**Semantic Similarity Analysis of Textual Data**

|  |  |  |  |
| --- | --- | --- | --- |
| Md Abdul Ahad  abdul.ahad@stud.fra-uas.de | Naeem  aaaa@stud.fra-uas.de | Faraz  Muhammad.abbasi@stud.fra-uas.de | Haimonti  aaaa@stud.fra-uas.de |

***Abstract—The document text similarity measurement and analysis is a growing application of Natural Language Processing. This paper presents the results of using different techniques for semantic text similarity measurements in documents used for safety-critical systems. The research objective of this work is to measure the degree of semantic equivalence of multi-word sentences for rules and procedures contained in the documents on railway safety. These documents, with unstructured data and different formats, need to be preprocessed the set of Natural Language Processing toolkits, and Cosine similarity metrics are applied. The results demonstrate that cosine similarity to compare embedding’s and generates CSV reports of similarity scores.***

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

INTRODUCTION

Semantic similarity measures the degree of equivalency between two textual entities, which can be words, phrases, sentences, or entire documents. When applied to sentences or documents, this concept is often referred to as Semantic Textual Similarity (STS). It plays a crucial role in Natural Language Processing (NLP) and is widely used in applications such as document classification, semantic search, information retrieval, question-answering systems, sentiment analysis, and plagiarism detection. 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.

Distributed representation, commonly known as word embedding, is based on the distributional hypothesis, which states that words appearing in similar contexts tend to have similar meanings. This concept is fundamental to many Natural Language Processing (NLP) techniques, as it enables computers to capture and understand word relationships beyond simple lexical matching. Various embedding techniques have been developed to improve the accuracy of text representation, allowing models to interpret words based on contextual meaning rather than mere surface-level similarities.

This project uses OpenAI's embedding API (specifically, we used text-embedding-ada-002 and text-embedding-3-large) 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.

**Methodology**

**EmbeddingGenerator**  
This class is responsible for generating word embedding’s using OpenAI's embedding API. Word embedding’s are numerical representations of words or text, which help in performing semantic similarity calculations.  
  
The provided C# 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. ​

|  |
| --- |
| GenerateEmbeddingsAsync(string content, string model)  public async Task<float[]> GenerateEmbeddingsAsync(string content, string model = "text-embedding-3-large")  {  if (string.IsNullOrWhiteSpace(content))  {  throw new ArgumentException("Content cannot be null or empty");  }  try  {  EmbeddingClient openAIClient = new EmbeddingClient(model, \_apiKey);  OpenAIEmbedding embedding = await openAIClient.GenerateEmbeddingAsync(content);  return embedding.ToFloats().ToArray();  }  catch (Exception ex)  {  // Log the exception and rethrow a more specific exception  Console.WriteLine($"Error generating embeddings: {ex.Message}");  throw new InvalidOperationException("Failed to generate embeddings.", ex);  }  } |

Mathematical Explanation:

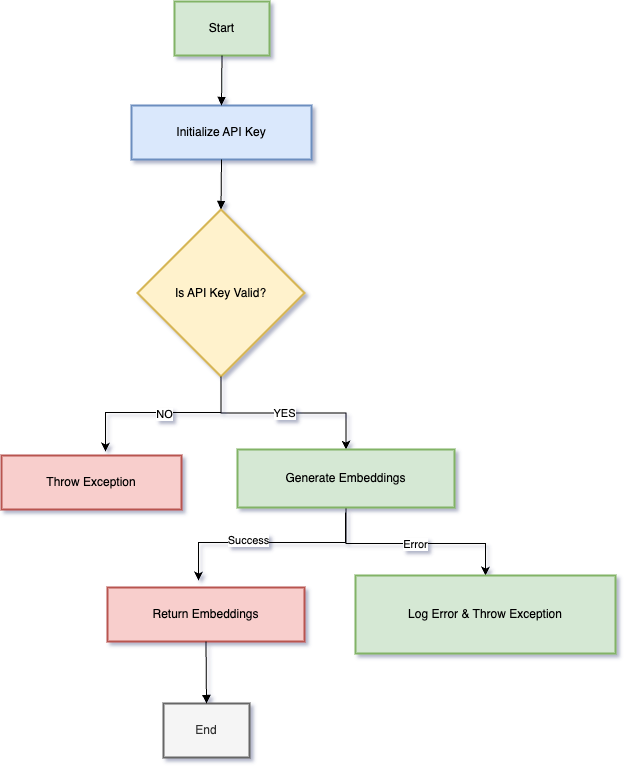
Here we have mantion an embedding is a vector representation of text. Suppose allows words or sentences to be compared mathematically based on their meaning. Let we have two sentences: "I love programming" & "Coding is fun"

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: 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.

**InputHelper**

The “GetFileContents” method in the “InputHelper” 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 “MultipleFileSimilarityProcessor” 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.

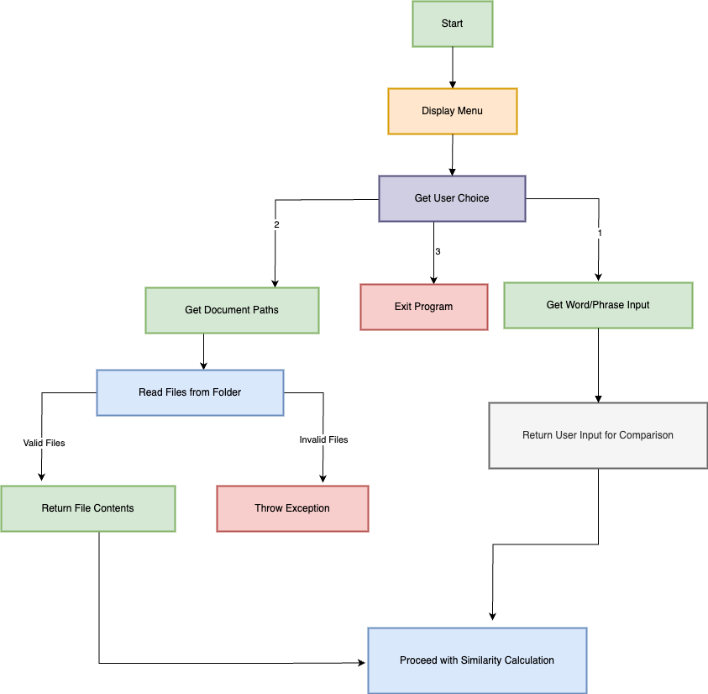
|  |
| --- |
| **GetFileContents(string folderPath)**  public List<string> GetFileContents(string folderPath)  {  if (!Directory.Exists(folderPath))  {  throw new ArgumentException("The specified folder path is invalid or does not exist.");  }  var files = Directory.GetFiles(folderPath, "\*.\*", SearchOption.TopDirectoryOnly)  .Where(file => file.EndsWith(".txt", StringComparison.OrdinalIgnoreCase) || file.EndsWith(".pdf", StringComparison.OrdinalIgnoreCase))  .ToList();  if (files.Count == 0)  {  throw new ArgumentException("The folder does not contain any .txt or .pdf files.");  }  var fileContents = new List<string>();  foreach (var file in files)  {  try  {  var multiFileProcessor = new MultipleFileSimilarityProcessor();  string content = multiFileProcessor.ReadFileText(file);  fileContents.Add(content);  }  catch (Exception ex)  {  throw new IOException($"An error occurred while reading the file: {file}", ex);  }  }  return fileContents;  } |

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()**  public (List<string>, List<string>) GetUserInputs()  {  List<string> sourceContents = new List<string>();  List<string> refContents = new List<string>();  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);  }  return (sourceContents, refContents);  } |

Purpose of collects **multiple source and reference texts** from the user. Allows users to manually enter **text for similarity analysis**.

Designed to facilitate user interaction by collecting two sets of textual inputs: "Source Input" and "Reference Input." Upon completion, the method returns a tuple containing both lists. This structured approach ensures that the application can efficiently gather and manage user-provided data for subsequent processing.



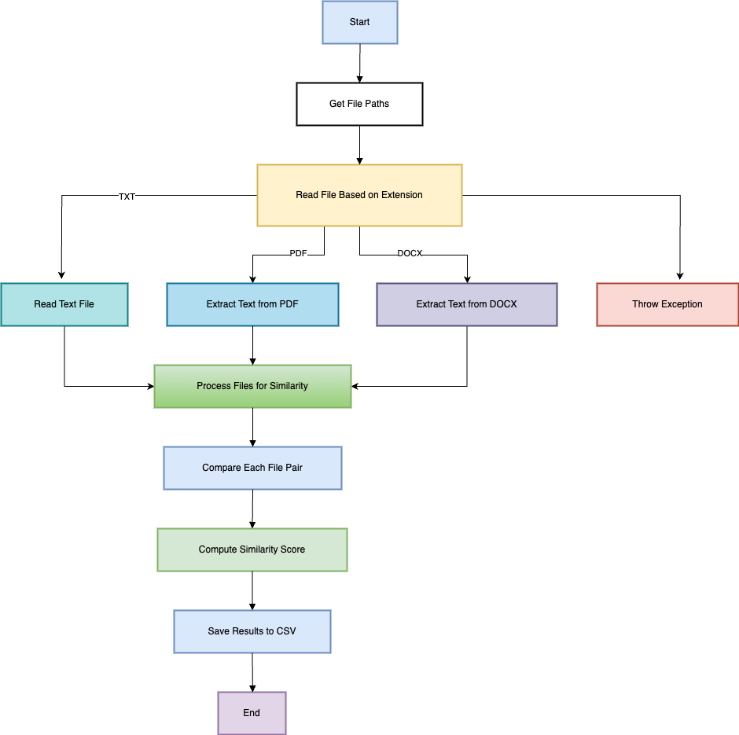
*Figure: Graphical Representation of Implemented Draw Input Helper*

**Multiple File Similarity**

The “MultipleFileSimilarityProcessor” 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 “MultipleFileSimilarityProcessor” 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.**

|  |
| --- |
| ReadFileText(string filePath)  public string ReadFileText(string filePath)  {  string extension = Path.GetExtension(filePath).ToLower();  if (extension == ".txt")  {  return File.ReadAllText(filePath);  }  else if (extension == ".pdf")  {  return ExtractTextFromPdf(filePath);  }  else if (extension == ".docx")  {  return ExtractTextFromDocx(filePath);  }  else  {  throw new NotSupportedException($"Unsupported file format: {extension}");  }  }  ExtractTextFromPdf(string filePath)  private string ExtractTextFromPdf(string filePath)  {  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;  }  } |

The MultipleFileSimilarityProcessor.cs class is designed to handle the extraction of text from various document formats, including .txt, .pdf, and .docx. Its primary function, ReadFileText(string filePath), begins by determining the file's extension to identify its type. If the file is a .txt file, the method reads its entire content using File.ReadAllText(filePath). For .pdf files, it calls the ExtractTextFromPdf(filePath) method, which utilizes the iText 7 library to parse the PDF and extract text from each page. While the implementation for .docx files isn't provided, it typically involves using libraries like Open XML SDK to read the document's content. If the file's extension doesn't match any of these formats, the method throws a NotSupportedException, indicating that the file type isn't supported. This structured approach ensures that the processor can handle multiple file types appropriately, extracting text in a consistent manner for further processing.



*Figure: Graphical Representation of Implemented Draw MultipleFile Similarity Processor*

**Mathematical Representation**

Let:

* PP = Number of pages in the PDF
* TiT\_i = Extracted text from page ii

Total extracted text:

|  |
| --- |
| **ExtractTextFromDocx(string filePath)**  private string ExtractTextFromDocx(string filePath)  {  using (DocX document = DocX.Load(filePath))  {  return document.Text;  }  } |

**OutputHelper**

The “OutputHelper” 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.

1. **Determine the Output File Path**

|  |
| --- |
| string projectRoot = Directory.GetParent(AppContext.BaseDirectory).Parent.Parent.Parent.FullName;  string outputFilePath = Path.Combine(projectRoot, "Output", "similarity\_results.csv"); |

1. **Initialize Variables for Processing**

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.

|  |
| --- |
| var results = new List<SimilarityResult>();  int totalPairs = sourceContents.Count \* refContents.Count;  int processedPairs = 0; |

1. **Calculate Similarity Scores**

Then, the function moves on to calculate the similarity scores using the “SimilarityHelper” 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 “SimilarityHelper” 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.

|  |
| --- |
| SimilarityHelper similarityHelper = new SimilarityHelper();  var result = new SimilarityResult  {  Source = source.Length > 20 ? source.Substring(0,20) + "..." : source,  Reference = refr.Length > 20 ? refr.Substring(0,20) + "..." : refr,  Score\_Ada = await similarityHelper.CalculateSimilarityAsync("text-embedding-ada-002", source, refr),  Score\_Small = await similarityHelper.CalculateSimilarityAsync("text-embedding-3-small", source, refr),  Score\_Large = await similarityHelper.CalculateSimilarityAsync("text-embedding-3-large", source, refr)  };  results.Add(result); |

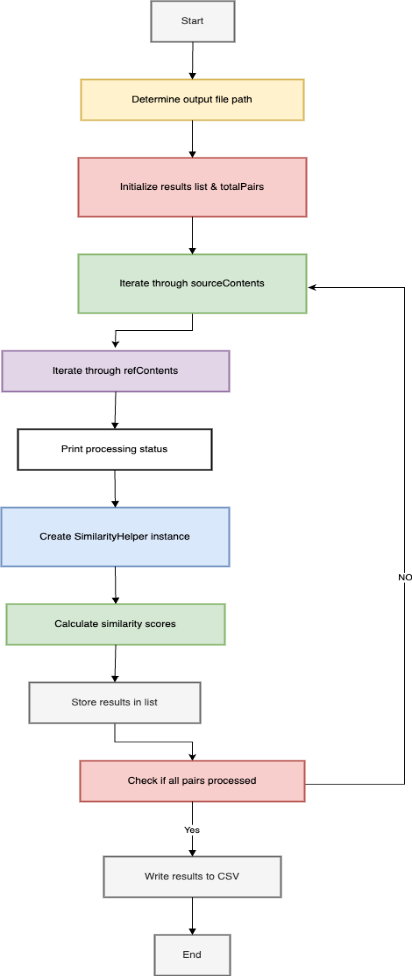
Equation for Similarity Score Calculation

The similarity score is computed using cosine similarity on the embeddings of source and reference:

Where:

Each of the three models (text-embedding-ada-002, text-embedding-3-small, text-embedding-3-large) provides its own embedding representation, leading to different similarity scores.

The OutputHelper.cs class is designed to manage the processing and output of similarity results between source and reference text content. The process begins by determining the output file path: it navigates the directory structure to locate the project's root directory and constructs a path to save the results as similarity\_results.csv within an Output folder. Next, the class initializes variables to handle the similarity calculations: a list named results is created to store individual similarity results, and two integers, totalPairs and processedPairs, are calculated to track the total number of comparisons and the number of comparisons processed, respectively. The core of the class involves calculating similarity scores for each combination of source and reference text. It utilizes the SimilarityHelper class to asynchronously compute similarity scores using different models, such as "text-embedding-ada-002," "text-embedding-3-small," and "text-embedding-3-large." For each pair, a SimilarityResult object is created, storing truncated versions of the source and reference texts (limited to 20 characters for brevity) along with their computed similarity scores. These results are then added to the results list. This structured approach ensures efficient calculation and organization of similarity data, facilitating subsequent analysis or storage operations.



*Figure: Graphical Representation of Implemented Draw* Output Helper

**SimilarityHelper**

The SimilarityHelper 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).

Compute the Dot Product and Magnitudes

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];

}

**Dot Product:** Measures how much two vectors align.

**Magnitude (Euclidean Norm):** Represents the length of each vector.

**Return Cosine Similarity**

return dotProduct / (magnitude1 \* magnitude2);

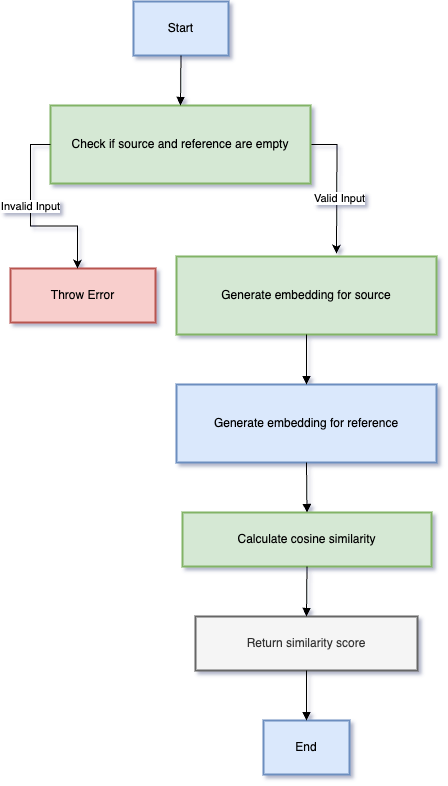
The final cosine similarity formula:

**Range:**

1 → Identical

0 → Unrelated

-1 → Opposite meaning (rare in embeddings)



*Figure: Graphical Representation of Implemented Draw* Similarity Helper

The SimilarityHelper.cs 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.

**Process Overview:**

1. **Model Selection:** The function accepts a modelName parameter, allowing the selection of different machine learning models for similarity computation.
2. **Text Encoding:** It utilizes the OpenAI API's Embeddings endpoint to convert the input texts into vector embeddings. These embeddings are numerical representations capturing the semantic content of the texts.
3. **Similarity Calculation:** Once the embeddings are obtained, the function computes the cosine similarity between them. Cosine similarity measures the cosine of the angle between two vectors, providing a value between -1 and 1, where 1 indicates identical texts, -1 indicates opposite texts, and 0 indicates orthogonality (no similarity).
4. **Error Handling:** The asynchronous nature of CalculateSimilarityAsync ensures that the application remains responsive. If any errors occur during the embedding retrieval or similarity calculation processes, they are caught and rethrown as InvalidOperationException with appropriate messages, ensuring that calling functions can handle exceptions gracefully.

This structured approach enables efficient and flexible similarity assessments between text pairs, supporting various models as specified.