Information Technology Course
Module Software Engineering
by Damir Dobric / Andreas Pech

Semantic Similarity Analysis of Textual Data

Senthil Arumugam
senthil.ramasamy@stud.fra-uas.de

Neethu Ninan
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 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 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 admission requirements. The software tool is made available as an open-source project, encouraging further research and potential use case.

Keywords—Embedding, Cosine Similarity, Scalar Values

I. Introduction

Billions of brief text messages are sent on social media every da according to statistics, nearly every tweet is between one and thir 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. It was suggested that character-based metrics be 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 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, we can turn text into dense vector representations that capture its semantic essence, allowing us to compute similarity metrics like cosine similarity. 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.

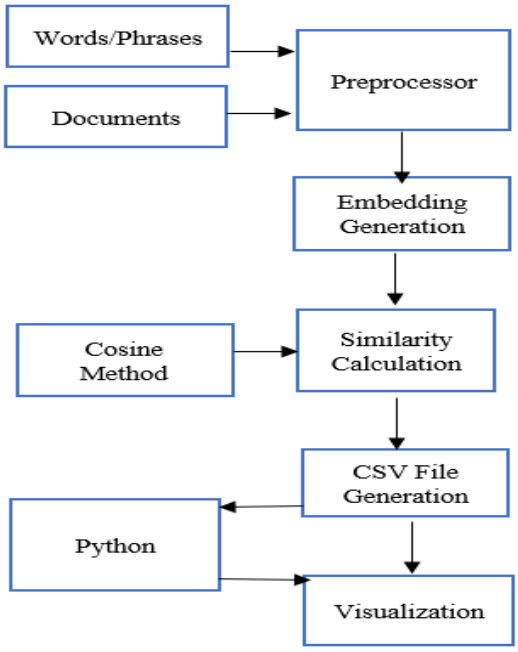

Fig 1: Simplified Block Diagram

As shown in Fig 1, the embedding module we used is the OpenAI's GPT-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 ⟶ 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.

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.

A.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.

A.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.

A.2 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.

i) LSA and ESA cannot capture contextual meaning because they rely on statistical connections instead of true semantic understanding.
ii) 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.
iii) Because Sentence Embeddings (SBERT, DeCLUTR) are context-aware, they require extensive training and fine-tuning

optimisations, 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 generalisation 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 inorder to use the embedding model appropriately, we need to ensure the token is optimized such that smaller text requires only fewer tokens.

B.1 Embedding model:

An embedding is a list of floating-point numbers that is vectorised. 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. Tokenising 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.

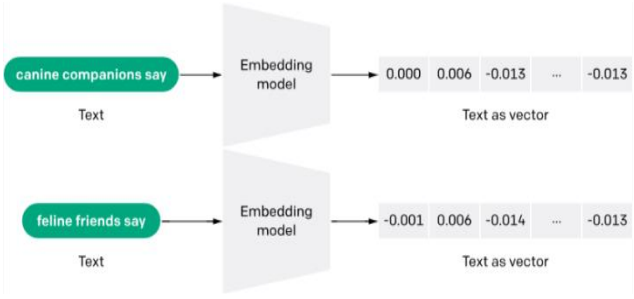

Fig 2: Conversion of Textual Data into Embeddings

In the above figure, shows 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 2; 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,
    -0.005536422007530928,
    -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 3: III Generation Models vs II Generation Model

From the figure, 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.

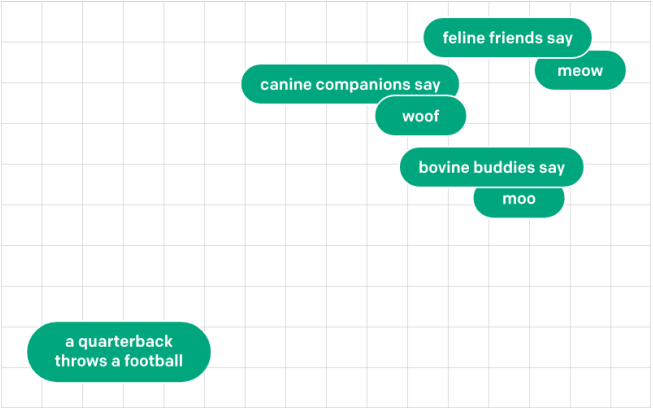

Fig 4: Sample Representation of texts in vector space

This figure illustrates 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. We have demonstrated it 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 ,

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

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.

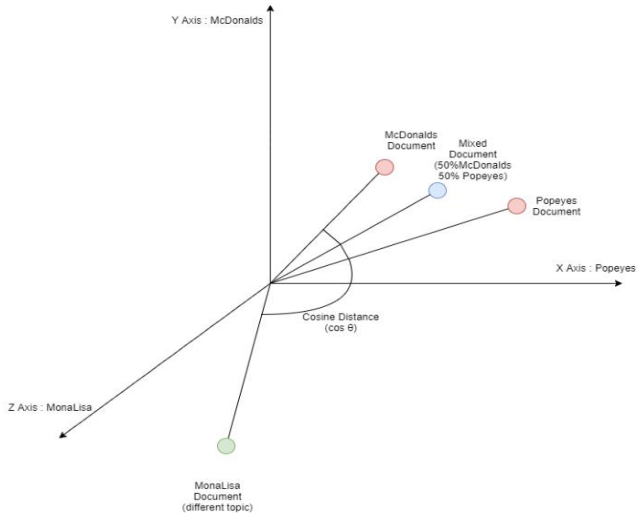

Fig 5: Sample Cosine similarity results

As shown in the fig, 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 concepts of Open AI embedding, Semantic analysis, application was designed in such a way that it supports the methods to do analysis of data using phrases and documents as well; We understood that in order to do such analysis using various datasets such as documents or phrases, analysis results may vary due to contextual meaning of the sentences in the documents, During the Initial Study we tried several samples with Open AI Embedding API's using Nuget package available in the dotnet also we tried one other tool called as Hugging Face API, we found out that due to the restrictions of usage of its API's is not available in Nuget package , research started with the sole focus on using technique of open AI Embedding because of its availability and its ability to create the embedding which supports the contextual analysis for different types of datasets such as Phrases/Words or documents comparisons as well.

In order to achieve the better comparison results, we wanted to categorize the implementation process into 6 main categories such as

1.) Defining dataset's based on specific Domains and processing Datasets
2.) Pre-Processing Interface to Process the Documents
3.) Create Embedding's for the input documents
4.) Calculate Similarity using Cosine Similarity Algorithm based on programmatically generated embedding's of input documents
5.) Generate the output of Similarity Score as a CSV File
6.) Utilizing the output data's from CSV to generate meaningful results which shows semantic analysis between documents
7.)

a.) How did we create Dataset's?

During the Initial research, we decided to make the analysis meaningful by classifying the datasets by domains; we come up the set of 5 to 10 words from each domain, Example- Electricity and Energy are the two different words by contextually it is related to the Power Sector domain and we wanted to ensure if this comparison using Open AI Embedding technique gives meaningful results by considering the contextual relevance, likewise we come up with more dataset around fifty to sixty different words from the same and different domain so that we could utilize such data for comparison and we also wanted to ensure it is editable by the developers to support changing the data for future analysis, Initially phrases comparison datasets are JSON formatted data which can be modified accordingly by the developers.

Document Comparison- We wanted our application to support comparison of any documents to show its contextual relevance between them, In order to do any comparison we need source documents and target documents, For Example, JobRequirment.txt is the source document and Job Profile A, Job Profile B is the target documents, so now when the source and target documents are compared, results will be meaningful and we could also come up the predefined threshold which can be managed by the application admin of any organization utilizing our application, Likewise we come up with few more meaningful document comparison by enabling the users of the app to add more documents either directly using file Manager or by enabling integration of user specific UI to our app to upload documents which requires development efforts, still we wanted to make sure application supports documents comparison dynamically making it more usable and researchable.

b.) 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, one folder is to keep all the documents as key document on the criteria in which documents has to be compared named as "SourceBasedOnDomains" and other folder named as " Source Based on Relevance" to keep all the documents which have to be sent for comparison (i.e.) in future if this application is integrated

with the UI or user facing application user can upload their resumes or motivation letter for joining the university which can be later used by our application to compare those with key category documents like “Job Requirement” or “Admission Requirement for Specific course in an University”

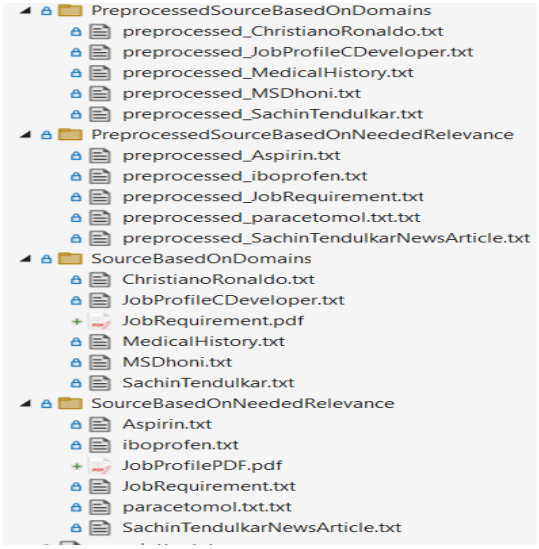

Fig 6: Folder Structure -Raw Documents and Preprocessed Documents

Interface IPreprocessor.cs is designed in such a way that it has all methods needed to do the basic functionalities which we have already mentioned in the introduction. Once such methods are mentioned below which are responsible to read the raw input dataset/documents/phrases and create a processed documents in the new folder by creating the folder name programmatically by appending the word “Preprocessed” and hence our program creates two new folders with “PreProcessedSourceBasedOnDomains” and “PreProcessedSourceBasedOnRelevance” respectively.

await ProcessTextFilesInFolderAsync (textPreprocessor, sourceDomainsFolder, outputDomainsFolder);

await ProcessTextFilesInFolderAsync (textPreprocessor, sourceRelevanceFolder, outputRelevanceFolder);

which have to be compared and produce the processed output into the new folders as shown in the above Fig 5.

Once the document is loaded into the program and processed documents are generated using the below method definition.

String PreprocessText (string text, TextDataType type);

Then the further methods which will be discussed below in our paper can be invoked to created embeddings and similarity score.

c.) Create Embedding’s for the input documents:

Interface CalculateEmbeddingAsync created for the purpose of accepting the phrases or documents as a text, and also additionally we want to ensure what is the category of the domain to find its relevance to the context it is created, hence it is designed to accept text 1, text 2 for processing source document or phrase in text1 and target document or target phrase in text2 and its corresponding filename respectively.

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

This Interface is made to utilize in the two different services called as SemanticSimilarityForDocumentsWithInputDataDynamic.cs and SemanticSimilarityPhrasesWithInputDataSet.cs where the actual implementation is created to produce the output embedding’s by utilizing the methods of open AI embedding nuget package.

OpenAIEmbeddingCollection collection = await client.GenerateEmbeddingsAsync (inputs);

GenerateEmbeddingsAsync () is the important method which accepts list of strings as inputs and produces collection of embedding as output for range of size 3052 as we have used larger embeddings (such as text-embedding-3-large, which has dimensions of 3072 as mentioned in the above literature review.

Public static void PrintScalarValues (float [] embedding)

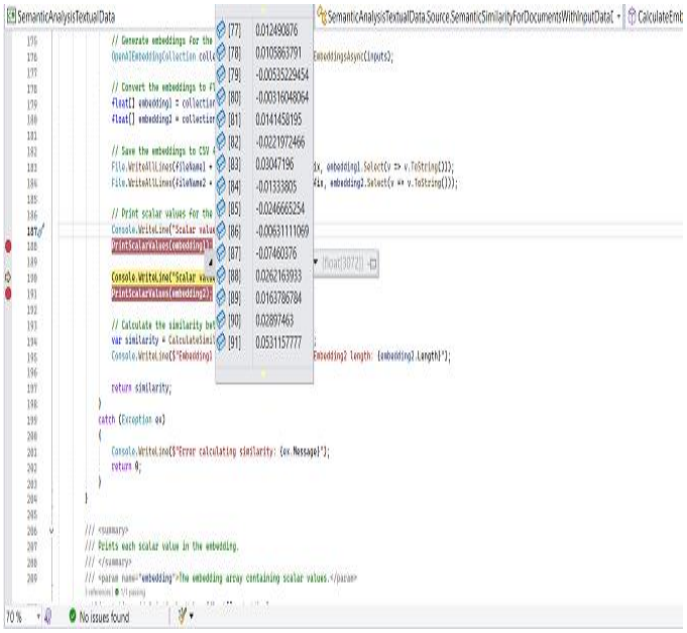

Fig 7: *Generated Embeddings in Debug Mode*

It is the custom method created to print the output of the each document or phrases as an embedding’s to print it individual values at every array till the end of the size of collection, Purpose of this method is utilize this values later during the visualization how closely or relatively the embedding’s at the vector space is being created. By knowing this difference it will help us to understand the correlation between the similarity score created VS Scalar values.

d.) Calculate Similarity using Cosine Similarity Algorithm based on programmatically generated embeddings of input documents

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 Calculate Similarity 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 (Σ 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 = 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.

e.) Generate the output of Similarity Score as a CSV File

Application is designed to support the generated output as a CSV file
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.

f.) Utilizing the output data's 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	preprocessed_JobProfileCDeveloper.txt	preprocessed_JobRequirement.txt	0.70199654	TRUE

Fig 8: Observation From with and Without Pre-Processing

From the Fig 7, we observed that Preprocessing with one sample of generated embedding score creates an impact where we could see some variation in similarity score as we are applying stop word removal and lamentizaiton and removing articles, Though the initial study was to understand the behavior of with and without preprocessing not to improve scores, Improvement of Scores may vary depends on the context. It really makes us understand that how good is that Open AI Embedding does make the contextual alignment, this result helps us to confirm by doing some specific improvising on preprocessing analysis on different use case, we will be able to achieve the better results and that we are focusing to do in our further improvements.

This Pre Processing is done for the understanding of contextual variations of small changes in the document though it is about the same topic or from same domain

In order to meet the ultimate end goal of understanding the Semantic Analysis of textual data between documents and create some meaningful output which would correlate to real time use cases**,** we used python as an external development tool to create a graphical chart, currently the application is designed to read the output CSV file generated dynamically if the files are placed in the root directory of the python app, so currently placement of output CSV required manual effort by the developers or the application admin, but we know this limitation which we are focusing during the later improvements either by us and paving way for other developers or ideas to improve the implementations. There are two types of plots we have designed; one chart is to graphically represent all the possible number of comparison of documents or phrases dataset designed by the user on X-axis VS its corresponding Similarity Score on Y axis; Other chart is designed in such way that developers are able to understand how the contextual relevance is actually generated by plotting its similarity score on Y axis VS Scalar Values (Ranges between 0-3052) on X-axis.

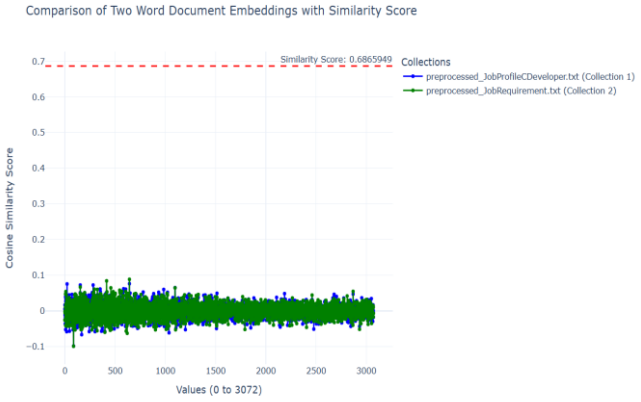

Fig 9: Scalar Values vs. Similarity Score Plot for One Comparison

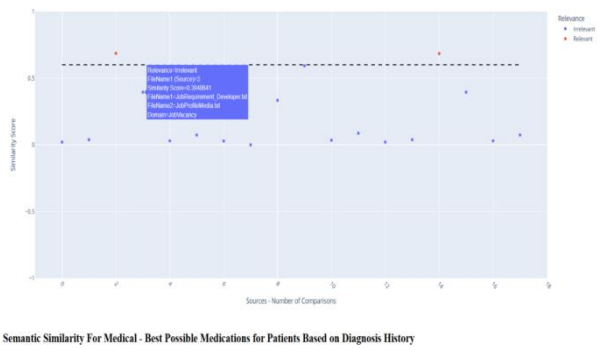

Fig 10: Semantic Similarity Score vs. Number of Comparison

We wanted to display the analysis over a single chart as it gives an easy represenation of values while hovering over the blue or red dots representing the user to click and observe the details attached to it along with its similarity score. We have implemented the Github Pages to make the users view the plot anywhere to make viewable over internet.Below are the links appended to represent both the form of charts.

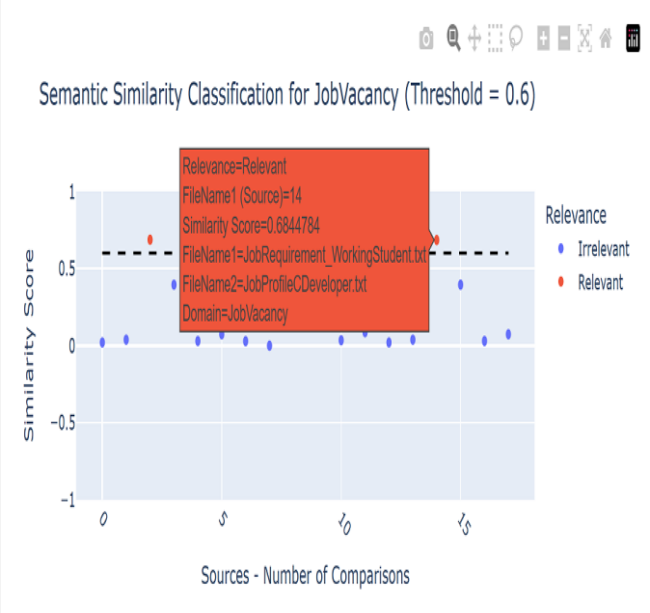

Fig 11: Total Number of Documents Compared VS Similarity Score

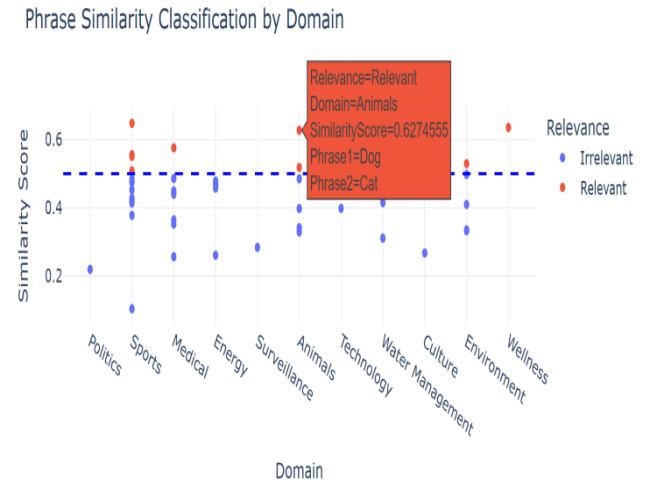

Fig 12: Phrases Comparison (X-axis) vs. Similarity Score (Y-axis)

g.) Utilizing CSV to Plot Using Python:

As there are lot of options to explore, python is a open source and it has good predefined libraries for creating scatter plots, after some initial study inorder to display the similarity score for varoius documents from several categories, we have implemetend a Python-based Flask application was developed to classify and visualize similarity scores across different domains. The tool reads CSV files containing similarity scores, maps them to respective domains, and applies predefined thresholds for classification.

Plots are generated using Plotly, and thresholds for domains such as "JobVacancy", "Medical", and "Sports" are visualized with interactive scatter plots. This tool enhances interpretability by categorizing similarity scores as "Relevant" or "Irrelevant" based on domain-specific thresholds.

Currently domain name mapping is done based on the initial application desing requirement, by modiying the needs of the user or future analyse, developers or we will continue refactoring this accordingly which enables support for more domain/context related mappings of data which will induce a better visualisation.

IV. TestCase with Results

We have implemented several test cases ensuring that when running the test cases through test explorer, all the code and methods are covered including positive and negative scenario's i.e. handling exceptions and null values, currently we did not use the mock data to ensure test cases as this application is the initial setup for the purpose of analysis we are utilizing the application "data" folder which we have defined for the purpose of maintaining datasets are copied into our test project and by reading those files dynamically using program, we are successfully in maintaining the code coverage and the test cases. Test cases are separated by creating individual test classes for each service we have defined based on the functionalities, as we have segregated our business functionalities into 4 classes; we created test classes also based on that and The test framework is implemented using Microsoft.VisualStudio.TestTools.UnitTesting, and the test cases are run through the Visual Studio Test Explorer to verify code coverage and robustness.

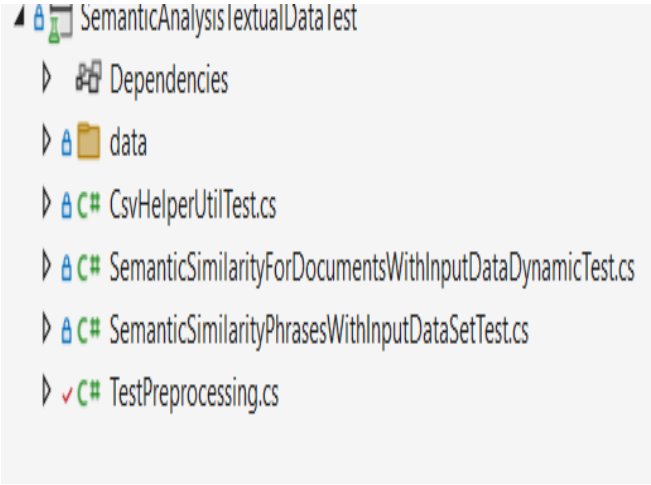

Fig 13: Representation of Semantic Analysis Test Project

Test for Preprocessor Method: This test ensures all methods in the preprocessing module function correctly. It is crucial for confirming accurate embedding generation.

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.

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.

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.

Service Provider Configuration Validation: Validates the configuration of services returned by the ConfigureServices method, ensuring it matches the expected type.

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

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

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

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

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.

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.

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.

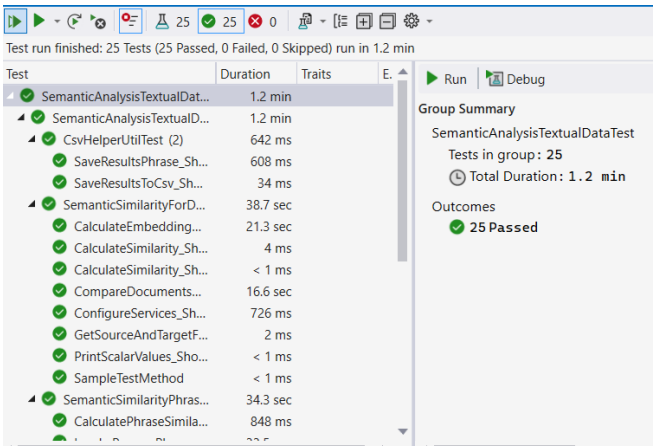

Fig 14: Respresenation of Test Run Results of All Test Cases

V. Overcoming Limitations:

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.Inorder 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.

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.Inorder 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.

Performance: The performance of text preprocessing, especially for large datasets, might be a concern. The current implementationmay 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.

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. Inorder 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 :

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.

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.

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.

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.

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. a. H. T. Aboelghit, "Textual Similarity Measurement Approaches: A Survey (1)," *The Egyptian Journal of Language Engineering*, vol. 7, no. 2, pp. 41-62, 2020.

[2] D. P. P. A. G. D. P. Goutam Majumder, "Semantic Textual Similarity Methods, Tools, and Applications: A Survey," Computacion y Sistemas, 2016.

[3] P. Wiemer-Hastings, "Latent Semantic Analysis," Citeseer, 2004.

[4] E. G. a. S. Markovitch, "Computing Semantic Relatedness usin Wikipedia-based Explicit Semantic Analysis," in *International Conference on Intelligent Text Processing and Computational Linguistics*, 2013.

[5] K. C. G. C. J. D. Tomas Mikolov, "Efficient Estimation of Word Representations in Vector Space," in International Conference on Learning Representations, 2013.

[6] R. S. C. M. Jeffrey Pennington, "GloVe: Global Vectors for Word Representation," in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 2014.

[7] I. G. Nils Reimers, Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks, Hong Kong, China: Association for Computational Linguistics, 2019.

[8] M.-W. C. K. L. K. T. Jacob Devlin, "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, Association for Computational Linguistics, 2019, pp. 4171--4186.

[9] https://openai.com/index/new-embedding-models-and-api-updates/

[10] https://learn.microsoft.com/en-us/azure/cosmos-db/gen-ai/vector-embeddings

[11] https://openai.com/index/introducing-text-and-code-embeddings/

[12] N. K. K. ,. K. S. Dipendra Prasad Yadav, "Distance Metrics for Machine Learning and it's Relation with Other Distances.," Mikailalsys Journal of Mathematics and Statistics, vol. 1, pp. 15-23, 2023.

[13] O. N. B. W. G. B. John Giorgi, "DeCLUTR: Deep Contrastive Learning for Unsupervised Textual Representations," 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), vol. 1, p. 879–895, 2021.