Semantic Similarity Analysis of Textual Data

M A B Siddique Naeem
m.naeem@stud.fra-uas.de

Md Abdul Ahad
abdul.ahad@stud.fra-uas.de

Haimanti Biswas
haimanti.biswas@stud.fra-uas.de

Muhammad Faraz Abbasi
muhammad.abbasi@stud.fra-uas.de

Abstract* - Semantic similarity analysis is a fundamental task in Natural Language Processing (NLP) that quantifies the degree of similarity between textual data, and semantic equivalence of multi-word sentences for rules and procedures contained in the documents on railway safety and ranging from search engine to machine translation and document clustering. This project leverages OpenAI's embedding models – text-embedding-ada-002, text-embedding-3-small, and text-embedding-3-large – to compute cosine similarity scores across different levels of textual data including words, phrases, and documents. The study explores semantic relationships at both word/phrase and document levels, employing well-known entities (e.g., "Angela Merkel" vs. "Government") and contextual document pairs to highlight variations across domains. Implemented in .NET, the project generates similarity metrics, the results are exported to CSV files for further analysis and visualized using a Python-based scatter plots. The study investigates the variation in similarity across different contexts and highlights the importance of embedding-based approaches in capturing semantic relationships. Through well-documented code, rigorous methodology, and intuitive visualization, this work provides a reproducible framework for assessing semantic similarity in diverse text-based applications.

***Keywords* —Natural Language Processing, OpenAI's embedding models, Cosine Similarity, Semantic Textual Similarity, Information Content.**

I. Introduction

Semantic similarity measures the degree of equivalency between two textual entities, which can be words, phrases, sentences, or entire documents. Semantic similarity analysis plays a pivotal role in NLP applications such as information retrieval, recommendation systems, plagiarism detection, question-answering system, and text classification. Traditional approaches to similarity measurement relied on lexical matching techniques, such as Jaccard similarity or term frequency-inverse document frequency (TF-IDF). However, these methods fail to capture the contextual and semantic meaning of words effectively.

The accuracy of semantic similarity models is critical to ensuring reliable and meaningful results, making it an essential factor in real-world NLP applications. There are several computational techniques available to measure the similarity between textual data, each offering different levels of precision and efficiency. This project explores and implements advanced methodologies for semantic similarity analysis to enhance the understanding of text-based relationships.

Text similarity can be evaluated using lexical or semantic approaches. Lexical similarity is based on surface-level string matching, where a sentence is treated as a sequence of characters or words. In contrast, semantic similarity focuses on the meaning behind the words, rather than their literal form. String-based or lexical similarity measures rely on comparing character sequences, which may fail to capture the true intent and meaning of a sentence. Various computational models have been developed to improve similarity metrics, including embedding-based approaches that convert text into high-dimensional numerical representations. This project leverages state-of-the-art embedding models to improve the accuracy and efficiency of semantic similarity analysis, enabling machines to interpret human language more effectively.

Recent advancements in deep learning have led to the adoption of word embeddings, where words and phrases are transformed into high-dimensional vector representations. Models like Word2Vec, GloVe, and transformer-based architectures such as BERT and OpenAI's embedding models have significantly improved the accuracy of semantic similarity computations. By using OpenAI's embedding API to generate these vector representations. The system compares embeddings using cosine similarity and supports multiple input modes—including single input comparisons and batch file comparisons across different file formats.

The study addresses two primary objectives:

1. **Word/Phrase-Level Analysis:** Investigating how semantically related terms (e.g., "Angela Merkel" and "Government") compare to unrelated pairs (e.g., "Cristiano Ronaldo" and "Government"). This highlights the role of domain-specific context in similarity metrics.

2. **Document-Level Analysis:** Evaluating similarity between documents on aligned topics (e.g., two articles about machine learning) versus disparate topics to assess the impact of contextual alignment.

The project provides efficient implementation using .NET framework and OpenAI NuGet package, enabling users to input textual data, computes similarity scores using multiple models, exports results to CSV files, and visualizes findings using Python-based scatter plot. The methodology emphasizes reproducibility, leveraging the OpenAI NuGet package and Python scripts for post-processing. By combining theoretical insights with practical implementation, this work contributes to the broader understanding of semantic similarity in NLP.

II. Literature Review

Semantic similarity analysis has been a key focus in NLP research for decades.

Early Approaches to Semantic Similarity

One of the earliest methods for measuring semantic similarity was lexical and syntactic matching, where similarity was determined based on exact word overlap or dictionary-based relationships. WordNet, a lexical database, played a significant role in this era by grouping words into synsets and using path-based similarity measures (Miller, 1995). However, these approaches struggled with issues of polysemy (words with multiple meanings) and synonymy (different words with similar meanings), making them insufficient for complex NLP tasks.

Statistical and Vector Space Models

With the advancement of computational linguistics, vector space models (VSMs) gained prominence. Term Frequency-Inverse Document Frequency (TF-IDF) was widely used to represent textual data in a high-dimensional space, computing similarity based on term co-occurrence (Salton & McGill, 1983). However, TF-IDF was unable to capture semantic relationships beyond surface-level term matching.

Latent Semantic Analysis (LSA) improved upon VSMs by applying Singular Value Decomposition (SVD) to reduce dimensionality and uncover hidden semantic structures in text corpora (Deerwester et al., 1990). LSA demonstrated effectiveness in capturing meaning beyond word matching but still faced limitations in modeling word order and context.

Neural Network-Based Embeddings

The introduction of word embeddings marked a major shift in NLP. Word2Vec, introduced by Mikolov et al. (2013), trained neural networks on large corpora to generate dense vector representations of words based on their co-occurrence. Word2Vec's Skip-gram and Continuous Bag-of-Words (CBOW) models captured word meaning more effectively than previous methods.

GloVe (Global Vectors for Word Representation) further enhanced word embeddings by incorporating both global corpus statistics and local context (Pennington, Socher, & Manning, 2014). These models significantly improved semantic similarity computations, enabling better performance in NLP tasks such as document classification and machine translation.

Transformer-Based Models and Contextual Embeddings

The advent of transformer-based architectures, particularly BERT (Bidirectional Encoder Representations from Transformers), revolutionized NLP. Unlike previous models, BERT produced dynamic embeddings, meaning the representation of a word changed based on its surrounding context (Devlin et al., 2019). This approach significantly improved semantic similarity analysis by considering the entire sentence structure.

More recent advancements include OpenAI's GPT-based embeddings, which leverage self-supervised learning on vast textual data to generate high-quality embeddings for semantic similarity tasks (Brown et al., 2020). The OpenAI embedding models used in this project—text-embedding-ada-002, text-embedding-3-small, and text-embedding-3-large—build on these transformer-based approaches to provide accurate and scalable semantic similarity measurements.

Current Study Contribution

This project builds upon the existing literature by implementing OpenAI’s embedding models within a structured framework in .NET. By computing cosine similarity between embeddings at word, phrase, and document levels, the study provides insights into the effectiveness of modern NLP embeddings. The project also incorporates visualization techniques to analyze trends in similarity scores, making it a valuable contribution to the field of semantic similarity analysis, aligning with recent trends in explainable AI (Samek et al., 2021).

Applications and Challenges:

- **Applications:** Semantic similarity is critical for chatbots (e.g., matching user queries to responses), plagiarism detection, and clustering (e.g., grouping news articles by topic).
- **Challenges:** Variability in similarity scores across models (e.g., ada-002 vs. text-embedding-3-large) and the need for domain adaptation remain active research areas.

III. Methodology

The Semantic Similarity Analysis of Textual Data project follows a structured approach to compute similarity scores between text inputs using OpenAI’s embedding models. This methodology section describes the key components, including data preprocessing, embedding generation, similarity computation, result storage, and visualization. The system is designed to handle both word/phrase-level and document-level comparisons, ensuring flexibility for diverse text analysis needs.

1. System Architecture

The system consists of the following key components:

i) User Input Handling: Accepts textual data from users via direct input or document upload.

ii) Text Preprocessing*:* Extracts and cleans text for analysis.

iii) Embedding Generation: Uses OpenAI’s API to generate vector representations of text.

iv) Similarity Computation: Applies cosine similarity to calculate similarity scores.

v) Result Storage and Output: Saves similarity scores in a CSV file for further analysis.

vi) Visualization: Uses Python for graphical representation of results.

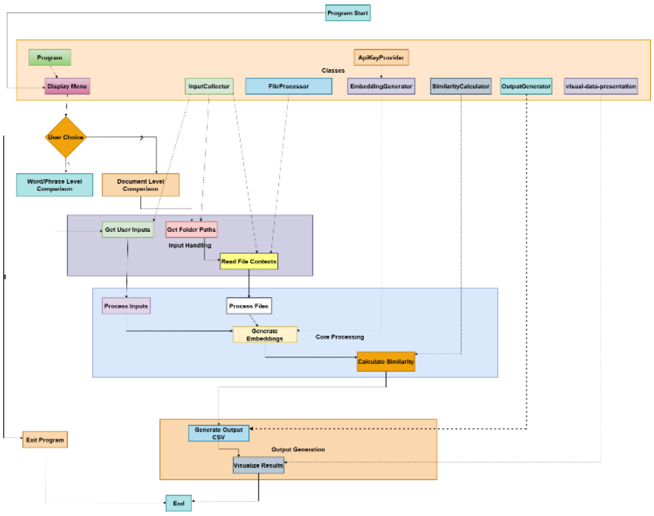


Figure 1: System Architecture flow chart.

2. *Data Collection and Preprocessing*

The system supports two modes of input for similarity analysis:

Word/Phrase Level Comparison: Users manually input multiple words or phrases, which are stored in a list for processing.

Document Level Comparison: Users provide directories containing .txt, or .pdf files from which the system extracts textual content for further processing.

To efficiently process these inputs, the system employs two key methods within the InputCollector class: GetFileContents and GetUserInputs.

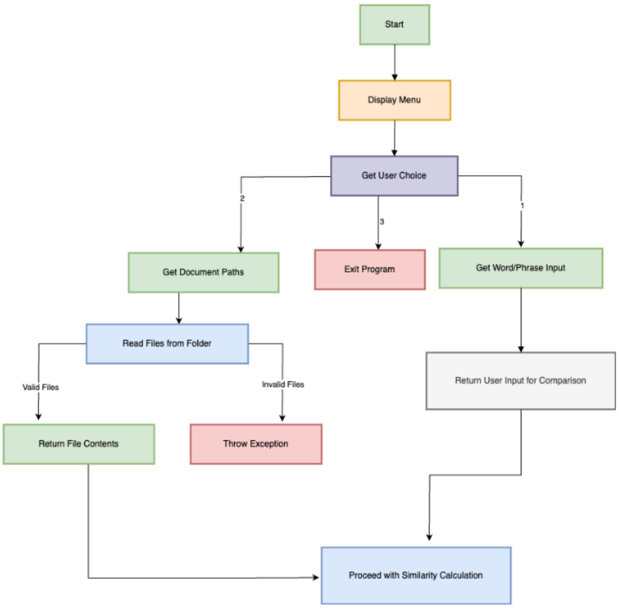


Figure 2: Data collection (InputCollector class) flow chart.

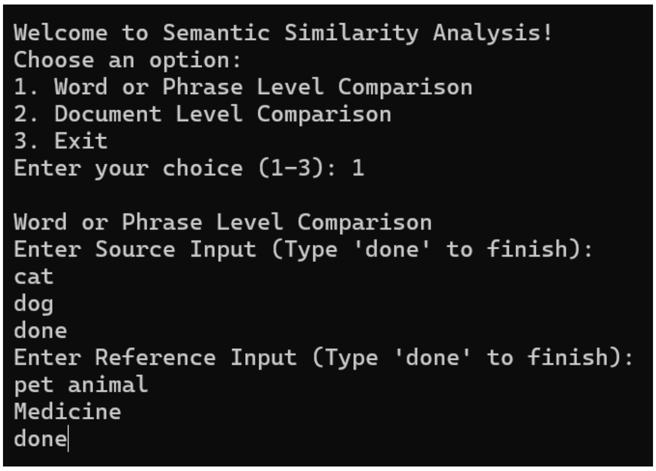
2.1 Manual Text Input Handling (GetUserInputs)

The “GetUserInputs” method in facilitates the collection of user-provided textual data for two distinct categories: "Source Input" and "Reference Input." It initializes two lists, “sourceContents” and “refContents”, to store these inputs separately. The method prompts the user to enter multiple lines of text for the source content, signaling to type 'done' when input is complete. It enters a loop where it reads user input using “Console.ReadLine()”, appends each line to the “sourceContents” list, and exits the loop upon detecting the sentinel value 'done' (case-insensitive). This process is then mirrored for the reference content, collecting inputs into the “refContents” list. Upon completion, the method returns a tuple containing both lists. This design allows for flexible and user-friendly data entry, enabling the input of multiple lines for each category until the user indicates completion.

`GetUserInputs`

```
Console.WriteLine("Enter Source Input
(Type 'done' to finish):");
while (true) {
    string input = Console.ReadLine();
    if (input.ToLower() == "done")
break;
    sourceContents.Add(input);
}
Console.WriteLine("Enter Reference Input
(Type 'done' to finish):");
while (true) {
    string input = Console.ReadLine();
    if (input.ToLower() == "done")
break;
    refContents.Add(input);
}
```

The purpose of the “GetUserInputs” method is to gather two sets of textual inputs—**Source Input** and **Reference Input**—from the user for similarity analysis. It provides an interactive way for users to manually enter multiple lines of text, terminating each input session with the keyword 'done'. The method then returns these inputs as separate lists within a tuple, ensuring structured and organized data collection for further processing. This approach enhances usability while maintaining clarity in handling user-provided content.



```
Welcome to Semantic Similarity Analysis!
Choose an option:
1. Word or Phrase Level Comparison
2. Document Level Comparison
3. Exit
Enter your choice (1-3): 1

Word or Phrase Level Comparison
Enter Source Input (Type 'done' to finish):
cat
dog
done
Enter Reference Input (Type 'done' to finish):
pet animal
Medicine
done
```

Figure 3: Word/Phrase level input collection from user

2.2 File-Based Input Handling (GetFileContents)

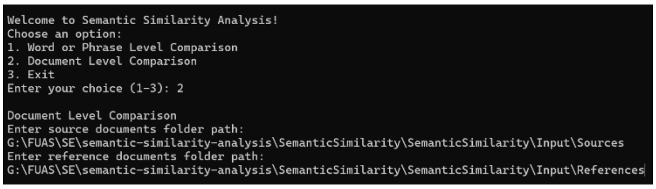
The “GetFileContents” method in the “InputCollector” is designed to retrieve and read the contents of all “.txt” and “.pdf” files located within a specified directory. Initially, it verifies the existence of the provided “folderPath”; if the directory does not exist, an “ArgumentException” is thrown, indicating the path is invalid. The method then utilizes “Directory.GetFiles” to enumerate files in the directory, filtering for those with “.txt” or “.pdf” extensions by applying a LINQ query with “EndsWith” checks. If no matching files are found, another “ArgumentException” is raised to inform that the directory lacks the specified file types. For each identified file, the method instantiates a “FileProcessor” object to read the file's content via its “ReadFileText” method, appending the retrieved text to a “List<string>“ named “fileContents”. Should an exception occur during the reading process, it is caught, and an “IOException” is thrown, detailing the specific file that caused the error. Upon successful processing, the method returns the “fileContents” list, containing the text of all relevant files. This implementation ensures robust error handling and efficient aggregation of file contents from the specified directory.

`GetFileContents`

```
var files =
Directory.GetFiles(folderPath, "*.*",
SearchOption.TopDirectoryOnly).Where(fil
e => file.EndsWith(".txt",
StringComparison.OrdinalIgnoreCase) ||
file.EndsWith(".pdf",
StringComparison.OrdinalIgnoreCase)).ToL
ist();

...

foreach (var file in files) {
...
  var multiFileProcessor = new
FileProcessor();
  string content =
multiFileProcessor.ReadFileText(file);
  fileContents.Add(content);
...
}
...
```



```
Welcome to Semantic Similarity Analysis!
Choose an option:
1. Word or Phrase Level Comparison
2. Document Level Comparison
3. Exit
Enter your choice (1-3): 2

Document Level Comparison
Enter source documents folder path:
G:\FUAS\SE\semantic-similarity-analysis\SemanticSimilarity\SemanticSimilarity\Input\Sources
Enter reference documents folder path:
G:\FUAS\SE\semantic-similarity-analysis\SemanticSimilarity\SemanticSimilarity\Input\References
```

Figure 4: Document level input collection from user

2.3 File Parsing:

The “FileProcessor” class is designed to handle the extraction of text content from various document formats, specifically “.txt”, “.pdf”, and “.docx” files. Its primary method, “ReadFileText(string filePath)”, determines the file type by examining the file extension and delegates the text extraction process to the appropriate method based on this extension. For “.txt” files, it utilizes “File.ReadAllText(filePath)” to read the content directly. When dealing with “.pdf” files, it calls the “ExtractTextFromPdf(filePath)” method, which employs the iText 7 library's “PdfReader” and “PdfDocument” classes to parse the PDF and extract text from each page using “PdfTextExtractor.GetTextFromPage”. This approach ensures that text is retrieved accurately from PDFs, considering their complex structure. For “.docx” files, although the specific implementation isn't provided in the given code, text extraction typically involves using libraries such as Open XML SDK or third-party tools like GroupDocs.Parser, which facilitate the reading of Word documents and extraction of text content. If an unsupported file format is encountered, the method throws a “NotSupportedException”, indicating the file type is not handled by the processor. This design allows the “FileProcessor” class to flexibly and efficiently extract text from multiple document types, enabling further processing or analysis as required by the application. *Note: to getting .docx file similarity need license key.*

Mathematical Representation for PDF Extraction
Let:
PP = Number of pages in the PDF
TiT_i = Extracted text from page ii

Total extracted text: $T = \sum_{i=1}^{P} T_i$

ExtractTextFromPdf

```
using (PdfReader reader = new
PdfReader(filePath))
using (PdfDocument pdfDoc = new
PdfDocument(reader))
{
    string text = "";
    for (int i = 1; i <=
pdfDoc.GetNumberOfPages(); i++) {
        text +=
PdfTextExtractor.GetTextFromPage(pdfDoc.
GetPage(i)) + "\n";
    }
    return text;
}
```

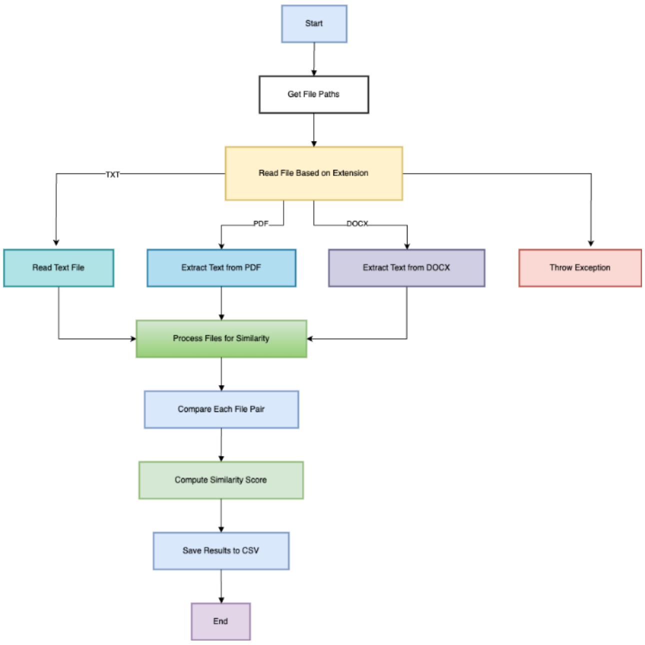


Figure 5: Graphical Representation of Implemented Draw InputCollector

3. *Embedding Generation*

The ***EmbeddingGenerator*** class is responsible for retrieving embeddings from OpenAI’s API. The following steps are followed:
API Key Handling: The API key is securely fetched from environment variables using the ApiKeyProvider class.

Embedding Request: Text inputs are sent to OpenAI’s API, specifying the model to be used.

Vector Conversion: The API returns numerical vectors, which are stored for further computation.

3.1 Embedding Models Used

The project supports three OpenAI embedding models:

text-embedding-ada-002: Lightweight and efficient.
text-embedding-3-small: Optimized for small-scale comparisons.
text-embedding-3-large: High-precision model for detailed analysis.

3.2 Implementation Details:

Word embedding’s are numerical representations of words or text, which help in performing semantic similarity calculations.

GenerateEmbeddingsAsync method is designed to asynchronously generate embedding’s for a given text input using OpenAI's embedding models. Embedding’s are dense vector representations that capture the semantic meaning of text, facilitating tasks like search Clustering, and classification. In this method, the input content is first validated to ensure it is neither null nor empty, throwing an ArgumentException if it is. An instance of EmbeddingClient is then created using the specified model and an API key. The method proceeds by calling GenerateEmbeddingAsync on the openAIClient instance, passing the content to obtain an OpenAIEmbedding object. This embedding is subsequently converted to a float array using the ToFloats().ToArray() method, which is then returned. If any exceptions occur during this process, they are caught, logged to the console, and rethrown as an InvalidOperationException with a descriptive message. This approach aligns with best practices for integrating OpenAI's embeddings into .NET applications, enabling developers to leverage advanced text analysis capabilities.

EmbeddingGenerator

```
{
...
    ApiKeyProvider apiKeyProvider = new
ApiKeyProvider();
    string apiKey =
apiKeyProvider.GetApiKey();
    _apiKey = apiKey ?? throw new
ArgumentNullException(nameof(apiKey));
...
EmbeddingClient openAIClient = new
EmbeddingClient(model, _apiKey);
OpenAIEmbedding embedding = await
openAIClient.GenerateEmbeddingAsync(cont
ent);
return embedding.ToFloats().ToArray();
...
}
```

3.3 Mathematical Explanation:

Embedding is a vector representation of text. Suppose allows words or sentences to be compared mathematically based on their meaning. Let we have two sentences:
"I love programming" & "Coding is fun"

After calling the OpenAI API, we get embedding vectors:

V1=[0.12,−0.45,0.87,0.23,0.55,...]V_1 = [0.12, -0.45, 0.87, 0.23, 0.55, ...]

V2=[0.11,−0.50,0.82,0.20,0.60,...]V_2 = [0.11, -0.50, 0.82, 0.20, 0.60, ...]

Each vector represents semantic meaning in high-dimensional space.

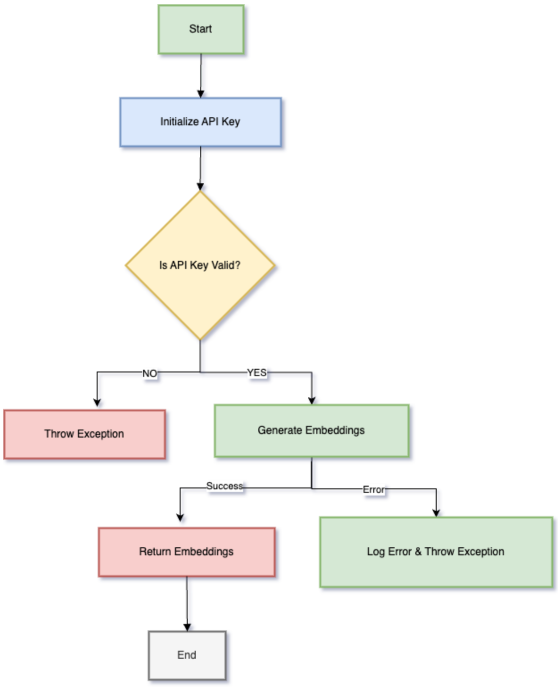


Figure 6: Graphical Representation of Implemented Draw Embedding Generator

The EmbeddingGenerator orchestrates the transformation of textual input into numerical vector representations, known as embedding's, by interfacing with AI models. Upon receiving a text input, the class initiates a validation process to ensure the content is neither null nor empty, safeguarding against erroneous operations. Following validation, it establishes a connection with a specified AI service, such as OpenAI's embedding model, utilizing the provided API key for authentication. The class then transmits the validated text to the AI service, requesting the generation of an embedding. Upon successful retrieval, the resulting embedding—typically in the form of a high-dimensional float array—is processed and returned to the caller. Throughout this sequence, the class incorporates robust error-handling mechanisms; should any exceptions arise during the embedding generation or retrieval phases, they are captured and logged, and a custom exception is thrown to inform the caller of the specific failure. This structured flow ensures that the "EmbeddingGenerator" provides a reliable and efficient means of generating text embedding's, facilitating seamless integration of advanced natural language processing capabilities into .NET applications.

4. *Similarity Computation*

The system calculates similarity scores using cosine similarity, a widely used metric for measuring the angle between two vectors in high-dimensional space. The steps include:

Fetching Embeddings: Retrieved vectors for both source and reference texts.

Computing Dot Product: Calculates similarity by comparing vector directions.

Normalizing Scores: Ensures values range between -1 (completely different) and 1 (identical).

Generating Multi-Model Scores: Each text pair is analyzed using all three embedding models.

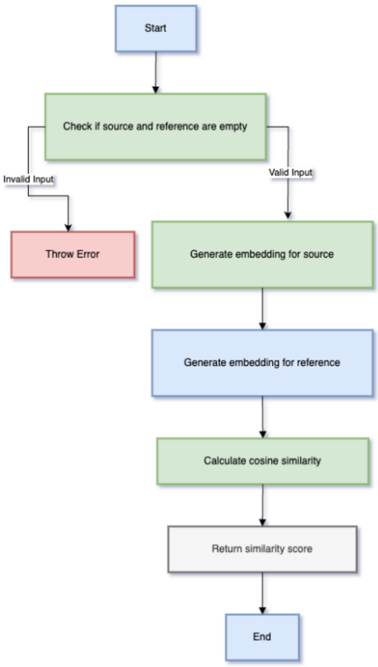

Start
Check if source and reference are empty
Invalid Input
Valid Input
Throw Error
Generate embedding for source
Generate embedding for reference
Calculate cosine similarity
Return similarity score
End

Figure 7: Graphical Representation of Implemented Draw SimilarityCalculator

The SimilarityCalculator class is responsible for computing similarity scores between two pieces of text using cosine similarity. It does this by generating numerical embeddings for each text and then comparing these embeddings mathematically. This function calculates cosine similarity between two embeddings (numerical representations of text). Cosine similarity measures the angle between two vectors in an n-dimensional space, giving a similarity score between -1 (completely opposite) and 1 (identical).

The SimilarityCalculator class is central to computing similarity scores between pairs of source and reference texts using
various machine learning models. Its primary function, CalculateSimilarityAsync, operates asynchronously to handle potentially time-consuming operations without blocking the main execution thread.

Formula for cosine similarity:

$$\text{Similarity} = \frac{\sum_{i=1}^{n} A_i B_i}{\sum_{i=1}^{n} A_i^2 \sum_{i=1}^{n} B_i^2}$$

Range:

1 → Identical
0 → Unrelated
-1 → Opposite meaning (rare in embeddings)

`CalculateCosineSimilarity`

```
...
float dotProduct = 0, magnitude1 = 0,
magnitude2 = 0;
for (int i = 0; i < embedding1.Length;
i++) {
    dotProduct += embedding1[i] *
embedding2[i];
    magnitude1 += embedding1[i] *
embedding1[i];
    magnitude2 += embedding2[i] *
embedding2[i];
}
magnitude1 =
(float)Math.Sqrt(magnitude1);
magnitude2 =
(float)Math.Sqrt(magnitude2);
...
return dotProduct / (magnitude1 *
magnitude2);
```

5. Output Generation and Visualization

The “OutputGenerator” class is responsible for managing the output of the similarity comparison results and saving them to a file. The first part of the function determines the output file path by navigating the directory structure of the project and setting the path for saving the results as a CSV file in the “Output” folder.
Next, the function initializes several variables to handle the processing of similarity scores. The “results” list is prepared to store the similarity results, and the “totalPairs” variable is calculated as the product of the counts of source and reference content. A counter “processedPairs” is initialized to track the number of processed pairs during the comparison.

Then, the function moves on to calculate the similarity scores using the “SimilarityCalculator” class. For each pair of source and reference content, it creates a “SimilarityResult” object, which holds the source, reference, and their similarity scores. The scores are calculated using the “CalculateSimilarityAsync” method from the “SimilarityCalculator” class, applying different models like "text-embedding-ada-002," "text-embedding-3-small," and "text-embedding-3-large." The result for each pair is added to the “results” list. This process ultimately produces a collection of similarity results, which can be saved into a CSV file for further analysis.

`GenerateOutputAsync & WriteResultsToCsv:`

```
foreach (var source in sourceContents) {
  foreach (var refr in refContents) {
    ...
    var (scoreLarge, srcEmbeddingLarge,
refEmbeddingLarge) = await
similarityCalculator.CalculateSimilarity
Async("text-embedding-3-large", source,
refr);
    ...
    var result = new SimilarityResult
{…};
    results.Add(result);
  }
}
```

```
using var writer = new
StreamWriter(filePath);
using var csv = new CsvWriter(writer,
System.Globalization.CultureInfo.Invaria
ntCulture);
csv.WriteRecords(results);
```

A flow char for understanding the process of output generation is shown in Figure 8.

A python script (similarity-score-visualization.py) is then used to visualize similarity scores of a reference word/phrase or documents against some source word/phrase or documents using interactive plots generated with Plotly. The script begins by importing necessary libraries such as Pandas for data manipulation and Plotly for visualization. The `read_csv` function read the CSV file and handle errors. The `generate_bar_chart` generate bar chart for a reference. The function groups the scores and represents them as bars, with hover text providing additional context such as the source and reference texts, embedding model name, and similarity score.

Another python script (scalar-value-visualization.py) is used to visualize scalar value of each source and reference embedding. It first takes file paths of source and reference scalar value CSV file and read those files. Later `plot_scatter_graph` function is called to create scatter plots for a source-reference pair.

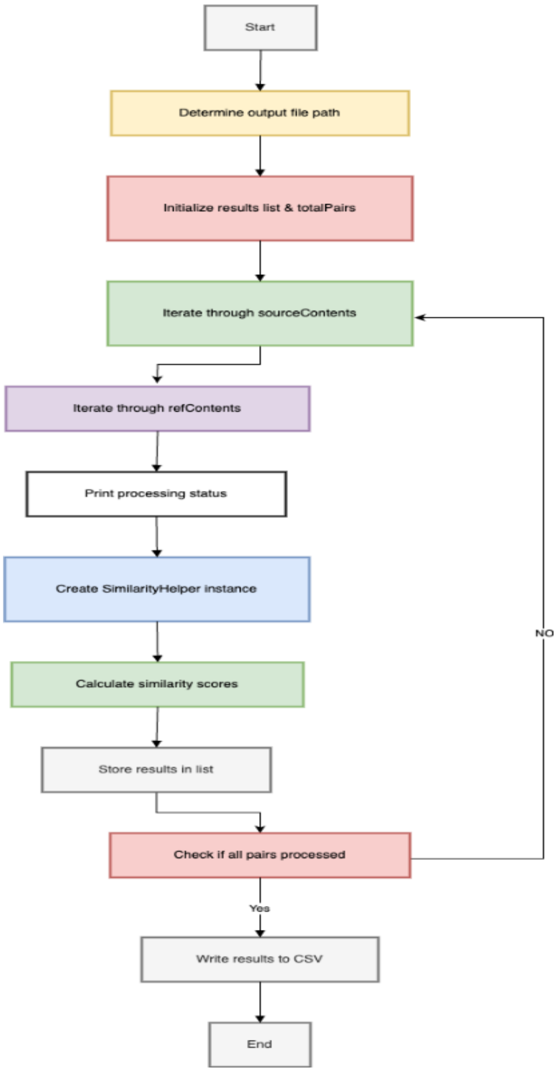

Start
Determine output file path
Initialize results list & totalPairs
Iterate through sourceContents
Iterate through refContents
Print processing status
Create SimilarityHelper instance
Calculate similarity scores
Store results in list
Check if all pairs processed
NO
Yes
Write results to CSV
End

Figure 8: Graphical Representation of Implemented Draw OutputGenerator

IV. Unit-Testing

Unit testing plays a crucial role in ensuring the correctness, reliability, and maintainability of the project. Since this project heavily relies on OpenAI API for embedding generation and performs mathematical operations like cosine similarity calculations, it is essential to validate individual components to detect potential issues early.

The unit tests are implemented using ***MSTest*** and ***xUnit***, two widely used testing frameworks for .NET applications. The tests cover key functionalities such as API key retrieval, embedding generation, input handling, output processing, and similarity calculations.

Test Classes and Key Scenarios

A. ApiKeyProviderTests

Objective: Ensure secure API key retrieval and environment variable fallback.

Key Tests:

`Constructor_WithValidApiKey_SetsApiKey`: Validates direct key assignment.

`Constructor_WithoutApiKey`: Verifies .env file loading and environment variable handling.

Outcome: Confirmed robust key management with graceful fallback to environment variables.

B. EmbeddingGeneratorTests:

Objective: Validate embedding generation and error handling for invalid inputs.

Key Tests:

`GenerateEmbeddingsAsync_ValidContent`: Ensures embeddings are generated for valid text.

`GenerateEmbeddingsAsync_NullContent`: Checks *ArgumentException* for null input.

`GenerateEmbeddingsAsync_ApiError`: Verifies *InvalidOperationException* for invalid API models.

Outcome: Reliable input validation and error propagation for API failures.

C. InputCollectorTests:

Objective: Test user input processing, menu display, and file content retrieval.

Key Tests:

`GetUserChoice_ValidInput`: Validates correct parsing of menu choices.

`GetFileContents_ValidFiles`: Ensures file content extraction from directories.

`GetUserInputs_EmptySourceAndReferenceInputs`: Confirms handling of empty inputs.

Outcome: Robust input sanitization and edge-case handling (e.g., invalid paths, mixed-case "done").

D. *OutputGeneratorTests:*

Objective: Verify CSV file generation and content formatting.

Key Tests:

`GenerateOutputAsync_ShouldGenerateCsvFile`: Checks file creation with valid data.

`GenerateOutputAsync_ShouldHandleSpecialCharacters`: Validates encoding of special characters.

`GenerateOutputAsync_ShouldNotThrowException`: Ensures no unhandled exceptions.

Outcome: Reliable output generation under varied conditions (empty lists, long strings).

E. *SimilarityCalculatorTests:*

Objective: Validate cosine similarity calculations and error cases.

Key Tests:

`CalculateCosineSimilarity_ValidEmbeddings`: Confirms similarity score bounds ([-1, 1]).

`CalculateCosineSimilarity_EmbeddingsDifferentLengths`: Ensures *ArgumentException* for mismatched vectors.

`CalculateSimilarityAsync_ErrorOccurs`: Tests graceful failure (returns -1 for invalid models).

Outcome: Accurate similarity computation and error resilience.

Testing Approach

Each test class follows a structured testing methodology:

- **Valid Input Testing:** Ensures correct functionality under normal operating conditions.
- **Error Handling & Exception Testing:** Verifies that the system correctly handles invalid inputs and throws appropriate exceptions.
- **Edge Case Handling:** Tests extreme scenarios, such as empty inputs, invalid API keys, and zero-vector embeddings.
- **Mocking External Dependencies:** Uses environment variables and controlled inputs to isolate and test individual components.

Mocking and Dependency Handling

OpenAI API calls are not directly mocked but are tested using controlled input cases (e.g., invalid models).

File I/O operations are tested with temporary directories to avoid dependency on actual file structures.

Test Execution & Code Coverage

Tests are executed using the MSTest and xUnit test runners.

Code coverage is measured to ensure critical functionalities are tested adequately.

V. Result Analysis

The Semantic Similarity Analysis of Textual Data project provides insights into the semantic relationships between words, phrases, and documents using OpenAI's embedding models. The results are analyzed at both word/phrase level and document level, utilizing cosine similarity as the primary metric. The findings from the similarity calculations are stored in a structured CSV format and visualized using a scatter plot. This section presents the key findings from our experiments, organized by the type of analysis conducted.

A. *Word and Phrase Level Similarity*

Our experiments with word and phrase comparisons revealed clear patterns in how different models capture semantic relationships:

1. Domain-Specific Similarity:

The comparison between "Football" and "Sports" yielded consistently high similarity scores across all three models (Ada: 0.94, Small: 0.62, Large: 0.76), reflecting the sports context of Football.

In contrast, "Tesla" showed significantly lower similarity to "Sports" (Ada: 0.80, Small: 0.19, Large: 0.21), demonstrating the models' ability to distinguish between car brand and sports domains.

2. Model Performance Comparison:

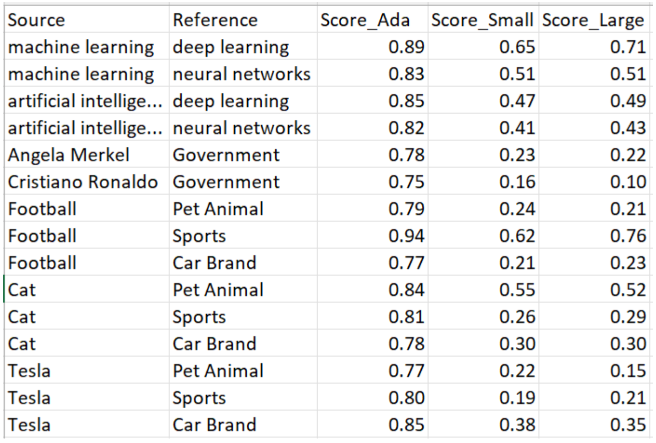
The text-embedding-3-large model consistently provided the most nuanced similarity scores, particularly in borderline cases where semantic relationships were complex.

While all models agreed on the relative ordering of similarity pairs, the absolute scores varied, with text-embedding-ada-002 often producing slightly higher values than the newer models.

3. Technical Terminology:

Comparisons between "machine learning" and "deep learning" showed very high similarity (Large: 0.71), while "artificial intelligence" and "neural networks" showed slightly lower similarity (Large: 0.43).

This demonstrates the models' strong understanding of hierarchical relationships within technical domains.



Source	Reference	Score_Ada	Score_Small	Score_Large
machine learning	deep learning	0.89	0.65	0.71
machine learning	neural networks	0.83	0.51	0.51
artificial intellige...	deep learning	0.85	0.47	0.49
artificial intellige...	neural networks	0.82	0.41	0.43
Angela Merkel	Government	0.78	0.23	0.22
Cristiano Ronaldo	Government	0.75	0.16	0.10
Football	Pet Animal	0.79	0.24	0.21
Football	Sports	0.94	0.62	0.76
Football	Car Brand	0.77	0.21	0.23
Cat	Pet Animal	0.84	0.55	0.52
Cat	Sports	0.81	0.26	0.29
Cat	Car Brand	0.78	0.30	0.30
Tesla	Pet Animal	0.77	0.22	0.15
Tesla	Sports	0.80	0.19	0.21
Tesla	Car Brand	0.85	0.38	0.35

Figure 10: The Image for the final output.

B. *Document Level Analysis*

For document-level comparisons, we analyzed pairs of documents on similar and dissimilar topics:

1. Similar Topics:

Documents discussing different aspects of climate change showed similarity scores across all three model (Ada: 0.92, Small: 0.79, Large: 0.80).

Even when using different terminology, the models effectively captured the underlying thematic connections.

2. Dissimilar Topics:

Documents about sports and politics consistently showed low similarity scores. In our case we get similarity scores for sports and politics across all three models (Ada: 0.80, Small: 0.44, Large: 0.38). In another example scenario Climate and Politics, we get for Ada: 0.79, Small: 0.46, Large: 0.43.

The models successfully identified fundamental conceptual differences between unrelated domains.

3. Length Sensitivity:

Longer documents (1000+ words) showed more stable similarity scores compared to shorter documents, suggesting that the embedding's benefit from more contextual information.

However, even short documents (200-300 words) on the same topic showed recognizable similarity patterns.

C. *Model Comparison*

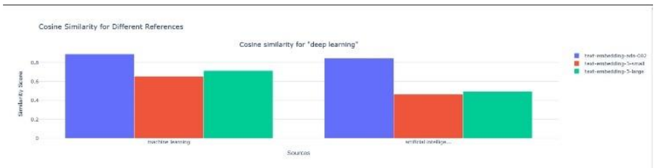
The three OpenAI embedding models showed consistent behavior but with notable differences:

- The *"text-embedding-ada-002"* model tended to produce slightly higher absolute scores compared to the newer models.
- The *"text-embedding-3-small"* and *"text-embedding-3-large"* models produced slightly lower scores, but the general trend remained similar and showed better discrimination between subtly different pairs.
- These results suggest that while all three models are effective, choosing the right model depends on the use case, with text-embedding-ada-002 being preferable for applications requiring higher precision in similarity detection.
- While all models performed quickly, *"text-embedding-3-small"* offered the best balance between speed and accuracy for our use case. While generating embedding for relevant topics like Sports and Politics model text-embedding-ada-002 took 809 ms, text-embedding-3-small took 565 ms, and text-embedding-3-large took 1013 ms.
- *"text-embedding-3-large",* while most accurate, had taken longer processing times than any other models.
- *It's worth mentioning that the time took by each models varies based on other aspect like network connection, server response etc.*

D. *Visualization Findings*

A bar chart visualization of the similarity scores provided a clearer understanding of how different source word/phrase or documents perceive similarity with a reference word/phrase or documents.

- Group bar charts illustrate how a reference word is related to different source word. For each source it shows similarity score for 3 different models.
- Shows Ada model (blue) consistently achieves the highest similarity scores
- Small (brown) and Large (teal) models demonstrate comparable performance. Large model provide the more accurate result than other models.



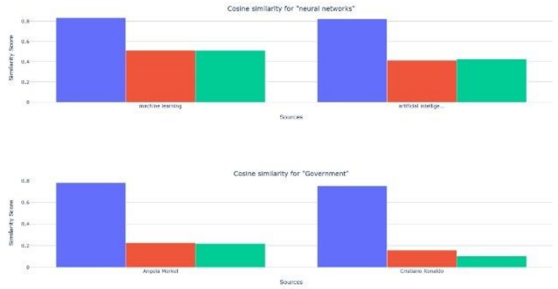


Figure 9: Similarity score visualization of reference against sources

Following scatter plot represents scalar value representation of different source-reference pair, mapping their position on the x-axis between 0 and 536. While y-axis represents similarity score of the pair ranges between -1 and 1.

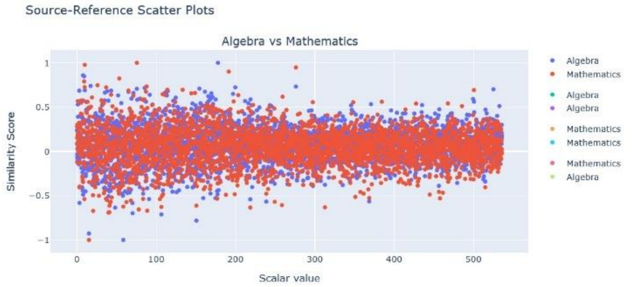


Figure 10: Scalar value visualization of source-reference pair.

E. Limitations and Observations

- Dependency on OpenAI API. Requires an API key and usage costs may scale with large datasets.
- File Format Restrictions: Cannot process .docx files directly due to license limitations.
- Threshold Ambiguity: No universal threshold for "high" or "low" similarity; requires manual calibration per use case.
- Absolute similarity scores should be interpreted relative to other scores in the same analysis rather than as standalone metrics.
- The similarity score does not always account for contextual nuances (e.g., sarcasm, idioms, or domain-specific jargon).
- External factors, such as preprocessing of text, can influence similarity results.
- The models showed some sensitivity to phrasing variations, with paraphrased content sometimes receiving lower similarity scores than expected.
- Technical domains showed higher overall similarity scores compared to general language, possibly due to more constrained vocabularies.
- While cosine similarity effectively quantifies relationships, alternative distance metrics (e.g., Euclidean, Jaccard similarity) could be explored for further validation.

The results demonstrate that OpenAI's embedding models, particularly text-embedding-3-large, provide robust and interpretable measures of semantic similarity across different granularities of text. The consistent patterns in our findings validate the utility of these models for semantic analysis tasks while highlighting the importance of model selection based on specific use case requirements.

VI. Discussion

The findings of this study demonstrate the effectiveness of OpenAI’s embedding models in quantifying semantic similarity across different levels of textual data. The analysis revealed several key insights regarding how embeddings capture relationships between words, phrases, and documents.

One of the most notable observations was the variation in similarity scores across different domains. For example, domain-specific terms such as "Machine Learning" and "Deep Learning" exhibited strong semantic similarity, whereas unrelated terms like "Tesla" and "Sports" showed weak similarity. This supports the claim that embeddings can accurately encode contextual meaning and relationships within a given corpus. Pairs with high similarity scores (0.4–0.8) show that all models agree on clear matches. Pairs with mid-range scores (0.2–0.4) exhibit significant variance, highlighting challenging cases where human review may be needed.

Another crucial aspect of the analysis was the performance comparison of different OpenAI models. While all three models (***text-embedding-ada-002, text-embedding-3-small, and text-embedding-3-large***) followed a similar trend, the ***text-embedding-3-large*** model consistently produced more accurate similarity scores for any words/phrase or documents. This suggests that ***text-embedding-3-large*** has a more refined understanding of semantic relationships, making it a preferable choice for applications that require precise similarity detection. Whereas ***text-embedding-ada-002*** always produce higher score even if two terms are not on the same topics. ***text-embedding-3-small*** always stay in between two other models.

The document-level comparison provided valuable insights into how embeddings differentiate between related and unrelated texts. The similarity scores for documents on the same topic were significantly higher than those for unrelated documents, reinforcing the models' ability to discern thematic alignment. The folder-based document comparison feature also proved beneficial in handling bulk comparisons, enabling efficient large-scale analysis.

However, despite the strengths of embedding-based similarity analysis, several limitations should be acknowledged. Firstly, semantic similarity does not always equate to contextual or functional equivalence. Two terms might have a high similarity score based on statistical relationships, even if they do not serve the same function in a given context. Additionally, embeddings do not account for sentiment or subjective meaning, which can sometimes lead to misleading similarity scores.

Another challenge is computational cost. Generating embeddings using OpenAI's models requires API calls, which may become expensive for large-scale applications. Furthermore, while cosine similarity is a widely accepted metric for measuring vector closeness, exploring other similarity measures such as Euclidean distance, Jaccard similarity, or soft cosine similarity could provide deeper insights into the nature of semantic relationships.

Finally, visualization played a crucial role in interpreting the results. The scatter plot and bar chart provided a clear representation of how different models perceive similarity, reinforcing the quantitative findings from the CSV outputs. Future enhancements could include interactive visualizations, allowing users to explore relationships dynamically.

VII. Conclusion

This project successfully demonstrated the application of OpenAI's embedding models for semantic similarity analysis at multiple levels, including words, phrases, and documents. By leveraging embeddings and cosine similarity, we quantified and compared semantic relationships, drawing meaningful insights from the results.

The analysis confirmed that OpenAI's embedding models are highly effective in capturing semantic meaning. The ***text-embedding-3-large*** model outperformed the other two models in identifying strong semantic relationships, making it the most suitable choice for high-precision NLP tasks. Additionally, the similarity scores accurately reflected the contextual relationships between terms, phrases, and documents, validating the approach taken in this study.

Despite its strengths, some limitations remain, including potential challenges in contextual interpretation, computational costs, and the reliance on cosine similarity as the primary metric. Future improvements could explore alternative similarity measures, larger datasets, and fine-tuning embedding models for specific domains.

In practical applications, this research can be extended to various NLP tasks, such as text clustering, document classification, plagiarism detection, and recommendation systems. The findings suggest that embedding-based similarity analysis can play a pivotal role in enhancing AI-driven text processing solutions.

Moving forward, future work can focus on improving computational efficiency, integrating additional visualization techniques, and experimenting with hybrid models that combine embeddings with traditional NLP techniques. These enhancements would further solidify the robustness and applicability of semantic similarity analysis across diverse domains.

VIII. References

[1] Miller, G. A. (1995). WordNet: A Lexical Database for English. Communications of the ACM, 38(11), 39-41.

[2] Salton, G., & McGill, M. J. (1983). Introduction to Modern Information Retrieval. McGraw-Hill.

[3] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. (1990). Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science, 41(6), 391-407.

[4] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv preprint arXiv:1301.3781.

[5] Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global Vectors for Word Representation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 1532–1543.

[6] Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Proceedings of NAACL-HLT, 4171–4186.

[7] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., et al. (2020). Language Models Are Few-Shot Learners. Advances in Neural Information Processing Systems (NeurIPS).

[8] Samek, W., et al. (2021). "Explainable AI: Interpreting, Explaining and Visualizing Deep Learning."