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

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Semantic Similarity Analysis of Textual Data

Muhammad Ahsan Ijaz
*m*uhammad.ijaz@stud.fra-uas.de

Aman Basha Patel
*a*man.patel@stud.fra-uas.de

Saquib Attar
*s*aquib.attar@stud.fra-uas.de

***Abstract*—This paper presents a comprehensive process for semantic similarity analysis of textual data, addressing the growing need for effective methods to process and extract meaningful insights from unstructured content. The framework features a modular architecture that ensures adaptability across various domains while maintaining efficiency through parallel processing and chunking techniques. The implementation includes error handling with retry strategies and automated check pointing for processing large datasets. Experimental results demonstrate the system's effectiveness across diverse applications: distinguishing between word categories with high precision, evaluating skill relevance against resumes, and determining document-level semantic relationships. Results consistently show strong similarity scores for related content and significantly lower scores for unrelated material, confirming the system's ability to capture meaningful semantic relationships. This approach overcomes the limitations of traditional keyword-based methods by capturing contextual relationships, making it particularly valuable for document clustering, content recommendation, anomaly detection, and information retrieval applications. The framework provides a scalable solution for analyzing textual data that delivers accurate results while remaining computationally efficient.**

***Keywords*—Semantic analysis, data extraction, embedding generation, cosine similarity, document similarity, parallel processing, textual data analysis.**

I. Introduction

The rapid growth of textual data has necessitated the development of efficient methods for analyzing and extracting meaningful insights from unstructured content. Semantic similarity analysis plays a crucial role in understanding the meaning and relationships within text, enabling applications such as document similarity comparison, content recommendation, and anomaly detection. Traditional approaches, such as keyword-based matching or basic statistical models often fail to capture complex, deep semantic relationships within textual data [1]. More recent advances in deep learning and embedding techniques have demonstrated improved performance in generating meaningful text representations [2]. However, many existing methods lack a structured, scalable workflow that efficiently integrates data extraction, embedding generation, similarity computation, and result visualization.

This paper presents a comprehensive and modular semantic similarity analysis structure designed to bridge the gap between raw document processing and meaningful content comparison. The framework consists of three core stages as shown in Figure 1: first data extraction and cleaning, second embedding generation using latest models and third document similarity computation using cosine similarity metrics.

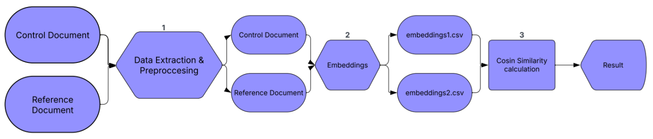


Fig. 1. Graphical representation of Semantic Similarity Analysis of Textual Data

By transforming text into high-dimensional semantic vectors, the system enables precise similarity comparisons beyond simple keyword-based approaches. The key contributions of this work are as follows:

• Comprehensive and Scalable Workflow: Unlike traditional methods that are often computationally expensive or inefficient, our framework integrates parallel processing techniques to optimize performance, enabling real-time or near-real-time analysis of large-scale textual data.
• Enhanced Semantic Representation: The system leverages modern embedding techniques to generate high-dimensional vectors that capture nuanced textual meanings.
• Efficient and Accurate Similarity Analysis: By utilizing cosine similarity, the framework accurately quantifies relationships between documents while minimizing computational overhead.

With its modular architecture and automated processing capabilities, the proposed framework is applicable to a wide range of use cases, including document clustering, content-based recommendation systems, and anomaly detection.

II. Literature Review

The field of semantic text analysis has evolved significantly, driven by advances in natural language processing (NLP), embedding techniques, and document similarity algorithms. Traditional methods for text representation, such as term frequency-inverse document frequency (TF-IDF) and Latent Semantic Analysis (LSA), rely on statistical approaches to measure word importance and co-occurrence patterns.

However, these methods often fail to capture the deeper semantic relationships between words and documents, as they do not consider contextual meaning.

The emergence of word embeddings revolutionized text analysis by introducing dense vector representations that encode semantic similarity. Early models like Word2Vec and GloVe demonstrated that words with similar meanings could be mapped to close proximity in a high-dimensional space, improving the accuracy of similarity measurements [1].

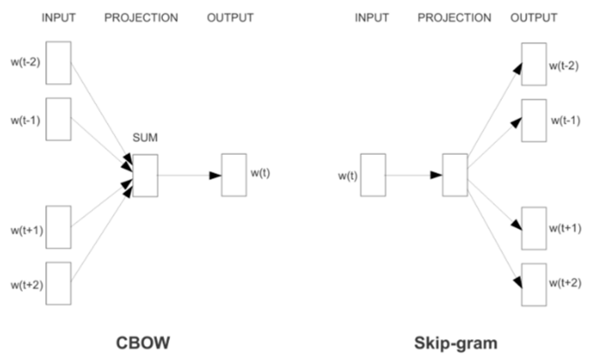


Fig. 2. The Skip-gram model predicts surrounding words based on a given word, allowing it to generate meaningful word embeddings [1]

Word2Vec introduced two primary architectures for learning word embeddings: Continuous Bag of Words (CBOW) and Skip-gram. As shown in Fig. 2, the CBOW model predicts a target word based on its surrounding words, making it computationally efficient for large datasets. In contrast, the Skip-gram model learns to predict surrounding words given a single input word, capturing word relationships more effectively, especially in smaller datasets or with rare words. However, these models generate static word representations that do not account for different contextual meanings of the same word.

To address this limitation, contextual embeddings such as BERT (Bidirectional Encoder Representations from Transformers) and OpenAI GPT-based models introduced dynamic word representations that adapt to sentence context [2]. These transformer-based architectures enable more precise text comparison, as they generate embeddings that capture word sense variations across different contexts. Despite their effectiveness, these models require substantial computational resources, making them less practical for large-scale real-time document analysis.

A. Document Similarity and Cosine Similarity Metrics

Document similarity measurement is a fundamental task in natural language processing (NLP), with applications in document clustering, search ranking, and content recommendation. Traditional methods, such as Euclidean distance and Jaccard similarity, often fail to perform well in high-dimensional spaces, as they do not adequately account for variations in document length and semantic meaning.

Cosine similarity is a widely utilized metric for comparing document embeddings, as it effectively measures the angular distance between two nonzero vectors. This property makes it invariant to variations in text length, ensuring a robust comparison across different document sizes. Cosine Similarity between two vectors A and B is given by [3].

$$\text{Cosine Similarity}(A, B) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

This metric generates a similarity score ranging from -1 to 1, where a score of 1 signifies identical vectors, 0 denotes orthogonality (indicating no similarity), and -1 represents completely opposing vectors [3]. Cosine similarity is a robust and widely applicable metric for assessing text similarity, providing meaningful insights for researchers and practitioners across various domains [3].

B. Large-Scale Semantic Analysis and Processing Challenges

Processing large-scale textual data for evaluation purposes presents challenges related to scalability, efficiency, and accuracy, particularly when assessing the quality of generated text. Traditional evaluation methods often struggle with capturing the nuanced semantic similarities between generated and reference texts. BERTScore, a method based on BERT embeddings, has been proposed to address this issue by leveraging contextualized word representations to calculate the similarity between texts [4]. This method significantly improves performance over traditional evaluation metrics such as BLEU and ROUGE, especially in handling long or complex sentences. Furthermore, BERTScore offers robustness in large-scale NLP workflows by providing more accurate evaluations that are less sensitive to surface-level mismatches, thus enhancing the overall reliability of automatic text evaluation.

C. Contribution of This Work

Building upon existing research, this paper presents a modular and scalable framework for semantic text analysis that integrates:
1. Multi-format document processing, ensuring automated extraction and standardization across various document types.
2. High-dimensional embedding models, leveraging transformer-based architectures for improved text representation.
3. Efficient similarity computation, utilizing cosine similarity with parallelized processing for large-scale text corpora.

III. Methodology

This project implements a Semantic Analysis Pipeline designed to extract, process, and analyze textual data using embeddings and cosine similarity. The methodology follows a structured approach that ensures efficient data handling, vectorization, and similarity computation.

The process begins with Data Extraction from two input files, where text is retrieved from various file formats (.txt, .csv, .json, .pdf, .docx, etc.), cleaned, and stored in a structured JSON format. Next, the Embedding Processor converts the extracted text into numerical representations (embeddings) using OpenAI's Embeddings API, ensuring scalability through chunking and batch processing. The embeddings are then stored in two separate CSV files corresponding to the input files. Finally, the Cosine Similarity module computes the semantic similarity between these CSV files by comparing their embeddings using the cosine similarity metric, which quantifies the closeness of two texts based on their vector representations. This methodology ensures accuracy, scalability, and efficiency, making it suitable for document clustering, similarity detection, and advanced NLP applications.

A. *Preprocessing of Raw Data*

The first step in the methodology involves extracting relevant text data from various input sources like JSON, .txt, and CSV files. To ensure adaptability, configuration settings such as file paths and output directories are loaded from a central configuration file (e.g., appsettings.json), offering flexibility and ease of maintenance. The system automatically checks for and creates necessary output directories, ensuring smooth data processing.

Data extraction is modular, with methods tailored for each input format. Text files (.txt) are read line-by-line, CSV files are parsed record by record, JSON files are deserialized into lists, and XML files are parsed for key elements. Non-text formats like PDF, DOCX, and HTML/Markdown are processed using iTextSharp to extract text from PDFs, Open XML SDK to extract content from DOCX files, and regular expressions to clean and extract readable text from HTML/Markdown files by removing non-textual elements like tags or formatting symbols.

After extraction, the data undergoes cleaning and normalization. This includes removing unnecessary whitespace, symbols, and non-alphanumeric characters, segmenting content into sentences, and converting text to lowercase for consistency. Essential punctuation marks are preserved to retain the text's integrity.

To ensure reusability, consistency, and efficient preprocessing, the raw extracted data is saved into a structured format (typically JSON) before any embedding. This step enhances debugging, flexibility, and future embedding strategies. The cleaned data is then serialized into a structured format (typically JSON) and stored for future use, ensuring it is ready for embedding or similarity analysis. Throughout, the system ensures that directories are created, and any errors are gracefully handled, maintaining robustness and efficiency in the data processing pipeline.

B. *Generating Vector Embeddings*

The text embedding processing system is designed to convert JSON content into numerical vector representations (embeddings) that can be utilized for semantic analysis and similarity computations.

Function reads two JSON files from a specified directory. For each file, the user can choose between two processing strategies: first is breaking down the JSON into individual elements (extracting all values with their respective paths) or second is processing the entire document as a single entity. For text that exceeds the maximum token limit (8,000 tokens), the system employs a chunking mechanism. This approach ensures that even extensive documents can be processed while maintaining the dimensionality (3,072) of the embedding space.

The embedding generation process incorporates robust error handling with an exponential back-off retry mechanism. If the API calls fail, the system attempts up to three retries with increasing delays between attempts. Additionally, if batch processing encounters errors, the system gracefully degrades to processing individual items, ensuring maximum data throughput even under challenging conditions. Use of multiple models can be incorporated, such as "text-embedding-3-small", "text-embedding-ada-002" and text-embedding-3-large.

To manage memory efficiently and provide progress visibility, the system implements a checkpoint mechanism that periodically flushes processed embeddings to disk. This approach not only reduces memory pressure but also provides recovery points in case of system failure during long-running processes.

The final output is saved as CSV files, with each row containing the original text content and its corresponding embedding vector. This format facilitates easy integration with downstream analysis tools and machine learning models.

C. *Cosine Similarity Computation*

The cosine similarity methodology is designed to compute the semantic similarity between pairs of numerical vector representations (embeddings). The process begins by reading embedding vectors from CSV files, where each vector is normalized to ensure consistent computations by dividing each element by the vector's magnitude.

This normalization focuses on the direction of the vectors rather than their scale. The system validates that all vectors have the same dimensionality, a critical requirement for accurate similarity calculations. Using the cosine similarity formula, the system computes the similarity between pairs of vectors as the dot product of the vectors divided by the product of their magnitudes.

The result is a value between -1 and 1, where 1 indicates identical vectors, -1 indicates diametrically opposed vectors, and 0 indicates orthogonality (no similarity). Pairwise similarity computations are performed across vectors from different files, enabling document-level semantic comparisons.

The system incorporates robust error handling by skipping malformed data and logging warnings, ensuring computational integrity. Finally, the results are stored in a structured CSV format, where each row contains the document pair and their corresponding similarity score, enabling seamless integration with downstream analysis tools and machine learning models.

IV. Implementation

The project consists of three main classes: DataExtraction, EmbeddingProcessor, and CosineSimilarity. The DataExtraction class is responsible for reading and processing JSON files to extract relevant textual data. The EmbeddingProcessor class generates vector embeddings from the extracted data and saves them as CSV files for further analysis. Finally, the CosineSimilarity class computes similarity scores between embeddings

A. DataExtraction

The DataExtraction class is responsible for reading and processing data from various input file formats, performing necessary cleaning steps, and ensuring that the extracted data is ready for further use. The implementation begins by loading configuration settings from a central configuration file, typically appsettings.json, using the LoadConfiguration function. This provides flexibility for configuring file paths, output directories, and other parameters without the need for hardcoded values in the code, allowing for easy adaptation to different environments.

The system verifies the existence of necessary output directories using the EnsureDirectoryExists method. If the required directories are absent, they are automatically created to ensure smooth data processing and output generation.

Data extraction is managed through the ExtractDataFromFile method, which handles different file types based on their extensions. It routes the files to corresponding methods for processing, such as text, CSV, JSON, or PDF files. Each extraction method is tailored to the specific format to ensure that relevant data is captured.

Once the data is extracted, it is cleaned using the CleanData method. This process involves several key steps to prepare the data for further analysis. Unwanted whitespace is removed, sentences are segmented using punctuation as delimiters, and non-alphanumeric characters are stripped from the text while preserving meaningful punctuation. If the data is empty or null, an informative message is displayed. The following function demonstrates how this cleaning process is implemented:

Listing 1 Code Reference Example

```
public List<string> CleanData(List<string>data)
{
    if (data == null || data.Count == 0)
    {
        Console.WriteLine
        ("No␣data␣to␣clean.");
        return new List<string>();
    }
    var cleanedData = new List<string>();
    foreach (var line in data)
    {
        string cleanedLine =
        line.Replace("\n", "␣").Trim();
        cleanedLine = Regex.Replace
        (cleanedLine, @"\s+", "␣");

        var sentences = Regex.Split
        (cleanedLine, @"(?<=[.!?])\s+");
        foreach (var sentence in sentences)
        {
            var cleanedSentence =
            sentence.Trim().ToLower();
            cleanedSentence = Regex.Replace
            (cleanedSentence,
            @"[^a-zA-Z0-9\s.,!?'-]", "");
            if (!string.IsNullOrEmpty
            (cleanedSentence))
                cleanedData.Add
                (cleanedSentence);
        }
    }
    return cleanedData;
}
```

After cleaning, the processed data is serialized into JSON format and written to an output file. This is achieved by the SaveDataToJson method, which ensures that the directory exists and handles potential errors gracefully.

B. EmbeddingProcessor

The EmbeddingProcessor class involves reading JSON files, processing content, generating embeddings using OpenAI's API, and saving results to CSV. The process begins with reading JSON files using the ReadJsonFileAsync function, which validates file existence before loading content asynchronously. JSON processing offers two approaches through dedicated methods ProcessWholeJson validates and processes entire documents as cohesive units, while AnalyzeJson recursively traverses JSON structures to extract individual elements with their full path context. For documents exceeding token limits, the SplitIntoChunks method divides content into manageable segments based on an approximation of 4 characters per token. The GenerateEmbeddingWithRetryAsync method shown in code listing 2, implements resilient API communication by automatically retrying failed calls with progressively longer intervals, improving throughput under transient connection issues.

Listing 2 Code Reference Example

```
public async Task<OpenAIEmbedding>
GenerateEmbeddingWithRetryAsync(EmbeddingClient
client, string text, int maxRetries = 3)
{
    int attempt = 0;
    while (attempt < maxRetries)
    {
        try {
            return await client
            .GenerateEmbeddingAsync(text);
        }
        catch (Exception ex) when
        (attempt < maxRetries - 1) {

        }
    }
}
```

When processing large texts, the GenerateChunkedEmbeddingAsync method shwon in code listing 3 divides content into segments, generates embeddings for each, and averages them to produce a cohesive representation within dimensional constraints.

Listing 3 Code Reference Example

```
private async Task<float[]>
GenerateChunkedEmbeddingAsync(EmbeddingClient
client, string text)
{
    int estimatedTokens = text.Length / 4;
    if (estimatedTokens <= MaxTokens)
    {
        OpenAIEmbedding embedding = await

        GenerateEmbeddingWithRetryAsync(client,
        text);
    }
    return embedding.ToFloats().ToArray();
}
```

The GenerateAndSaveEmbeddingsAsync method optimizes throughput by processing descriptions in configurable batches while implementing a periodic checkpoint system that flushes results to disk at specified intervals. This multi-layered implementation ensures the system reliably processes JSON content of varying complexity and size, generating high-quality embeddings while gracefully handling errors and resource constraints. The workflow is orchestrated by the ProcessJsonFileAsync method, which coordinates the entire pipeline from reading input files to saving final outputs, with configuration settings managed through the LoadConfiguration function to ensure flexibility via external settings files.

C. *CosineSimilarity*

The CosineSimilarity class handles data processing, vector normalization, similarity computation, and result management. Vector embeddings are extracted from a CSV file via ReadVectorsFromCsv method, which validates input, extracts text, and parses numerical vectors. If necessary, malformed entries are filtered out to ensure data integrity. Vectors are then normalized using NormalizeVector function converts each vector into a unit vector by dividing its components by the vector magnitude.

The core similarity computation is handled by the method CosineSimilarityCalculation, applying the standard cosine similarity formula. Code snippet is shown below.

Listing 4 Code Reference Example

```
public double CosineSimilarityCalculation
(double[] vectorA, double[] vectorB)
{
    double dotProduct = vectorA
    .Zip(vectorB, (a, b) => a * b).Sum();
    double magnitudeA = Math
    .Sqrt(vectorA.Sum(v => v * v));
    double magnitudeB = Math.
    Sqrt(vectorB.Sum(v => v * v));

    return (magnitudeA == 0 || magnitudeB == 0)
    ? 0.0 :
    dotProduct / (magnitudeA * magnitudeB);
}
```

ValidateVectors method ensures uniform vector dimensions to prevent inconsistencies. CalculateDocumentSimilarity function computes the average cosine similarity across all vector pairs from two input files. The results are stored in a structured format using SaveOutputToCsv method, ensuring reproducibility and ease of further analysis.

V. Results

After cleaning, the processed data is serialized into JSON format and written to an output file. This is achieved by the SaveDataToJson method, which ensures that the directory exists and handles potential errors gracefully. Below are the results of semantic analysis of textual data across different categories, highlighting the similarity scores between reference words and predefined domains.

A. Testcase - Comparing words with different categories.

This testcase was conducted using text-embedding-3-small model. The Table I represent the semantic similarity scores between a set of reference words and different categories, including Technology, Biology, Restaurant, Mathematics, and Arts & Music. Each numerical value represents the similarity score, where a higher value indicates a stronger semantic association between the reference word and the corresponding category. For example, ”Robotics” has the highest similarity with Technology (0,5886), and lowest similarity with Arts & Music (0,3800).

TABLE I
SIMILARITY SCORES ACROSS DIFFERENT CATEGORIES

Reference Words	Technology	Biology	Restaurant	Mathematics	Arts & Music
Robotics.	0,5886	0,4496	0,3330	0,4074	0,3801
Evolution.	0,4625	0,6358	0,3043	0,3923	0,3450
food.	0,5102	0,4232	0,6485	0,4491	0,5177
Symphony.	0,4183	0,3261	0,4188	0,5127	0,6348
Algorithm.	0,4729	0,3795	0,3538	0,5613	0,4889
Photosynthesis.	0,3934	0,4883	0,2975	0,3700	0,4172
Neuron.	0,4220	0,4953	0,3190	0,3799	0,3564
Astronomy.	0,4910	0,4360	0,3200	0,5176	0,4439
dishes.	0,4187	0,3383	0,5286	0,3949	0,4722
Algebra.	0,3960	0,3885	0,3065	0,6994	0,3758
Composition.	0,4439	0,4147	0,3471	0,4601	0,5204
Quantum.	0,4669	0,4076	0,3454	0,4881	0,4489
songs.	0,3310	0,2998	0,3505	0,3315	0,5707
Satellite.	0,4501	0,2964	0,4143	0,3689	0,3793

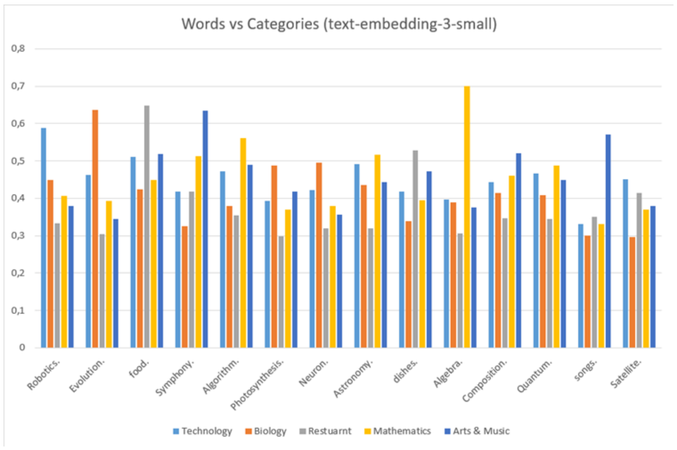


Fig. 3. Graphical representation of semantic anlysis of words against categories.

The bar graph in Figure 3, visualize these results highlighting the distribution of similarity scores across categories. Visually, it emphasizes the stronger associations of words with fields, making patterns more discernible. For instance, bars representing words like ”Algebra” exhibit a clear peak in the mathematics category (0.6993), while ”Photosynthesis” shows a notable association with Biology (0.4883). The graphical representation aids in identifying clusters of words that naturally fit within specific disciplines, reinforcing the accuracy of the semantic analysis in categorizing words based on contextual similarities.

B. Testcase - Evaluating skill relevance based on resume.

This testcase was conducted using text-embedding-3-large model. The Table II represents similarity scores between the two job skills, demonstrating how well the skills align with the given resume. For the Software Developer role, the similarity score with the corresponding resume is 0,5811 indicating a strong match. On the other hand, the Fashion Designer skill set, when compared to the same Software Developer resume, yielded a significantly lower similarity score of 0,3509 highlighting the weaker semantic connection. These results validate the effectiveness of the model in distinguishing relevant from unrelated skill sets.

TABLE II
SEMANTIC ANALYSIS BETWEEN SKILLS AND RESUME

Skills	Role	Reference Document	Similarity
Software developer. Automation testing. GitHub. Jenkins. aws. react. python. java. c++. devops.	Software Developer	Resume (Software Developer)	0,581106929
Sketching. Textiles. Sewing. Draping. Styling.	Fashion Designer	Resume (Software Developer)	0,350948464

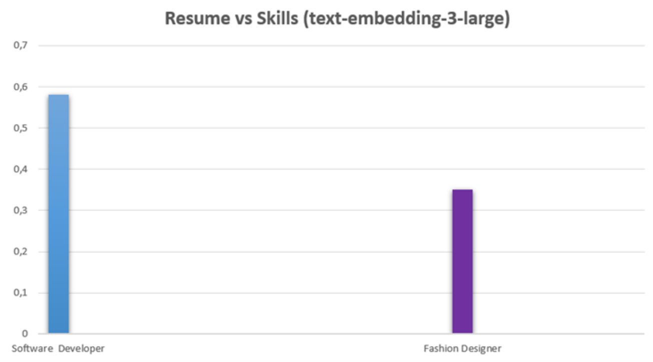


Fig. 4. Graphical representation of semantic analysis of skills against resume.

The bar graph in Figure 4, visually represents these similarity scores, making the differences between the two roles more evident. The Software Developer skill set is positioned higher on the scale, reflecting strong alignment, while the Fashion Designer skills are placed lower, reinforcing the lack of relevance. This graphical representation further illustrates how semantic embeddings can effectively quantify the relationship between a resume and various skills, aiding in automated resume screening and role matching.

C. Testcase - Semantic analysis of documents against reference document.

This testcase was conducted using text-embedding-3-large model. The Table III shows the semantic similarity between different documents by comparing their embeddings with a reference document titled Machine Learning. The table displays the similarity scores, where the Deep Learning document achieved a higher similarity score of 0,5803 indicating a strong conceptual relationship with the Machine Learning reference. In contrast, the Water Pollution document scored 0,3567 demonstrating a weaker semantic connection, as it pertains to an unrelated domain. This test effectively highlights how embeddings capture document-level relevance in various contexts.

TABLE III
SEMANTIC ANALYSIS BETWEEN DOCUMENTS

Documents	Reference Document (Machine Learning)
Water Pollution.	0,356727953
Deep Learning.	0,580319778

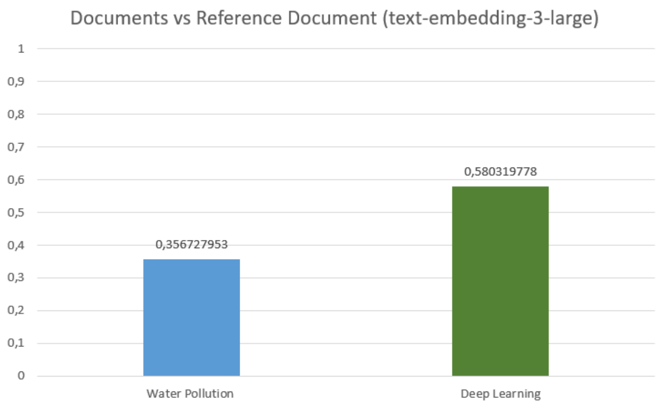


Fig. 5. Graphical representation of semantic analysis between documents.

The bar graph in Figure 5, visually represents these similarity scores, clearly distinguishing the closer relationship between Deep Learning and Machine Learning compared to Water Pollution. The significantly lower score for water pollution reaffirms the model's ability to differentiate between related and unrelated topics, highlighting the strength of document-level semantic analysis for classification and retrieval tasks.

D. Testcase - Semantic Analysis of Skillset against different Resumes

Thi test case was conducted using text-embedding-3-large model. The Table IV presents an evaluation of resume screening performance across multiple technical skills. The "Skills" column lists a set of technical competencies including Software development, Automation testing, GitHub, Jenkins, AWS etc. For each skill, specific resumes (Resume 1 through Resume 10) were evaluated, producing similarity scores that range from approximately 0,38 to 0,64.

TABLE IV
SEMANTIC ANALYSIS BETWEEN SKILL SET AND RESUMES

Skills	**Resumes**	**Similarity Scores**	**Threshold Line**	**Actual Result**	**Expected Result**
Software developer.	Resume 1	0,636275573	0,58	Pass	Pass
Automation testing.	Resume 2	0,580691175	0,58	Pass	Pass
GitHub.	Resume 3	0,554155149	0,58	Fail	Pass
Jenkins.	Resume 4	0,640774694	0,58	Pass	Pass
AWS.	Resume 5	0,545618435	0,58	Fail	Fail
React.	Resume 6	0,498235897	0,58	Fail	Fail
Python.	Resume 7	0,388376861	0,58	Fail	Fail
Java.	Resume 8	0,496273769	0,58	Fail	Fail
C++.	Resume 9	0,57773988	0,58	Fail	Pass
Devops.	Resume 10	0,407527077	0,58	Fail	Fail
MS Office.					
JavaScript.					
Tenserflow.					

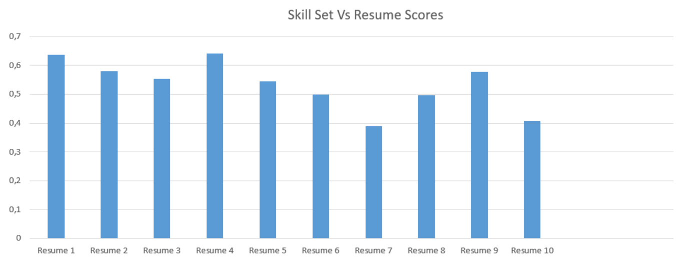


Fig. 6. Graphical representation of Resumes similarity score.

The bar graph in Figure 6, visualizes the values for each resume. Resume 1 and Resume 4 show the highest scores (approximately 0,63-0,65), while Resume 7 shows the lowest score (approximately 0,39). The remaining resumes show moderate scores between these extremes.

The Scatter Plot in Figure 7, visualizes the relationship between similarity scores (blue dots) for each resume and the established threshold line (at 0,58). The x-axis represents the resume number (1-10), while the y-axis indicates similarity scores ranging from -1 to 1.

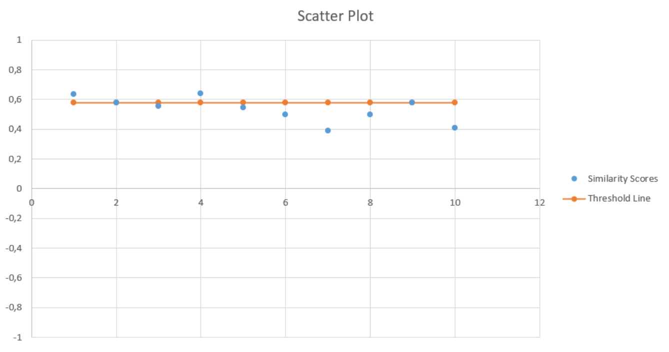


Fig. 7. Scatter Plot of Similarity Scores against Threshold Line.

The screening process employs a threshold value of 0,58 with each resume receiving either a "Pass" or "Fail" designation in the "Actual Result" column in Table IV based on whether its similarity score exceeded this threshold. The "Expected Result" column in Table IV indicates the anticipated outcome.

Notable discrepancies exist between actual and expected outcomes for two cases: Resume 3 (with a score of 0,554) failed despite an expected pass, and Resume 9 (with a score of 0,578) failed despite having a score very close to the threshold and an expected pass outcome.

Out of 10 resumes evaluated, 8 have matching actual and expected results (Resume 1, Resume 2, Resume 4, Resume 5, Resume 6, Resume 7, Resume 8, and Resume 10) 2 have mismatches (Resume 3 and Resume 9) Success rate = 8/10 = 80%.
Actual pass rate: 3 out of 10 resumes passed = 30% Expected pass rate: 5 out of 10 resumes were expected to pass = 50%.

The system therefore has an 80% accuracy rate when comparing its decisions to the expected outcomes, with a 20% difference between actual and expected pass rates.

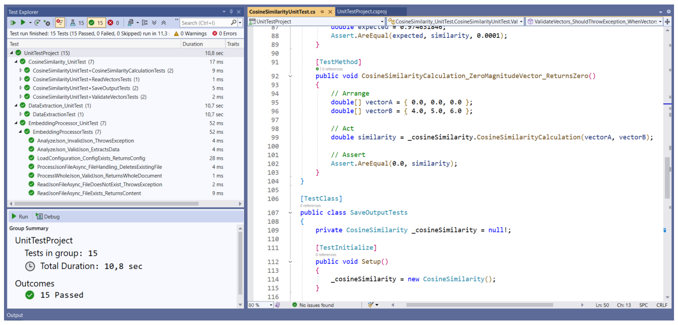
E. Unit Testing

The unit test project successfully verified the functionality of the three core classes within the Semantic Analysis of Textual Data framework. A comprehensive suite of unit tests was executed to ensure that all implemented methods produce accurate and consistent results across diverse input datasets.

As illustrated in Figure 8, the test suite encompasses multiple unit tests, including cosine similarity calculations, data extraction validation, and embedding processor operations. For instance, the CosineSimilarityCalculationTests validate the correctness of similarity computations by assessing various vector scenarios. One such test case, CosineSimilarityCalculation_ZeroMagnitudeVector_ReturnsZero(), ensures that when a zero-magnitude vector is provided, the function correctly returns a similarity score of zero, thereby handling edge cases effectively.

Additionally, the EmbeddingProcessorTests rigorously evaluate the processing of JSON files and their transformation into structured embeddings. The test case AnalyzeJson_InvalidJson_ThrowsException() ensures robust error handling by verifying that improperly formatted JSON inputs trigger the expected exceptions, preventing incorrect data propagation.

This practical validation approach confirms the reliability and robustness of the framework, ensuring its capability to handle real-world textual data analysis efficiently.


Test Explorer
Search (Ctrl+I)
CosineSimilarityUnitTest.cs
UnitTestProject.csproj
UnitTestProject
CosineSimilarity_UnitTest.CosineSimilarityUnitTest.Val
ValidateVectors_ShouldThrowException_WhenVector
Run
Debug
Output
80 %
No issues found
Ln: 50 Ch: 13 SPC CRLF

```
Test run finished: 15 Tests (15 Passed, 0 Failed, 0 Skipped) run in 11,3 s   0 Warnings   0 Errors
Test                                                                        Duration   Traits
UnitTestProject (15)                                                        10,8 sec
  CosineSimilarity_UnitTest (7)                                             17 ms
    CosineSimilarityUnitTest+CosineSimilarityCalculationTests (2)           9 ms
    CosineSimilarityUnitTest+ReadVectorsTests (1)                           1 ms
    CosineSimilarityUnitTest+SaveOutputTests (2)                            5 ms
    CosineSimilarityUnitTest+ValidateVectorsTests (2)                       2 ms
  DataExtraction_UnitTest (1)                                               10,7 sec
    DataExtractionTest (1)                                                  10,7 sec
  EmbeddingProcessor_UnitTest (7)                                           52 ms
    EmbeddingProcessorTests (7)                                             52 ms
      AnalyzeJson_InvalidJson_ThrowsException                               4 ms
      AnalyzeJson_ValidJson_ExtractsData                                    4 ms
      LoadConfiguration_ConfigExists_ReturnsConfig                          28 ms
      ProcessJsonFileAsync_FileHandling_DeletesExistingFile                 4 ms
      ProcessWholeJson_ValidJson_ReturnsWholeDocument                       1 ms
      ReadJsonFileAsync_FileDoesNotExist_ThrowsException                    2 ms
      ReadJsonFileAsync_FileExists_ReturnsContent                           9 ms

Group Summary
UnitTestProject
  Tests in group: 15
  Total Duration: 10,8 sec
Outcomes
  15 Passed
```

```
            double expected = 0.974631846;
            Assert.AreEqual(expected, similarity, 0.0001);
        }

        [TestMethod]
        public void CosineSimilarityCalculation_ZeroMagnitudeVector_ReturnsZero()
        {
            // Arrange
            double[] vectorA = { 0.0, 0.0, 0.0 };
            double[] vectorB = { 4.0, 5.0, 6.0 };

            // Act
            double similarity = _cosineSimilarity.CosineSimilarityCalculation(vectorA, vectorB);

            // Assert
            Assert.AreEqual(0.0, similarity);
        }
    }

    [TestClass]
    public class SaveOutputTests
    {
        private CosineSimilarity _cosineSimilarity = null!;

        [TestInitialize]
        public void Setup()
        {
            _cosineSimilarity = new CosineSimilarity();
        }

```

Fig. 8. Unit Test Project

VI. Applications of Semantic Analysis

Semantic analysis offers a versatile set of tools for extracting deeper insights from textual data. Its applications span several domains:
• Document Clustering and Classification:
By transforming text into high-dimensional semantic vectors, documents can be automatically grouped based on contextual similarities. This is especially useful for organizing large corpora, categorizing research papers, or structuring digital libraries.
• Content Recommendation Systems:
Semantic similarity metrics enable systems to suggest related articles, products, or media. By understanding the underlying meaning of content, recommendations become more accurate and personalized, enhancing user engagement.
• Information Retrieval and Search Optimization:
Enhanced search engines leverage semantic analysis to go beyond keyword matching. By understanding context, these systems can return more relevant results and support effective query expansion.
• Anomaly Detection and Trend Analysis:
By comparing semantic representations, systems can identify outliers or shifts in topic distributions across documents. This capability is crucial in monitoring trends, detecting fraudulent content, or flagging inconsistent data in large-scale text analytics.

VII. Discussion

Semantic analysis transforms text into high-dimensional vectors that capture the nuanced, contextual meaning beyond simple keyword matching. This approach improves accuracy and flexibility in comparing documents, clustering related content, and detecting anomalies. The modular framework integrating data extraction, embedding generation, and cosine similarity computation addresses many limitations of traditional methods, making the system adaptable for various applications such as content recommendation, resume screening, and information retrieval.

Looking ahead, several steps can further enhance semantic analysis:

• Model Fine-Tuning: Tailoring embedding models to specific domains for more accurate representations.

• Real-Time Processing: Optimizing the pipeline for dynamic data environments.

• Multi-Modal Integration: Incorporating images and audio to enrich analysis.

• Interactive Visualization: Developing advanced tools for deeper user-driven insights.

• Adaptive Thresholding: Refining similarity measures to better handle noisy or diverse datasets.

These advancements will build on the current framework, further improving its scalability, precision, and applicability to real world challenges.

VIII. Conclusion

This paper presents a modular framework for semantic text analysis that bridges the gap between raw document processing and meaningful content representation. It achieves this by leveraging word embeddings and similarity algorithms to transform unstructured text into high-dimensional vectors, capturing semantic relationships and contextual meaning more effectively than traditional methods. Our three-stage approach integrating data extraction, embedding generation, and similarity computation provides a robust solution that captures contextual relationships beyond traditional keyword-based methods. The implementation incorporates parallel processing techniques and chunking mechanisms to ensure computational efficiency when handling large-scale datasets. Experimental results demonstrate the framework's effectiveness across multiple use cases, consistently distinguishing between related and unrelated content with high accuracy. The system's ability to transform text into high-dimensional semantic vectors enables applications in document clustering, content recommendation, information retrieval, and anomaly detection. Future work will focus on domain-specific model fine-tuning, real-time processing optimization, multi-modal data integration, and interactive visualization tools. This research contributes to the field of semantic analysis by providing a structured approach to understanding textual data, enabling more intelligent systems capable of processing the growing volume of digital content.

References

[1] T. Mikolov, K. Chen, G. Corrado, and J. Dean, ”Efficient estimation of word representations in vector space,” Proceedings of the International Conference on Learning Representations (ICLR), 2013. https://arxiv.org/pdf/1301.3781

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, ”BERT: Pre-training of deep bidirectional transformers for language understanding,” Proceedings of NAACL-HLT, 2019. https://arxiv.org/pdf/1810.04805

[3] S. Jawale, S. Nehete, H. Patil, S. Pathak, P. Sapale, and S. Zite, “Cosine similarity: A key driver for enhanced recommendation systems,” Int. Res. J. Mod. Eng. Technol. Sci., vol. 6, no. 4, Apr. 2024. https://ijmets.com/irjmets1712589489/e-ISSN: 2582-5208

[4] Zhang, Y., Zhao, R., LeCun, Y. (2020). BERTScore: Evaluating Text Generation with BERT. https://arxiv.org/pdf/1904.09675