to produce output embeddings by employing the methods available in the OpenAI Embedding NuGet package.

The embeddings are generated through the method GenerateEmbeddingsAsync(), as shown below:

```
OpenAIEmbeddingCollection collection = await
client.GenerateEmbeddingsAsync(inputs);
```

The method GenerateEmbeddingsAsync() was designed to accept a list of strings as inputs and produce a collection of embeddings as output. The range of size 3052 is generated because larger embeddings, such as text-embedding-3-large (which has dimensions of 3072 as mentioned in the literature review), are being utilized.

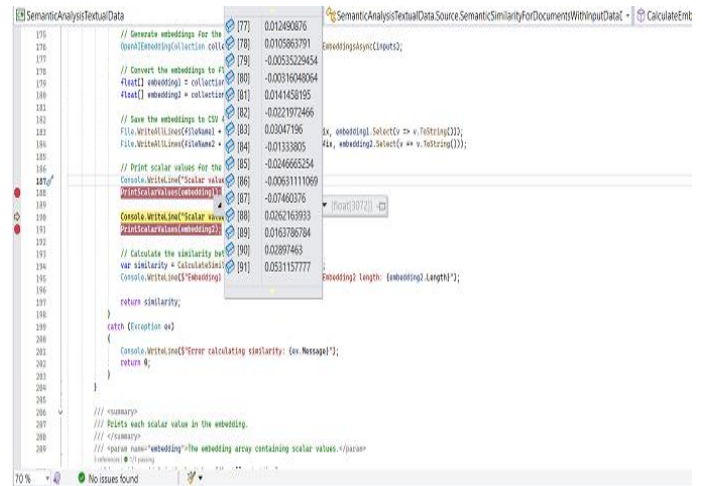


Fig 7: Generated Embeddings in Debug Mode

As shown in Fig 7, custom method named PrintScalarValues() is also created to display the output of each document or phrase as embeddings, that is to print individual values of every array until the end of the collection size.

```
Public static void PrintScalarValues (float[] embedding)
```

This method is developed to print the scalar values of embeddings to assist in visualization. By examining these values, the correlation between the generated similarity score and the scalar values in vector space can be better understood.

#### 4. Calculate Similarity using Cosine Similarity Algorithm based on programmatically generated embeddings of input documents

In order to calculate the similarity from the vector embedding, in our project we are using the method;

CalculateSimilarity (float [] embedding1, float [] embedding2);

This method is designed to accept embeddings generated from the previous GenerateEmbeddingsAsync (inputs) method, which triggers the implementation of the similarity calculation by applying the Cosine Similarity algorithm. The algorithm is capable of returning a single similarity score as a double value, with results ranging from -1 to 1. A value of 1 indicates perfect similarity (identical embeddings), 0 indicates orthogonal vectors (no similarity), and -1 signifies complete dissimilarity (opposite vectors).

The implementation of the CalculateSimilarity process involves several steps. First, Input Validation is performed to ensure that the lengths of the two embeddings are equal. If not, the method returns a score of zero, indicating an invalid comparison. The second step involves Dot Product and Magnitude Calculation, where the dot product between the two embedding vectors is computed, followed by calculating the magnitude of each embedding vector separately using the formula:  $Magnitude = \sqrt{\sum embedding[i]^2}$ .

The third step is Normalization, where the cosine similarity score is derived by dividing the dot product by the product of the magnitudes, using the formula:  $Cosine\ Similarity = \frac{Dot\ Product}{(Magnitude1 * Magnitude2)}$ . This normalization ensures that the similarity score remains within the valid range of -1 to 1. Finally, the implementation includes Error Handling to address scenarios where the magnitude of any vector is zero, which would result in invalid calculations. In such cases, an error is raised, and if any other unexpected errors occur, the method returns a similarity score of zero and logs the error. This structured approach ensures that the similarity calculation is robust, reliable, and capable of handling a wide range of input scenarios.

#### 5. Generate the output of Similarity Score as a CSV File

Application is designed to support the generated output as a CSV file. For saving the generated similarity score after cosine similarity we are using the following methods:

Public static void SaveResultsToCsv (List<DocumentSimilarity> results)

Public static void SaveResultsToCsv (List<PhraseSimilarity> results)

Document Similarity and Phrase Similarity is the domain classes created to support the implementation of saving all the state of different values generated during the processing stage including the similarity score, domain, fileName1, fileName2 and score, domain, Phrase1, Phrase2, context respectively so that we achieve the clear understanding of what data's can be mapped to which data to represent the generated data graphically using visualization methods.

#### 6. Utilizing the output data from CSV to generate meaningful results which shows semantic analysis between documents

Run	Document Key	Document To be Compared	Similarity Score	isPreProcessing Enabled
Run Without Pre Processing For One Sample	JobProfileCDeveloper.txt	JobRequirement.txt	0.70290482	FALSE
Run With Pre Processing For One Sample	preprocessed_JobProfileCDeveloper.txt	preprocessed_JobRequirement.txt	0.70199654	TRUE

Fig 8: Observation From with and Without Pre-Processing

From the Fig 8, it has been observed that preprocessing a single sample of generated embedding scores creates an impact where variations in similarity scores can be detected. These variations occur due to the application of techniques such as stop word removal, lemmatization, and the elimination of articles. Although the initial study was intended to examine the behavior of similarity scoring with and without preprocessing, rather than to improve scores, it was understood that improvements may vary depending on the context and this observation in variation appeared to be so small which gives the confidence of efficient usage of token while utilizing the open AI text GPT model as this change in variation of similarity score is not significant hence can improves the cost saving.

The observation confirmed that OpenAI Embedding effectively achieves contextual alignment, and the results indicate that by applying specific improvements to preprocessing analysis across different use cases, better results can be achieved. These improvements are intended to be the focus of further research and implementation. This preprocessing was conducted to gain insight into the contextual variations caused by small changes in documents, even when those documents discuss the same topic or belong to the same domain. To fulfill the ultimate goal of understanding the semantic analysis of textual data between documents and generating meaningful outputs that correlate with real-time use cases, Python was employed as an external development tool to generate graphical charts. The application was designed to read the output CSV file, which is dynamically generated and placed in the root directory of the Python application.

Currently, the placement of the output CSV file requires manual effort by developers or the application administrator. However, this limitation has been identified, and improvements are planned for later stages, either by the developers themselves or by providing opportunities for other developers to enhance the implementation.

Comparison of Two Word Document Embeddings with Similarity Score

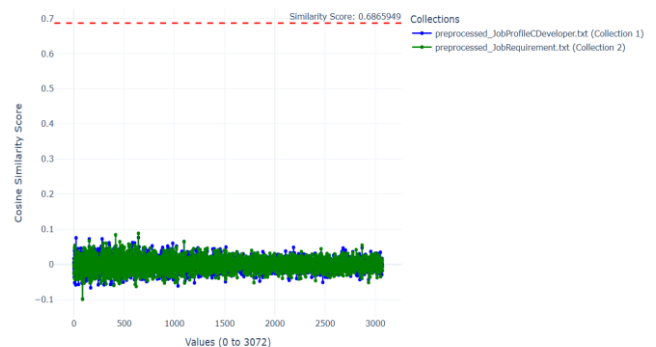


Fig 9: Scalar Values vs. Similarity Score Plot for One Comparison

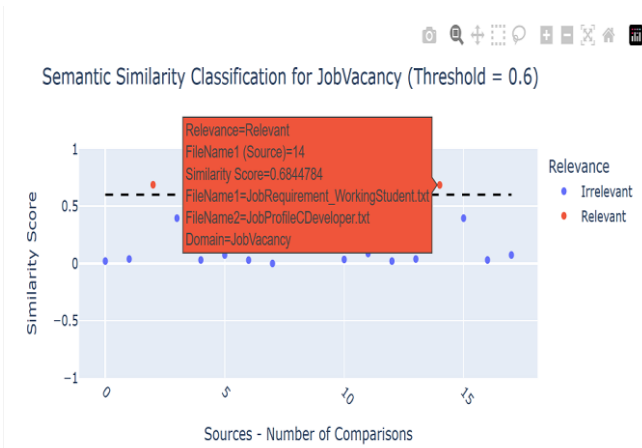


Fig 10: Total Number of Documents Compared VS Similarity Score

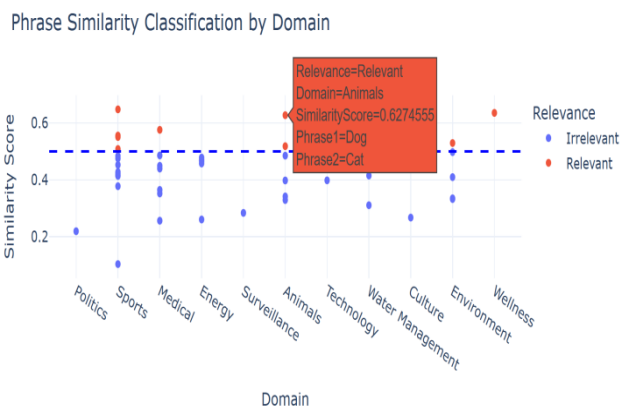


Fig 11: Phrases Comparison (X-axis) vs. Similarity Score (Y-axis)

Three types of plots are designed to support analysis the data distribution. A shown above, Fig 9 was designed to assist developers in understanding how contextual relevance is generated by plotting similarity scores on the Y-axis against scalar values (ranging from 0 to 3052) on the X-axis. The second chart as shown in Fig 10, is created to graphically represent all possible comparisons of documents within the dataset, with the X-axis representing the designed comparisons and the Y-axis showing the corresponding similarity scores. The Third chart as shown in Fig 11 has been plotted to graphically represent all possible comparisons of phrases within the dataset, with the X-axis representing the designed comparisons across different domains and the Y-axis showing the corresponding similarity scores. Also, in Fig 10 and Fig 11, it was intended to display the analysis on a single chart to provide an easier representation of all values. This approach allows users to hover over the blue or red dots, which can be clicked to observe the details associated with them along with their similarity scores.

## 7. Utilizing CSV to Plot Using Python:

As there are numerous options available for exploration, Python was chosen due to its open-source nature and the availability of well-defined libraries for creating scatter plots. After conducting some initial studies, a Python-based Flask application was developed to classify and visualize similarity scores across different domains.

The tool was designed to read CSV files containing similarity scores, which are then mapped to their respective domains, and

predefined thresholds are applied for classification. Plots are generated using Plotly, and thresholds for domains such as "JobVacancy," "Medical," and "Sports" are visualized through interactive scatter plots. Interpretability is enhanced by categorizing similarity scores as "Relevant" or "Irrelevant" based on domain-specific thresholds. Currently, domain name mapping is performed according to the initial application design requirements. This mapping will be continuously refactored as needed by developers or during future analyses to support additional domain/context-related mappings of data, thereby improving visualization quality..

## IV. TESTCASE WITH RESULTS

Several test cases have been implemented to ensure that all code and methods are covered when executed through the Test Explorer. This coverage includes both positive and negative scenarios, such as handling exceptions and null values. Mock data has not been used in the current implementation, as this application serves as the initial setup for analysis purposes.

The application's "data" folder, which was defined for maintaining datasets, has been copied into the test project. These files are read dynamically by the program to successfully maintain code coverage and execute the test cases.

The test framework has been implemented using Microsoft.VisualStudio.TestTools.UnitTesting, and the test cases are executed through Visual Studio Test Explorer to verify code coverage and robustness.

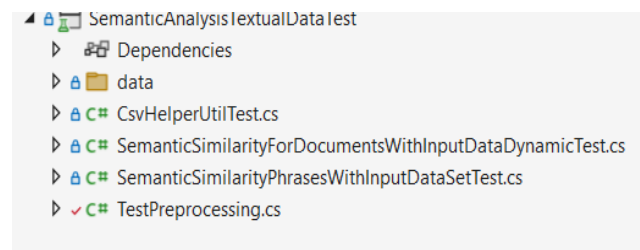


Fig 12: Representation of Semantic Analysis Test Project

The test cases have been separated by creating individual test classes for each defined service, based on their functionalities. Since the business functionalities have been segregated into four classes, corresponding test classes have been created, as illustrated in Fig. 12. The implemented test cases are briefly explained as follows:

- 1) **Test for Preprocessor Method:** This test ensures all methods in the preprocessing module function correctly. It is crucial for confirming accurate embedding generation.
- 2) **Basic Test Framework Validation:** This test verifies the setup of the test framework by creating an instance of the class and performing a simple assertion check.
- 3) **Exception Handling in Document Comparison:** This test sets up directories and files, invokes the CompareDocumentsAsync method, and checks for valid results, ensuring proper exception handling.
- 4) **Validity of Similarity Score Calculation:** Reads content from source and target files, calculates similarity scores using the CalculateEmbeddingAsync method, and confirms that scores are greater than zero.
- 5) **Service Provider Configuration Validation:** Validates the configuration of services returned by the ConfigureServices method, ensuring it matches the expected type.



**6) Source and Target File Retrieval:** Tests the `GetSourceAndTargetFiles` method, ensuring the correct retrieval of source and target files with valid file extensions.

**7) Scalar Value Printing Validation:** Captures and validates console outputs from the `PrintScalarValues` method, ensuring correct scalar value printing.

**8) Accuracy of Similarity Score Calculation:** Evaluates similarity scores between different pairs of embeddings, ensuring correctness.

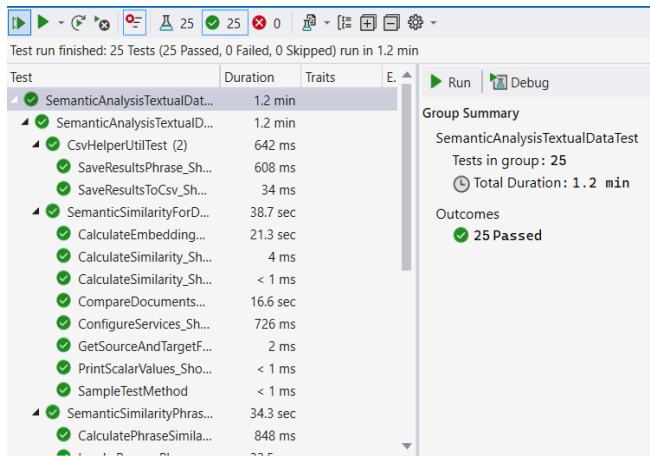
**9) Handling of Different Length Embeddings:** Confirms that the `CalculateSimilarity` method returns zero for embeddings of different lengths.

**10) Handling of Empty Inputs:** Ensures the `CalculateEmbeddingAsync` method returns a similarity score of 0 when given empty input strings, verifying robustness and reliability for edge cases.

**11) Handling of Invalid File Paths:** Validates that the document comparison service properly detects and handles invalid or non-existent file paths by raising appropriate exceptions.

**12) Phrase Processing and Result Saving:** Confirms the correct processing of phrases and saving of results using the `CsvHelperUtilTest` class.

This test method ensures that the `SaveResultsPhrase` method in the `CsvHelperUtil` class correctly saves a list of `PhraseSimilarity` objects to both CSV and JSON files. It verifies that the JSON file is created and contains the correct number of records, while also ensuring the CSV file is created with the correct number of records. Additionally, the test checks whether the content of both files matches the original data. This validation is crucial for confirming the functionality of the `SaveResultsPhrase` method, ensuring it accurately writes data to the specified file formats. Such accuracy is essential for applications that rely on exporting data for further analysis or sharing.



Test	Duration	Traits	E
SemanticAnalysisTextualDataTest	1.2 min		
SemanticAnalysisTextualDataTest	1.2 min		
CsvHelperUtilTest (2)	642 ms		
SaveResultsPhraseToJson...	608 ms		
SaveResultsToCsv_Sho...	34 ms		
SemanticSimilarityForD...	38.7 sec		
CalculateEmbedding...	21.3 sec		
CalculateSimilarity_Sh...	4 ms		
CalculateSimilarity_Sh...	< 1 ms		
CompareDocuments...	16.6 sec		
ConfigureServices_Sh...	726 ms		
GetSourceAndTargetF...	2 ms		
PrintScalarValues_Sho...	< 1 ms		
SampleTestMethod	< 1 ms		
SemanticSimilarityPhras...	34.3 sec		
CalculatePhraseSimila...	848 ms		

Fig 13: Representation of Test Run Results of All Test Cases

The test cases have been successfully implemented as illustrated by Fig. 13, where it is demonstrated that all test cases have been executed, and unit testing has been completed with satisfactory code coverage. This ensures that the implemented methods for the system are robust, hence a finely tuned semantic similarity analysis system.

## V. OVERCOMING LIMITATIONS

*a) Dependency on External Libraries:* The project relies on external libraries such as Plotly.NET and Microsoft.VisualStudio.TestTools.UnitTesting. Any changes or deprecations in these libraries could affect the functionality of the project. In order to overcome, Regularly update and test the project with the latest versions of external libraries like Plotly.NET and Microsoft.VisualStudio.TestTools.UnitTesting. Maintain backward compatibility by implementing adapter patterns or creating wrappers around critical functions. Also include automated tests to verify compatibility whenever library versions are updated.

*b) Text Preprocessing Scope:* The text preprocessing methods implemented may not cover all possible text variations and edge cases. For example, handling of complex HTML tags, nested URLs, or advanced lemmatization might require additional logic. In order to overcome, implement additional preprocessing techniques such as advanced lemmatization, complex HTML parsing, and nested URL handling. Introduce modular preprocessing functions allowing users to customize rules for stop-word removal, tokenization, and other steps.

*c) Performance:* The performance of text preprocessing, especially for large datasets, might be a concern. The current implementation may need optimization for handling large volumes of text efficiently. In order to overcome, developers can use parallel processing techniques and also try to implement batch processing of datasets using memory efficient algorithms.

*d) Customization:* The preprocessing rules (e.g., stop words removal, special characters handling) are hardcoded. Users might need more flexibility to customize these rules based on their specific requirements. In order to overcome, a configuration interface can be developed to allow users to define their own preprocessing rules and similarity thresholds without modifying the core codebase. This interface would provide support for user-defined similarity metrics beyond cosine similarity, making the system adaptable to various research needs. By implementing user-specific configuration files or UI settings, researchers can customize the similarity computation process, including the choice of preprocessing techniques, similarity metrics, and relevance thresholds for different domains. This modular approach will ensure broader applicability and scalability of the framework for diverse NLP applications.

## VI. APPLICATIONS OF SEMANTIC ANALYSIS TEXTUAL DATA

*a) Natural Language Processing (NLP):* This project can be used as a foundational tool for various NLP tasks such as text classification, sentiment analysis, and entity recognition by providing preprocessed and cleaned text data.

*b) Data Cleaning:* The text preprocessing methods can be applied to clean and normalize text data in data science projects, ensuring consistency and improving the quality of the data.

*c) Search Engine Optimization (SEO):* By preprocessing and normalizing text, this project can help in optimizing content for search engines, making it more accessible and relevant.

*d) Content Management Systems (CMS):* The project can be integrated into CMS platforms to preprocess and clean content before publishing, ensuring high-quality and readable content.

e) *Academic Research*: Researchers can use this project to preprocess textual data for various academic studies, including linguistic analysis, social media analysis, and more.

## VII. CONCLUSION:

In conclusion, this research and project explore various topics related to OpenAI embeddings and similarity algorithms in the realm of AI and programming. By leveraging AI libraries, we bridge the gap between theoretical concepts and real-world applications, utilizing real-time data to create useful applications tailored to specific domain requirements. This analysis serves as a foundational step, enabling other developers to build upon it and create a lasting impact. The visualizations demonstrate how texts, documents, and phrases are contextually aligned which can be clearly observed in the results of the application's similarity scores and plots, thus highlighting the potential for various applications. For instance, this application can be adapted for matching job profiles to job requirements or aligning student profiles with university admission criteria, simply by changing the dataset. This flexibility underscores the practical utility of our approach in diverse scenarios.

The proposed framework successfully computes phrase and document similarities while providing visualizations that enhance interpretability. Future enhancements aim to improve the system's usability and applicability by automating CSV file integration for visualization, expanding dataset support across various domains, and enhancing the user interface for document uploads. These improvements will make the framework more versatile and user-friendly, promoting its use in diverse applications. This study contributes to NLP applications in content categorization, job-matching algorithms, and automated document classification, with promising potential for future research.

Overall, this project not only advances our understanding of AI embeddings and similarity algorithms but also provides a versatile tool for real-world applications, paving the way for future innovations.

## VIII. REFERENCES

- [1] A. Aboelghit and T. Hamza, "Textual Similarity Measurement Approaches: A Survey," *The Egyptian Journal of Language Engineering*, vol. 10, 2020, doi: :10.21608/ejle.2020.42018.1012
- [2] G. Majumder, P. Pakray, A. Gelbukh, and D. Pinto, "Semantic Textual Similarity Methods, Tools, and Applications: A Survey," *Computacion y Sistemas*, vol. 20, pp. 647–665, 2016, doi:10.13053/CyS-20-4-2506
- [3] P. Hastings, "Latent Semantic Analysis," 2004. Available: [https://www.researchgate.net/publication/230854757\\_Latent\\_Semantic\\_Analysis](https://www.researchgate.net/publication/230854757_Latent_Semantic_Analysis)
- [4] E. Gabrilovich and S. Markovitch, "Computing Semantic Relatedness using Wikipedia-based Explicit Semantic Analysis," in *Proceedings of the IJCAI International Joint Conference on Artificial Intelligence*, vol. 6, 2007. Available: [https://www.researchgate.net/publication/200042392\\_Computing\\_Semantic\\_Relatedness\\_using\\_Wikipedia-based\\_Explicit\\_Semantic\\_Analysis](https://www.researchgate.net/publication/200042392_Computing_Semantic_Relatedness_using_Wikipedia-based_Explicit_Semantic_Analysis)
- [5] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," in *Proceedings of the 1st International Conference on Learning Representations, ICLR 2013, Workshop Track Proceedings*, Scottsdale, Arizona, USA, May 2–4, 2013. Available: [https://www.researchgate.net/publication/234131319\\_Efficient\\_Estimation\\_of\\_Word\\_Representations\\_in\\_Vector\\_Space](https://www.researchgate.net/publication/234131319_Efficient_Estimation_of_Word_Representations_in_Vector_Space)
- [6] J. Pennington, R. Socher, and C. Manning, "GloVe: Global Vectors for Word Representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, 2014, pp. 1532–1543, Association for Computational Linguistics, doi: 10.3115/v1/D14-1162
- [7] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China, 2019, pp. 3982–3992, Association for Computational Linguistics, doi: 10.18653/v1/D19-1410
- [8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota, 2019, pp. 4171–4186, Association for Computational Linguistics, doi: 10.18653/v1/N19-1423
- [9] OpenAI, "New Embedding Models and API Updates," 2024[Online]. Available: <https://openai.com/index/new-embedding-models-and-api-updates/>.
- [10] Microsoft, "Vector embeddings in Azure Cosmos DB," *Microsoft Learn*, Dec. 3, 2024[Online]. Available: <https://learn.microsoft.com/en-us/azure/cosmos-db/gen-ai/vector-embeddings>.
- [11] OpenAI, "Introducing text and code embeddings," *OpenAI*, Jan. 25, 2022[Online]. Available: <https://openai.com/index/introducing-text-and-code-embeddings>
- [12] D. P. Yadav, N. K. Kumar, and S. K. Sahani, "Distance Metrics for Machine Learning and its Relation with Other Distances," *Mikailsys Journal of Mathematics and Statistics*, vol. 1, no. 1, pp. 15–23, 2023. Available: <https://doi.org/10.58578/mjms.v1i1.1990>.
- [13] J. Giorgi, O. Nitski, B. Wang, and G. Bader, "DeCLUTR: Deep Contrastive Learning for Unsupervised Textual Representations," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Online, 2021, pp. 879–895, Association for Computational Linguistics, doi: 10.18653/v1/2021.acl-long.72.