ALGORITHMIC SEMANTIC SEARCH USING VECTOR DATABASE APPROACH

Summary of our work:

- Implementation of a custom vector database system (similar to that of ChromaDB,PineCone) with **TF-IDF**-based document embedding for semantic text search, including collections management and **cosine similarity** comparison for query matching.
- Natural language processing pipeline featuring rule-**based lemmatization, stopword removal, and custom tokenization** for both sentences and paragraphs.
- Text preprocessing framework that converts raw documents into numerical vector representations while preserving original text mappings for result retrieval.
- **Interactive visualization interface** built with prompt_toolkit, enabling users to navigate through search results with highlighted relevant sections.
- End-to-end document retrieval workflow that processes text files, embeds content at both sentence and paragraph levels, and **retrieves semantically similar content** based on query relevance.

Basically, we have implemented an improved sentence/paragraph search for a given text file input based on semantic similarity.

DONE BY:

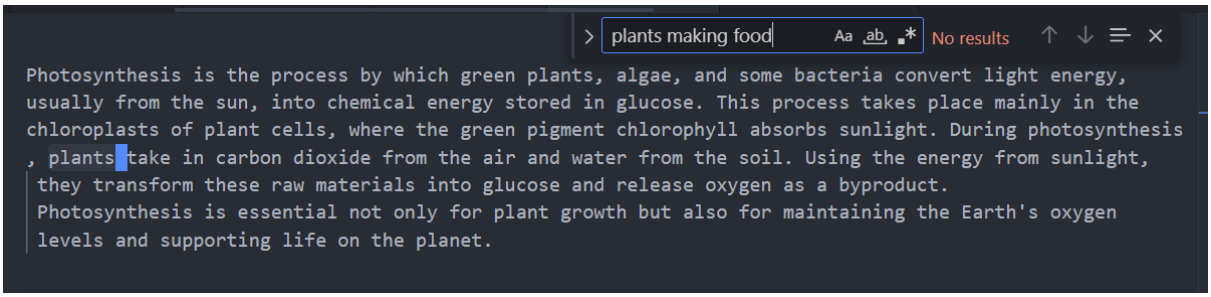
Btech.AI & DS –"B"

SATHYA NARAYANAN (23011101107)

PRANAV B (23011101099)

1. ANALYSIS OF CHALLENGES:

- **Vocabulary gap**: Implementing semantic search requires addressing the fundamental challenge of vocabulary mismatch between queries and documents. Users often formulate queries using different terminology than what appears in the relevant documents, making exact keyword matching insufficient.



(In this screenshot, we are trying to look for the word “photosynthesis” using it’s meaning “plants making food” but the exact matching find doesn’t provide proper results due to the vocabulary gap in the finding tool)

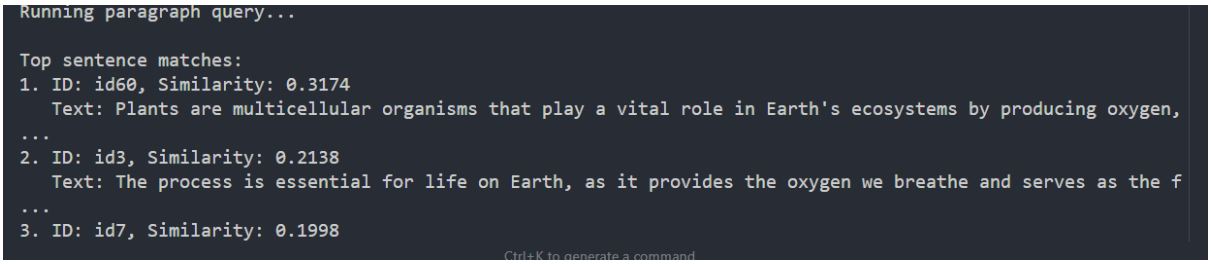
- **Contextual understanding**: Words have different meanings in different contexts. Building a system that correctly interprets the semantic intent of both queries and documents requires sophisticated natural language understanding techniques.

- **Handling long documents**: Processing lengthy text documents presents computational challenges, especially when trying to identify which specific sections are most relevant to a query rather than treating the entire document as a single unit.

- **Language complexity**: Natural language contains ambiguities, synonyms, hypernyms, idioms, and domain-specific terminology that complicate semantic interpretation. Creating accurate semantic representations requires addressing these linguistic nuances.

- **Computational efficiency**: Balancing search quality with performance is crucial. Sophisticated semantic understanding typically requires complex computations that must be optimized for practical real-time search applications.

- **Evaluation methodology**: Defining objective metrics to measure semantic search quality is difficult, as relevance can be subjective and context-dependent. Creating appropriate evaluation frameworks and test cases requires careful consideration.



(The query “plants making foods” returns these top sentence matches with it’s similarity scores. This metric cannot be used to Evaluate the model’s performance as it’s subjective to the query and the text document provided)

- **Feature selection**: Determining which textual features are most important for semantic understanding in the specific domain/use case requires experimentation and domain knowledge to avoid both underfitting and overfitting.

2.Application Selection:

The semantic search application follows similarity-based content retrieval.

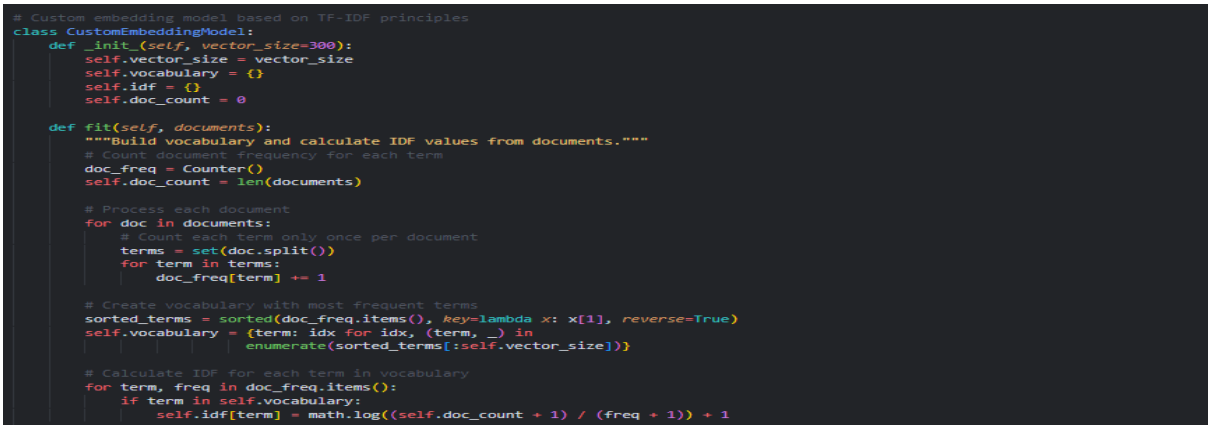
1) Text Ingestion and Preprocessing:

The process begins by loading raw text from user-provided documents. This content is immediately cleaned and standardized to ensure consistency across all downstream steps. Sentence and word tokenization break the input into manageable linguistic units. The **preprocess_text** function orchestrates the full pipeline—removing **stopwords**, normalizing case, and applying **rule_based_lemmatize**, which strips inflectional forms using handcrafted rules. This standardization step is essential for semantic integrity, ensuring that varied word forms (e.g., “running” and “ran”) are treated uniformly.

2) Semantic Representation via TF-IDF-Based Embeddings:

Once the text is preprocessed, it is passed into the **CustomEmbeddingModel**, which vectorizes the data using TF-IDF principles. The model first constructs a vocabulary based on term frequency and document frequency, then encodes each document or sentence into a numerical vector. These vectors reflect the relative importance of terms in context, allowing the system to represent meaning in a form that can be mathematically compared.

CustomEmbeddingModel: Builds a term-frequency and inverse-document-frequency-based vocabulary and encodes the cleaned text into fixed-size numerical vectors. This model plays a central role in capturing the semantic relevance of different documents or queries.



```
# Custom embedding model based on TF-IDF principles
class CustomEmbeddingModel:
    def __init__(self, vector_size=300):
        self.vector_size = vector_size
        self.vocabulary = {}
        self.idf = {}
        self.doc_count = 0

    def fit(self, documents):
        """Build vocabulary and calculate IDF values from documents."""
        # Count document frequency for each term
        doc_freq = Counter()
        self.doc_count = len(documents)

        # Process each document
        for doc in documents:
            # Count each term only once per document
            terms = set(doc.split())
            for term in terms:
                doc_freq[term] += 1

        # Create vocabulary with most frequent terms
        sorted_terms = sorted(doc_freq.items(), key=lambda x: x[1], reverse=True)
        self.vocabulary = {term: idx for idx, (term, _) in
                           enumerate(sorted_terms[:self.vector_size])}

        # Calculate IDF for each term in vocabulary
        for term, freq in doc_freq.items():
            if term in self.vocabulary:
                self.idf[term] = math.log((self.doc_count + 1) / (freq + 1)) + 1
```

3) Vector Storage and Similarity Search:

The semantic vectors are then stored in an in-memory vector database managed by **CustomVectorDatabase** and its collections **CustomCollection**. These classes maintain a record of text units, their IDs, and associated vectors. When a query is later introduced, this vector database allows for efficient cosine similarity comparison between the query vector and all stored vectors—returning ranked, semantically closest matches.

```
# Custom Vector Database Implementation
class CustomVectorDatabase:
    def __init__(self, persistence_dir="./vector_db"):
        self.collections = {}
        self.persistence_dir = persistence_dir
        os.makedirs(persistence_dir, exist_ok=True)

    def get_or_create_collection(self, name):
        if name not in self.collections:
            self.collections[name] = CustomCollection(name)
        return self.collections[name]

class CustomCollection:
    def __init__(self, name):
        self.name = name
        self.documents = []
        self.embeddings = []
        self.ids = []

    def add(self, documents, ids, embeddings):
        self.documents.extend(documents)
        self.embeddings.extend(embeddings)
        self.ids.extend(ids)

    def query(self, query_embeddings, n_results=3):
        results = []
        distances = []

        for query_embedding in query_embeddings:
            similarities = []
            for i, embedding in enumerate(self.embeddings):
                similarity = self._cosine_similarity(query_embedding, embedding)
                similarities.append((similarity, i))

            # Sort by similarity (highest first)
            similarities.sort(reverse=True)

            # Get top results
            top_indices = [similarities[i][1] for i in range(min(n_results, len(similarities)))]
            top_distances = [1 - similarities[i][0] for i in range(min(n_results, len(similarities)))]

            results.append([self.ids[i] for i in top_indices])
            distances.append(top_distances)

        return {
            "ids": results,
            "documents": [[self.documents[self.ids.index(id_)] for id_ in result] for result in results],
            "distances": distances
        }
```

4) Query Processing and Result Matching:

When a user inputs a search query, it follows the same preprocessing and embedding steps as the original text, ensuring alignment in vector space. The **run_query** function encapsulates this workflow—preprocessing the input, generating an embedding, and retrieving the top matches based on cosine similarity. The text corpus itself is prepared and indexed via **process_sentences** and **process_paragraphs**, which batch-process the content, generate embeddings, and register them in the database.

```python
def process_sentences(content, collection, model, stopwords, output_dir=None):
    """Process and store sentences with their embeddings."""
    # Split content into sentences
    raw_sentences = sent_tokenize(content)

    # Save original sentences
    if output_dir:
        with open(f"{output_dir}/original_sentences.txt", "w", encoding="utf-8") as f:
            for i, s in enumerate(raw_sentences):
                f.write(f"Sentence {i+1}: {s}\n\n")

    # Preprocess sentences and get tokens
    cleaned_sentences = []
    all_tokens = []
    for s in raw_sentences:
        if s.strip():
            cleaned, tokens = preprocess_text(s, stopwords, save_tokens=True)
            cleaned_sentences.append(cleaned)
            all_tokens.append(tokens)

    # Save preprocessed sentences
    if output_dir:
        save_preprocessed_text(cleaned_sentences, f"{output_dir}/preprocessed_sentences.txt")

    # Fit model to the corpus
    model.fit(cleaned_sentences)

    # Save model data
    if output_dir:
        model.save_model_data(f"{output_dir}/sentence_model_data.json")

    # Generate embeddings
    ids = [f"id{i+1}" for i in range(len(cleaned_sentences))]
    embeddings = model.encode(cleaned_sentences, batch_size=32, show_progress_bar=True)

    # Save embeddings matrix
    if output_dir:
        save_embeddings_matrix(embeddings, model.vocabulary, f"{output_dir}/sentence_embeddings.txt")

    # Store in collection
    collection.add(documents=cleaned_sentences, ids=ids, embeddings=embeddings)

    # Return mapping from IDs to original sentences
    return dict(zip(ids, raw_sentences))
```

These functions tie together the preprocessing, embedding, and vector management phases into a seamless pipeline.

5) Interactive Result Exploration:

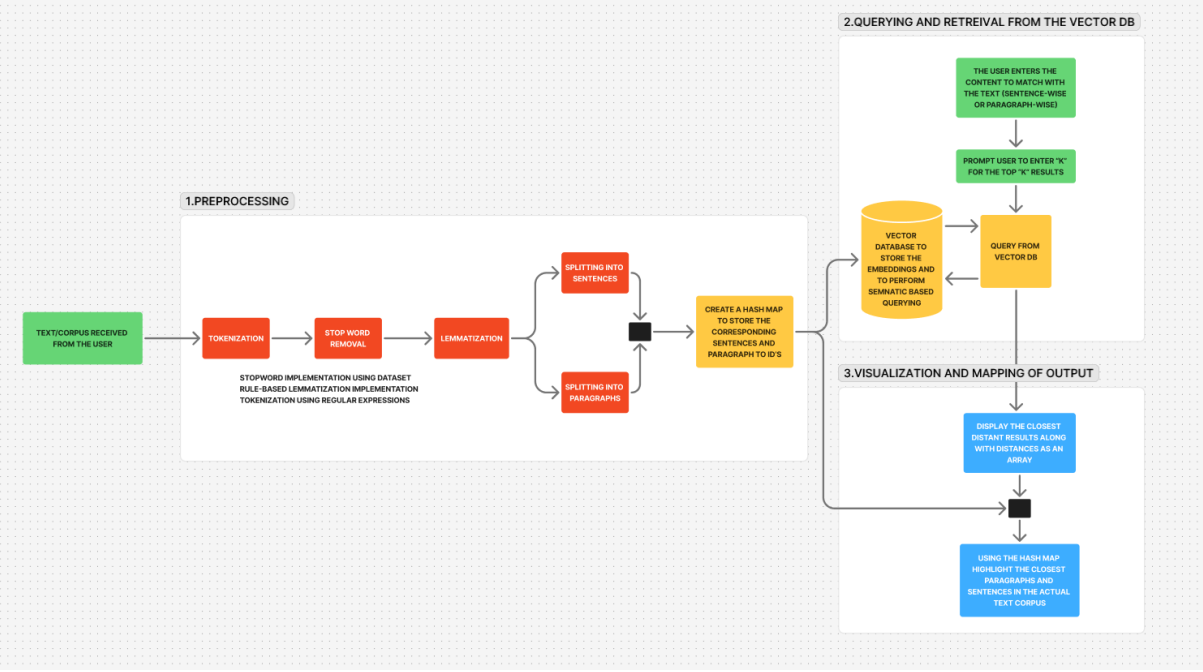
Finally, results are rendered via the **interactive_highlight_view**, a terminal-based UI powered by **prompt_toolkit**. It allows users to navigate through highlighted matches using keyboard controls, with styled formatting to distinguish selected results. Though lightweight, this interface makes the semantic search experience both usable and intuitive.

```python
# Interactive UI using prompt_toolkit
def interactive_highlight_view(highlight_ids, id_map):
    """Create an interactive view to navigate through highlighted text."""
    index = [0]  # Use a list for mutable state

    def get_text():
        """Generate formatted text for display."""
        display = []
        for id_, text in id_map.items():
            if id_ in highlight_ids:
                if highlight_ids[index[0]] == id_:
                    display.append(('[SetCursorPosition]', ''))
                    display.append(('class:selected', f">>> {text} <<<\n\n"))
                else:
                    display.append(('class:highlight', f"{text}\n\n"))
            else:
                display.append(('', f"{text}\n\n"))
        return display

    # Set up key bindings
    kb = KeyBindings()

    @kb.add('down')
    def next_highlight(event):
        index[0] = min(index[0] + 1, len(highlight_ids) - 1)
        event.app.invalidate()

    @kb.add('up')
    def prev_highlight(event):
        index[0] = max(index[0] - 1, 0)
        event.app.invalidate()

    @kb.add('q')
    def exit_(event):
        event.app.exit()

    # Create the UI layout
    content_control = FormattedTextControl(get_text)
    root_container = HSplit([Window(content_control, always_hide_cursor=False, wrap_lines=True)])
    layout = Layout(root_container)

    # Define styles
    style = Style.from_dict({
        "highlight": "#00ff00",
        "selected": "bold underline #ff0066",
    })

    # Create and run the application
    app = Application(layout=layout, key_bindings=kb, full_screen=True, style=style)
    app.run()
```

Though it’s a presentation layer, it’s tightly coupled with the semantic engine and included to support explainability and usability.

3.ARCHITECTURE DIAGRAM:

High Level System Design Architecture:



EXPLANATION:

This system is designed to provide enhanced semantic search capabilities over text documents by implementing a custom vector database with TF-IDF-based embeddings. The pipeline is divided into three main stages:

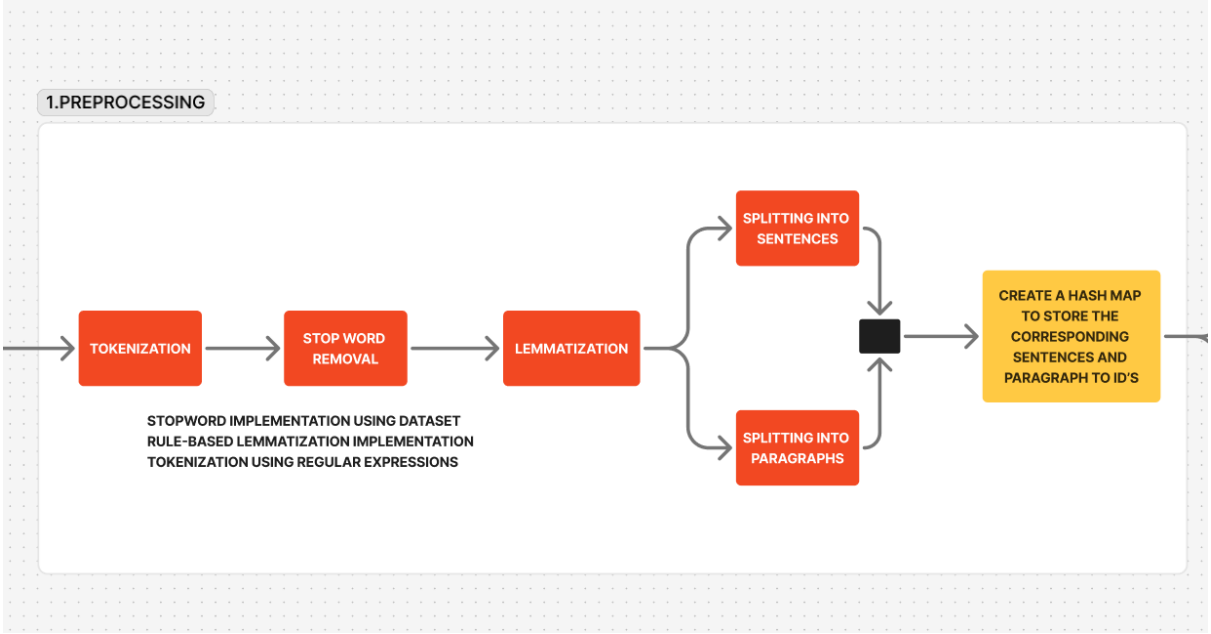
1. **Preprocessing**:
   The input document undergoes a structured natural language processing pipeline. This includes **tokenization**, **stopword removal**, and **lemmatization**, applied at both **sentence** and **paragraph** levels. The cleaned text is then embedded using TF-IDF to generate vector representations while preserving mappings to the original raw text. This allows accurate result retrieval later.
2. **Querying and Retrieval**:
   When a user inputs a query, it is processed through the same preprocessing and embedding pipeline. The resulting vector is compared to the document vectors stored in a custom vector store using **cosine similarity**. The system retrieves the most relevant sentence or paragraph matches based on semantic closeness to the query.
3. **Visualization and Output Mapping**:
   The results of the similarity matching are presented through an

interactive terminal interface built with prompt_toolkit. The interface allows users to browse through matched results with corresponding original text highlighted, enabling clear contextual understanding.

ANALYSIS OF EACH OF THE MAIN STAGES IN THE DESIGN ARCITECTURE PIPELINE: (IN-DEPTH EXPLANATION IS PROVIDED IN THE MODULE DESCRIPTION)

1.PREPROCESSING STAGE:

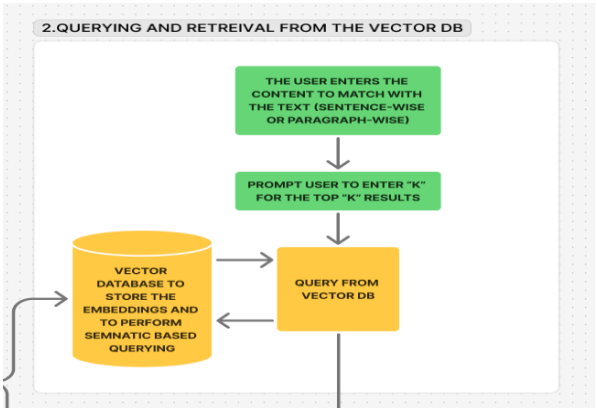
- Sentence Tokenization
  - Splits a paragraph into individual sentences using punctuation like '.', '?', and '!'.
- Word Tokenization
  - Breaks each sentence into words by removing punctuation and converting to lowercase.
- Stopword Removal
  - Eliminates common, uninformative words (e.g., "the", "is") using a stopwords list.
- Rule-based Lemmatization
  - Converts words to their root forms using predefined rules and handling irregular forms.
- Text Cleaning and Integration
  - Applies all steps together to produce a normalized, meaningful text ready for further processing.



(PREPROCESSING STAGE)

2. QUERYING AND RETRIEVAL STAGE

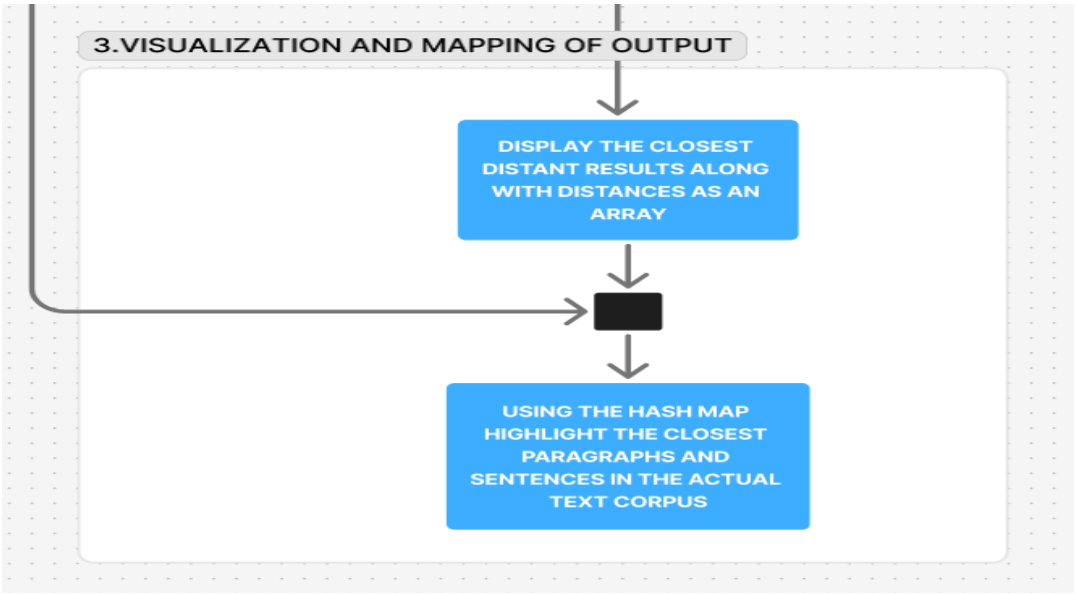
- Fitting the Embedding Model
  - Builds a vocabulary from training documents using term frequency.
  - Computes Inverse Document Frequency (IDF) for each term to weight its importance.
  - Limits the vocabulary to the most frequent `vector_size` terms.
- Encoding Documents
  - Converts documents into TF-IDF vectors.
  - Each term's weight is calculated using its frequency and corresponding IDF value.
  - The vector is normalized to unit length to enable cosine similarity comparison.
- Storing Embeddings in the Vector Database
  - Embeddings along with corresponding document IDs and texts are stored in collections inside a custom vector database.
  - Each collection maintains its own list of embeddings and associated metadata.
- Querying the Vector Database
  - Query texts are encoded using the same TF-IDF method.
  - Cosine similarity is calculated between each query vector and stored document vectors.
  - The top N most similar documents are retrieved based on similarity scores.
  - Returning the Results
  - For each query, the system returns a list of closest document IDs, the documents themselves, and similarity-based distances.
- These results form the basis for downstream applications like answering user questions or generating summaries.



(QUERY AND RETREIVAL STAGE)

3. VISUALIZATION AND MAPPING OF OUTPUT:

- **Query Execution and Result Mapping**
  - The function `run_query` preprocesses a query, converts it into an embedding using the custom model, and retrieves top matching documents using cosine similarity from the vector database.
  - Each retrieved result includes document IDs, content, and similarity distances.
  - Preprocessing details and the embedding vector can optionally be saved for inspection.
- **Highlight Mapping using IDs**
  - Document IDs from query results are matched to corresponding texts using an `id_map` (a dictionary mapping IDs to their full text content).
  - This allows display of only relevant documents, especially the ones that matched the query.
- **Interactive Navigation with `prompt_toolkit`**
  - An interactive terminal interface is created using `prompt_toolkit`.
  - Users can navigate through highlighted documents using the up and down arrow keys.
  - The currently selected document is visually emphasized using styles (e.g., bold, underline).
- **Custom Styling for Highlighting**
  - Retrieved documents are styled based on their relevance:
  - The currently focused match is marked with a distinct style (`selected`).
  - Other matches are marked using a general `highlight` style.
  - Non-matching documents remain unstyled but visible.
- **Terminal Controls for Exploration**
  - Arrow keys (`up`/`down`) change the selected highlight.
  - Pressing `q` exits the interactive view.
  - This setup enables smooth, keyboard-controlled browsing through matched results, improving usability for terminal-based applications.



(VISUALIZATION AND MAPPING OF OUTPUT)

4.MODULE DESCRIPTION:

The semantic search system is composed of carefully architected modules that collectively handle text ingestion, linguistic normalization, semantic embedding, vector storage, and query-based retrieval.

Below is a detailed breakdown of each module, its purpose, the techniques employed, and key implementation highlights.

1) Text Preprocessing Module:

Purpose: Normalize raw input text into a clean, lemmatized format suitable for vectorization.

Key Techniques Used: Regex-based sentence and word tokenization, Custom rule-based lemmatization, Stopword filtering

Core Functions:

sent_tokenize: Splits input text into sentences using punctuation followed by whitespace. This segmentation is crucial for enabling sentence-level semantic embedding and retrieval, allowing fine-grained matching during query time

```
# Intelligent text processing functions
def sent_tokenize(text):
    """Split text into sentences using regex patterns."""
    # Match sentence endings with punctuation followed by space or end of string
    sentences = re.split(r'(?<=[.!?])\s+', text)
    return [s.strip() for s in sentences if s.strip()]
```

word_tokenize: Extracts words from text via regex (\b\w+\b) ensuring case-folded, alphanumeric tokens.By standardizing word boundaries, this function ensures consistent vocalbulary formation across documents

```
def word_tokenize(text):
    """Split text into words."""
    # Split on whitespace and remove punctuation
    words = re.findall(r'\b\w+\b', text.lower())
    return words
```

rule_based_lemmatize: Applies handcrafted suffix-stripping and transformation rules to reduce words to their morphological roots.

This normalization reduces vocabulary sparsity and enhances semantic consistency, improving embedding quality and similarity matching.

It also includes handling of irregular nouns and verbs, which is typically managed by external NLP libraries like NLTK or spaCy—here implemented from scratch for full transparency and control.

preprocess_text: Chains tokenization, stopword removal, and lemmatization. It returns space-separated tokens for further vectorization.

This function serves as the gatekeeper of data quality before any text is encoded into vectors. Its thorough cleaning pipeline ensures uniform representation across the dataset, directly impacting the accuracy and reliability of the semantic search.

```
# Apply rule-based lemmatization
lemmatized_tokens = [rule_based_lemmatize(token) for token in filtered_tokens]
```

This line operationalizes custom lemmatization logic that mimics NLP libraries without external dependencies.

Stop word removal: Stopword removal is performed using a custom stopword list sourced from a Kaggle dataset, provided as a text file. The list can be accessed at the following link

https://www.kaggle.com/datasets/rowhitswami/stopwords

The text file **"stopwords.txt"** is placed in the main project directory and is manually loaded within the code to perform stopword removal during preprocessing.

2) Embedding Module (CustomEmbeddingModel) :

Purpose: Transform preprocessed text into fixed-size semantic vectors using a TF-IDF-inspired method.

Key Techniques Used: Vocabulary construction from top N frequent terms, Log-scaled inverse document frequency (IDF), Vector normalization for cosine similarity

Core Methods:

Fit : Builds the vocabulary and calculates IDF per term across documents.

Encode : Converts tokenized text into sparse, normalized TF-IDF vectors.

```
# Calculate TF-IDF for each term
for term, count in term_freq.items():
    if term in self.vocabulary:
        idx = self.vocabulary[term]
        tf = count / max(len(words), 1)  # Term frequency
        tfidf = tf * self.idf.get(term, 1.0)  # TF-IDF
        vector[idx] = tfidf
```

This fuses term frequency and inverse document frequency, forming the semantic weight of each token in a document vector.

3) Vector Storage & Search Module (CustomVectorDatabase / CustomCollection):

Purpose: Efficiently store and retrieve semantically encoded documents using cosine similarity.

Key Techniques Used: In-memory storage of vectors and metadata, Brute-force cosine similarity for small-scale prototyping, Sorted similarity ranking

Core Methods in CustomCollection:

add(documents, ids, embeddings): Stores document embeddings.

This is a crucial data ingestion point for the system. Once text is preprocessed and vectorized, it's handed off to this method to be persisted in memory. Without it, semantic retrieval would not be possible, as the query engine depends on this indexed structure.

It bridges the preprocessing/embedding pipeline with the retrieval system. The richer and cleaner the added data, the more effective the similarity matching becomes.

```
# Store in collection
collection.add(documents=cleaned_sentences, ids=ids, embeddings=embeddings)
```

query(query_embeddings, n_results):

Calculates cosine similarity and returns top matches.

```
# Query the collection
return collection.query(query_embeddings=[query_embedding], n_results=n_results)
```

similarity = self._cosine_similarity(query_embedding, embedding):

It calculates cosine similarity as:

cosine_similarity(A,B)=A·B||A||·||B||\text{cosine_similarity}(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||}cosine_similarity(A,B)=||A||·||B||A·B

Enables direct vector comparison for semantic proximity scoring.

Cosine similarity is ideal for comparing high-dimensional semantic vectors, as it captures direction-based similarity rather than magnitude. This approach ensures that semantically similar texts (even of different lengths) are accurately aligned.

```
for query_embedding in query_embeddings:
    similarities = []
    for i, embedding in enumerate(self.embeddings):
        similarity = self._cosine_similarity(query_embedding, embedding)
        similarities.append((similarity, i))
```

4) Corpus Processing Module:

Purpose: Automate the end-to-end transformation of raw text into a searchable vector database.

Core Functions: –

process_sentences(content, collection, model, stopwords): Tokenizes content into sentences, preprocesses them, fits the embedding model, and stores the results.

The function begins by using `sent_tokenize` to segment the input content into individual sentences. Each sentence is then processed through `preprocess_text`, which applies lowercasing, removes punctuation, filters out stopwords, and performs rule-based lemmatization.

The cleaned sentences are used to build a vocabulary and compute IDF values via `model.fit()`.

Next, the sentences are converted into TF-IDF-based vector embeddings, with batching used to manage memory efficiently and support performance monitoring during large-scale processing.

Finally, the document texts, unique identifiers, and their corresponding embeddings are stored in the provided `collection` object through `collection.add()`.

This function is essential to the overall system, as it serves as the primary mechanism for sentence-level semantic indexing, enabling accurate and fine-grained retrieval of relevant content in response to user queries.

```python
def process_sentences(content, collection, model, stopwords, output_dir=None):
    """Process and store sentences with their embeddings."""
    # Split content into sentences
    raw_sentences = sent_tokenize(content)

    # Save original sentences
    if output_dir:
        with open(f"{output_dir}/original_sentences.txt", "w", encoding="utf-8") as f:
            for i, s in enumerate(raw_sentences):
                f.write(f"Sentence {i+1}: {s}\n\n")

    # Preprocess sentences and get tokens
    cleaned_sentences = []
    all_tokens = []
    for s in raw_sentences:
        if s.strip():
            cleaned, tokens = preprocess_text(s, stopwords, save_tokens=True)
            cleaned_sentences.append(cleaned)
            all_tokens.append(tokens)

    # Save preprocessed sentences
    if output_dir:
        save_preprocessed_text(cleaned_sentences, f"{output_dir}/preprocessed_sentences.txt")

    # Fit model to the corpus
    model.fit(cleaned_sentences)

    # Save model data
    if output_dir:
        model.save_model_data(f"{output_dir}/sentence_model_data.json")

    # Generate embeddings
    ids = [f"id{i+1}" for i in range(len(cleaned_sentences))]
    embeddings = model.encode(cleaned_sentences, batch_size=32, show_progress_bar=True)

    # Save embeddings matrix
    if output_dir:
        save_embeddings_matrix(embeddings, model.vocabulary, f"{output_dir}/sentence_embeddings.txt")

    # Store in collection
    collection.add(documents=cleaned_sentences, ids=ids, embeddings=embeddings)

    # Return mapping from IDs to original sentences
    return dict(zip(ids, raw_sentences))
```

process_paragraphs(content, collection, model, stopwords):

Identical flow for paragraphs.

This function enables semantic search over larger blocks of text, which is useful for retrieving contextually rich matches (e.g., full ideas or explanations). It complements sentence-level retrieval by offering a broader scope.

```python
def process_paragraphs(content, collection, model, stopwords, output_dir=None):
    """Process and store paragraphs with their embeddings."""
    # Split content into paragraphs
    raw_paragraphs = [p.strip() for p in content.split("\n\n") if p.strip()]

    # Save original paragraphs
    if output_dir:
        with open(f"{output_dir}/original_paragraphs.txt", "w", encoding="utf-8") as f:
            for i, p in enumerate(raw_paragraphs):
                f.write(f"Paragraph {i+1}: {p}\n\n")

    # Preprocess paragraphs and get tokens
    cleaned_paragraphs = []
    all_tokens = []
    for p in raw_paragraphs:
        cleaned, tokens = preprocess_text(p, stopwords, save_tokens=True)
        cleaned_paragraphs.append(cleaned)
        all_tokens.append(tokens)

    # Save preprocessed paragraphs
    if output_dir:
        save_preprocessed_text(cleaned_paragraphs, f"{output_dir}/preprocessed_paragraphs.txt")

    # Fit model to the corpus
    model.fit(cleaned_paragraphs)

    # Save model data
    if output_dir:
        model.save_model_data(f"{output_dir}/paragraph_model_data.json")

    # Generate embeddings
    ids = [f"id{i+1}" for i in range(len(cleaned_paragraphs))]
    embeddings = model.encode(cleaned_paragraphs, batch_size=32, show_progress_bar=True)

    # Save embeddings matrix
    if output_dir:
        save_embeddings_matrix(embeddings, model.vocabulary, f"{output_dir}/paragraph_embeddings.txt")

    # Store in collection
    collection.add(documents=cleaned_paragraphs, ids=ids, embeddings=embeddings)

    # Return mapping from IDs to original paragraphs
    return dict(zip(ids, raw_paragraphs))
```

embeddings = model.encode(cleaned_sentences, batch_size=32, show_progress_bar=True) :

This enables batched embedding generation with optional progress feedback for scaling.

This line is critical for performance and scalability. It allows the system to encode text efficiently in chunks and track progress, which is essential for processing large datasets.

Batched encoding improves resource utilization and can easily be extended to distributed or parallel systems if scaled up.

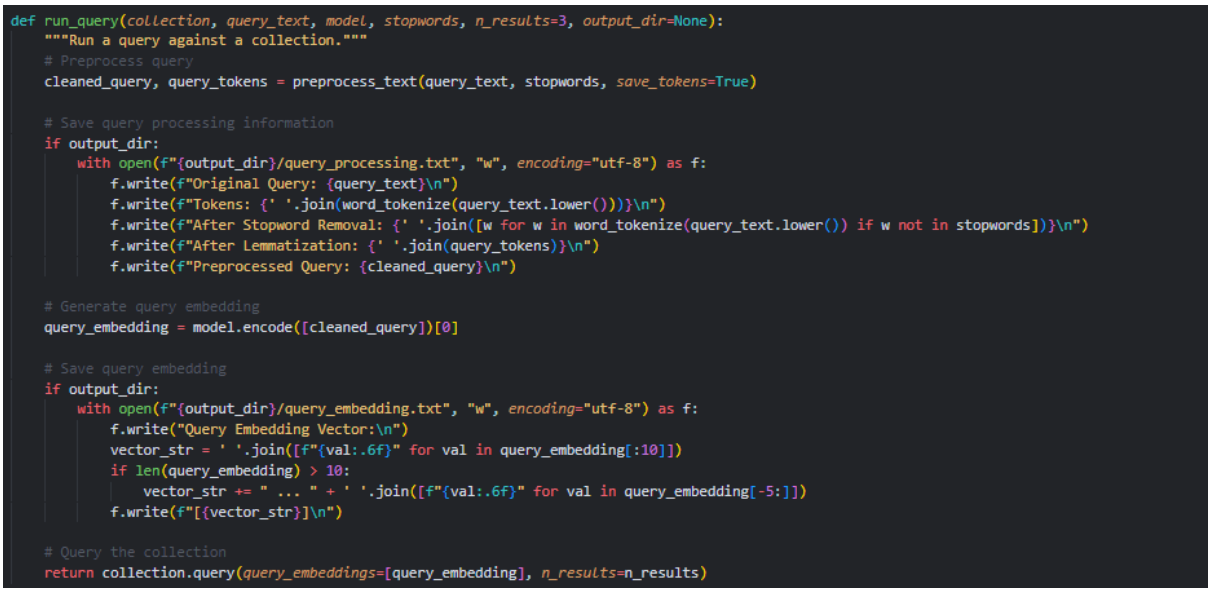
5) Query Handling Module:

Purpose: Handle real-time search requests by embedding queries and matching against stored vectors

Function:

run_query(collection, query_text, model, stopwords, n_results): Preprocesses the user query, encodes it, and retrieves top-N similar results using cosine similarity.

The function is the core of the semantic retrieval process. It starts by preprocessing the user query using the same cleaning steps applied to the dataset—lowercasing, punctuation removal, stopword filtering, and rule-based lemmatization. This ensures the query aligns with the structure of the stored text data.



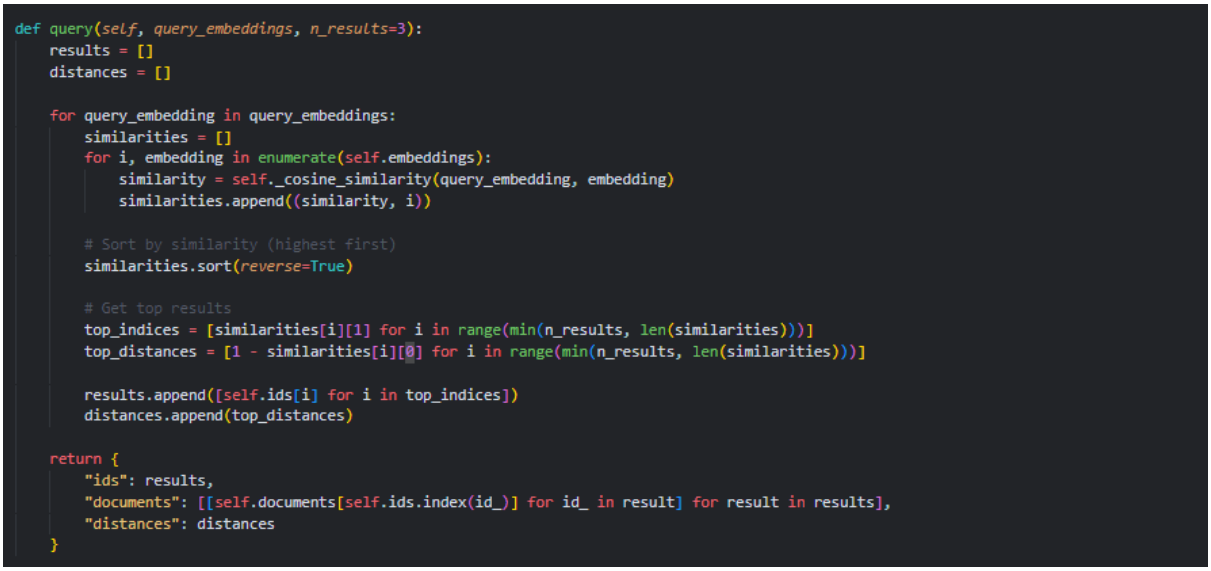
```
def run_query(collection, query_text, model, stopwords, n_results=3, output_dir=None):
    """Run a query against a collection."""
    # Preprocess query
    cleaned_query, query_tokens = preprocess_text(query_text, stopwords, save_tokens=True)

    # Save query processing information
    if output_dir:
        with open(f"{output_dir}/query_processing.txt", "w", encoding="utf-8") as f:
            f.write(f"Original Query: {query_text}\n")
            f.write(f"Tokens: {' '.join(word_tokenize(query_text.lower()))}\n")
            f.write(f"After Stopword Removal: {' '.join([w for w in word_tokenize(query_text.lower()) if w not in stopwords])}\n")
            f.write(f"After Lemmatization: {' '.join(query_tokens)}\n")
            f.write(f"Preprocessed Query: {cleaned_query}\n")

    # Generate query embedding
    query_embedding = model.encode([cleaned_query])[0]

    # Save query embedding
    if output_dir:
        with open(f"{output_dir}/query_embedding.txt", "w", encoding="utf-8") as f:
            f.write("Query Embedding Vector:\n")
            vector_str = ' '.join([f"{val:.6f}" for val in query_embedding[:10]])
            if len(query_embedding) > 10:
                vector_str += " ... " + ' '.join([f"{val:.6f}" for val in query_embedding[-5:]])
            f.write(f"[{vector_str}]\n")

    # Query the collection
    return collection.query(query_embeddings=[query_embedding], n_results=n_results)
```

query_embedding = model.encode([cleaned_query])[0]:

Extracts the embedding for a single cleaned query, enabling real-time semantic retrieval.
The line query_embedding = model.encode([cleaned_query])[0] converts the query into a vector representation that shares the same semantic space as the stored documents. This vector is then passed to collection.query(),

which computes cosine similarity between the query and all stored vectors to retrieve the top-N most relevant matches.

This function is essential to enabling real-time, semantically-aware search, directly linking user intent to meaningful content retrieval. Its accuracy determines the relevance and quality of the system's responses.



```
def query(self, query_embeddings, n_results=3):
    results = []
    distances = []

    for query_embedding in query_embeddings:
        similarities = []
        for i, embedding in enumerate(self.embeddings):
            similarity = self._cosine_similarity(query_embedding, embedding)
            similarities.append((similarity, i))

        # Sort by similarity (highest first)
        similarities.sort(reverse=True)

        # Get top results
        top_indices = [similarities[i][1] for i in range(min(n_results, len(similarities)))]
        top_distances = [1 - similarities[i][0] for i in range(min(n_results, len(similarities)))]

        results.append([self.ids[i] for i in top_indices])
        distances.append(top_distances)

    return {
        "ids": results,
        "documents": [[self.documents[self.ids.index(id_)] for id_ in result] for result in results],
        "distances": distances
    }
```

6) Interactive Visualization Module:

Purpose: Provide an intuitive terminal-based navigation interface for matched results.

Function:

interactive_highlight_view(highlight_ids, id_map):

Displays results with keyboard controls and styled highlighting (via prompt_toolkit).

The function provides an interactive, terminal-based user interface for viewing semantic search results. It is built using the **prompt_toolkit** library and allows users to navigate through the top-matching sentences or paragraphs returned from a query using keyboard inputs.

The function dynamically renders text in the terminal, visually distinguishing selected and highlighted results with custom styles.

Within this function, the nested get_text() method generates formatted text for display.

```python
# Interactive UI using prompt_toolkit
def interactive_highlight_view(highlight_ids, id_map):
    """Create an interactive view to navigate through highlighted text."""
    index = [0]  # Use a list for mutable state

    def get_text():
        """Generate formatted text for display."""
        display = []
        for id_, text in id_map.items():
            if id_ in highlight_ids:
                if highlight_ids[index[0]] == id_:
                    display.append(('[SetCursorPosition]', ''))
                    display.append(('class:selected', f">>> {text} <<<\n\n"))
                else:
                    display.append(('class:highlight', f"{text}\n\n"))
            else:
                display.append(('', f"{text}\n\n"))
        return display
```

display.append(('class:selected', f">>> {text} <<<\n\n")):

This line is key—it adds styling to the currently selected result, making it stand out visually to the user.

Other matched items are displayed using a different style, enabling users to scan results efficiently. Users can navigate between results using the up and down arrow keys, and exit the interface with a simple 'q' keypress.

This function plays a crucial role in user interaction. While the backend handles semantic logic and retrieval, **interactive_highlight_view** is what makes the results accessible, readable, and explorable. It transforms a static list of outputs into an intuitive and responsive exploration experience. This is especially useful for reviewing context-sensitive content like documents or paragraphs, where positioning and emphasis help users grasp relevance. Overall, it significantly enhances usability and provides a polished, interactive layer over the semantic engine.

5.DATA SELECTION AND PREPROCESSING:

1. **Creation of Data:** For the Demonstration purposes, we have used a sample text document from: “The Third Level”- https://ncert.nic.in/textbook/pdf/levt101.pdf (Grade 12 NCERT Vistas), Which contains words of with various grammatical usage (such as various tenses and proverbs/other grammatical structures to enable to test our functionality of the stopword, lemmatization and tokenization modules of our implementation). We saved this file with “.txt” extension.
2. **Applying preprocessing (Tokenization, lemmatization, and Stopword Removal):**
   Using the rule-based lemmatization, regex based tokenization and removal of stopword from the setmap (with the Kaggle stopword dataset). We have preprocessed and cleaned the ThirdLevel.txt document.
   The below images provide insights on before vs after of the preprocessing techniques:

   BEFORE PROCESSING: (sentences)

```
Sentence 1: The presidents of the New York Central and the New York, New Haven and Hartford railroads
will swear on a stack of timetables that there are only two levels at Grand Central Station.

Sentence 2: But I say there are three, because I’ve been on the third level of the Grand Central Station.

Sentence 3: Of course, I took the obvious step — I talked to a psychiatrist friend of mine, among others.

Sentence 4: I told him about the third level, and he said it was a "waking-dream wish fulfillment."
He claimed I was unhappy.

Sentence 5: That made my wife kind of mad, but he explained that what he meant was the modern world is
full of insecurity, fear, war, worry, and all the rest of it — and I just wanted to escape.

Sentence 6: Well, who doesn’t?

Sentence 7: Everybody I know wants to escape, but they don’t wander into a third level at Grand Central.

Sentence 8: That, he said, was the reason behind it.

Sentence 9: My friends agreed.
```

   AFTER PROCESSING: (sentences)

```
Item 1:
president new york central new york new haven hartford railroad will swear stack timetabl on level grand
central station

Item 2:
say ive third level grand central station

Item 3:
course take obviou step talk psychiatrist friend mine

Item 4:
told third level wakingdream wish fulfill claim unhappy

Item 5:
make wife kind mad explain meant modern world full insecure fear war worry rest just want escape

Item 6:
well who doesnt

Item 7:
know want escape dont wand third level grand central

Item 8:
reason

Item 9:
friend agree
```

BEFORE PROCESSING: (paragraphs)

```
Paragraph 1: The presidents of the New York Central and the New York, New Haven and Hartford railroads will swear on a stack of
timetables that there are only two levels at Grand Central Station. But I say there are three, because I've been on the third level
of the Grand Central Station. Of course, I took the obvious step – I talked to a psychiatrist friend of mine, among others. I told him
about the third level, and he said it was a "waking-dream wish fulfillment." He claimed I was unhappy. That made my wife kind of mad, but
he explained that what he meant was the modern world is full of insecurity, fear, war, worry, and all the rest of it – and I just wanted
to escape.

Paragraph 2: Well, who doesn't? Everybody I know wants to escape, but they don't wander into a third level at Grand Central. That, he said,
was the reason behind it. My friends agreed. They pointed to things like my stamp collecting as evidence – calling it a "temporary refuge
from reality." Maybe so. But my grandfather didn't need a refuge from reality; from all I hear, things were nice and peaceful in his time.
He started my collection, in fact. It's a nice collection – blocks of four of practically every U.S. issue, first-day covers, and so on.
President Roosevelt collected stamps too, you know.

Paragraph 3: Anyway, here's what happened at Grand Central. One night last summer, I worked late at the office. I was in a hurry to get uptown
to my apartment, so I decided to take the subway from Grand Central because it's faster than the bus. I don't know why this happened to me –
I'm just an ordinary guy named Charley, thirty-one years old. I was wearing a tan gabardine suit and a straw hat with a fancy band. I passed
a dozen men who looked just like me. I wasn't trying to escape from anything – I just wanted to get home to my wife, Louisa.
```

AFTER PROCESSING: (paragraphs)

```
Item 1:
president new york central new york new haven hartford railroad will swear stack timetabl on level grand central station say ive third level
grand central station course take obviou step talk psychiatrist friend mine told third level wakingdream wish fulfill claim unhappy make wife
kind mad explain meant modern world full insecure fear war worry rest just want escape

Item 2:
well who doesnt know want escape dont wand third level grand central reason friend agree point like stamp collect evidence cal temporary
refuge reale maybe grandfath didnt nee refuge reale hear nice peace time start collection nice collection block practical issue firstday
cover president roosevelt collect stamp know

Item 3:
anyway her happen grand central night last sum work late office hurry uptown apart decid take subway grand central fast bu dont know happen
im just ordinary guy nam charley thirtyone year old wear tan gabardine suit straw hat fancy band pas dozen man who look just like wasnt try
escape just want home wife louisa
```

3.GENERATING REPRESENTATIONS (WORD EMBEDDINGS) TO USE IN THE CUSTOM MADE VECTORDB:

For Sentences:

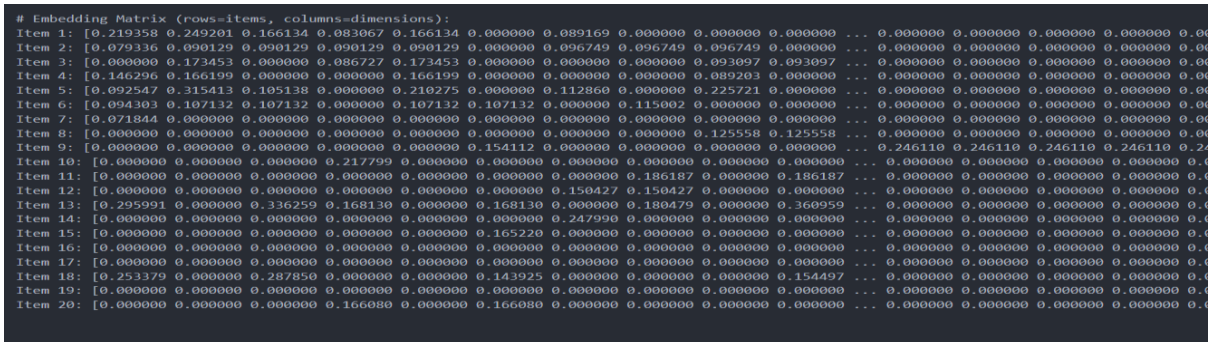
```
# Embedding Matrix: 111 items × 300 dimensions

# Vocabulary Index Mapping:
# 0: level
# 1: central
# 2: grand
# 3: third
# 4: want
# 5: stamp
# 6: old
# 7: look
# 8: found
# 9: station
# 10: friend
# 11: know
# 12: like
# 13: didnt
# 14: year
# 15: down
# 16: go
# 17: that
# 18: ticket
# 19: galesburg
# 20: president
# 21: ive
# 22: psychiatrist
# 23: just
# 24: escape
# 25: world
```

```
# Embedding Matrix (rows=items, columns=dimensions):
Item 1: [0.130117 0.273261 0.140307 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.165918 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 2: [0.313835 0.329545 0.338413 0.348126 0.000000 0.000000 0.000000 0.000000 0.000000 0.400186 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 3: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 4: [0.241735 0.000000 0.000000 0.268148 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 5: [0.000000 0.000000 0.000000 0.000000 0.186565 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 6: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 7: [0.265254 0.278533 0.286027 0.294237 0.313458 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 8: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 9: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000
Item 10: [0.000000 0.000000 0.000000 0.000000 0.000000 0.258211 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 11: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 12: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 13: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 14: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 15: [0.000000 0.000000 0.000000 0.000000 0.000000 0.399394 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 16: [0.000000 0.334707 0.343714 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 17: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 18: [0.000000 0.221500 0.227461 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 19: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.242857 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 20: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 21: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.329159 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 22: [0.000000 0.000000 0.000000 0.000000 0.000000 0.297854 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 23: [0.175192 0.183961 0.188911 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 24: [0.255033 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 25: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
Item 26: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.000
```

(111*130 dimensional word embeddings formed for sentences)

For paragraphs:

```
# Embedding Matrix: 20 items × 300 dimensions

# Vocabulary Index Mapping:
# 0: level
# 1: central
# 2: third
# 3: want
# 4: grand
# 5: found
# 6: friend
# 7: didnt
# 8: like
# 9: look
# 10: that
# 11: galesburg
# 12: station
# 13: ive
# 14: president
# 15: psychiatrist
# 16: escape
# 17: stamp
# 18: know
# 19: year
# 20: old
# 21: louisa
# 22: go
# 23: down
# 24: long
# 25: corridor
# 26: street
```



```
# Embedding Matrix (rows=items, columns=dimensions):
Item 1: [0.219358 0.249201 0.166134 0.083067 0.166134 0.000000 0.089169 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.00
Item 2: [0.079336 0.090129 0.090129 0.090129 0.090129 0.000000 0.096749 0.096749 0.096749 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.00
Item 3: [0.000000 0.173453 0.000000 0.086727 0.173453 0.000000 0.000000 0.000000 0.093097 0.093097 ... 0.000000 0.000000 0.000000 0.000000 0.00
Item 4: [0.146296 0.166199 0.000000 0.000000 0.166199 0.000000 0.000000 0.000000 0.089203 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.00
Item 5: [0.092547 0.315413 0.105138 0.000000 0.210275 0.000000 0.112860 0.000000 0.225721 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.00
Item 6: [0.094303 0.107132 0.107132 0.000000 0.107132 0.107132 0.000000 0.115002 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.00
Item 7: [0.071844 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.00
Item 8: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.125558 0.125558 ... 0.000000 0.000000 0.000000 0.000000 0.00
Item 9: [0.000000 0.000000 0.000000 0.000000 0.000000 0.154112 0.000000 0.000000 0.000000 0.000000 ... 0.246110 0.246110 0.246110 0.246110 0.24
Item 10: [0.000000 0.000000 0.000000 0.217799 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 11: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.186187 0.000000 0.186187 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 12: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.150427 0.150427 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 13: [0.295991 0.000000 0.336259 0.168130 0.000000 0.168130 0.000000 0.180479 0.000000 0.360959 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 14: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.247990 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 15: [0.000000 0.000000 0.000000 0.000000 0.000000 0.165220 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 16: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 17: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 18: [0.253379 0.000000 0.287850 0.000000 0.000000 0.143925 0.000000 0.000000 0.000000 0.154497 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 19: [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.0
Item 20: [0.000000 0.000000 0.000000 0.166080 0.000000 0.166080 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000 0.0
```

(20*300 dimensional word embeddings formed for paragraphs)

4.IMPLEMENTATION OF VECTOR DB (COSINE SIMILARITY TO QUERY AND RETREIVE:

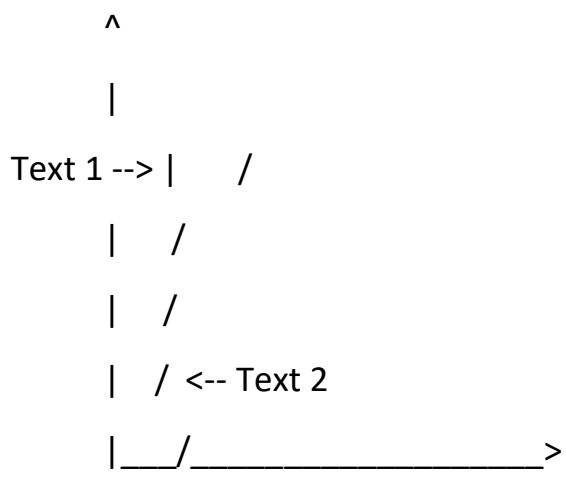
```
H:\CODE\DM>python 5.py
Enter the path to the text file (e.g., sample.txt): sample2.txt
Enter the path to the stopwords file (or press Enter for default 'stopwords.txt'): stopwords.txt
Enter your query: The Entrance to the third Level StairCase
Enter how many top n sentences you want to retrieve (or type 'all'): all
Enter how many top n paragraphs you want to retrieve (or type 'all'): all
Created output directory: preprocessed_output
Loading stopwords from stopwords.txt...
Loaded 733 stopwords.
Processing sentences...
Preprocessed text saved to preprocessed_output/preprocessed_sentences.txt
Processing batch 1/4
Embeddings matrix saved to preprocessed_output/sentence_embeddings.txt
Processing paragraphs...
Preprocessed text saved to preprocessed_output/preprocessed_paragraphs.txt
Processing batch 1/1
Embeddings matrix saved to preprocessed_output/paragraph_embeddings.txt
Running queries...

Top sentence matches:
1. And, Charley, it’s true; I found the third level!...
2. But now, we’re both looking again — every weekend — because we have proof that the third level is re...
3. Keep looking till you find the third level!...

################################################################################

Top paragraph matches:
1. But I’ve never again found the corridor that leads to the third level, though I’ve tried often. Loui...
2. Charley,
I got to wishing that you were right. Then I got to believing you were right. And, Charley,...
3. The presidents of the New York Central and the New York, New Haven and Hartford railroads will sweep
```

(Sample2.txt is the Third Level story text document and stopwords.txt is the Kaggle dataset used for stopword recognization)

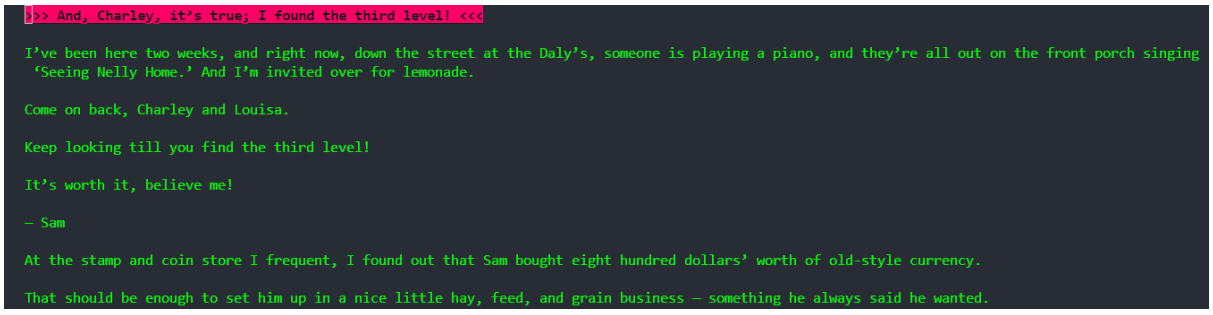
INTUITION FOR THE SIMILARITY SEARCH AND COSINE-SIMILARITY ALGORITHM USED TO RETREIVE FROM VECTOR DB:

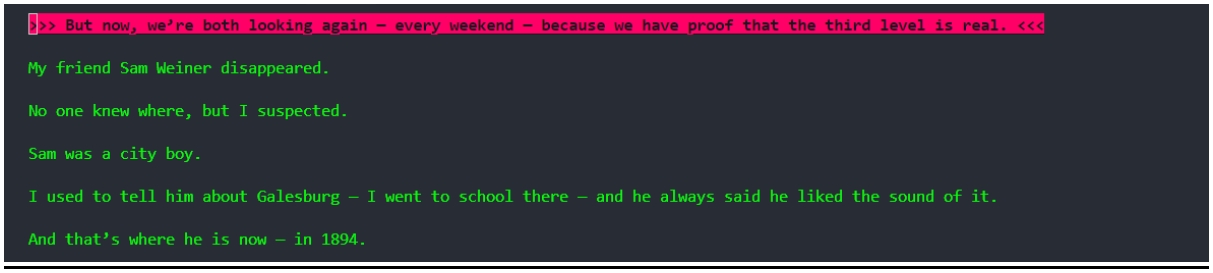
Text-1: The Entry to the Third Level StairCase

Text-2: And, Charley, it's true; I found the third level

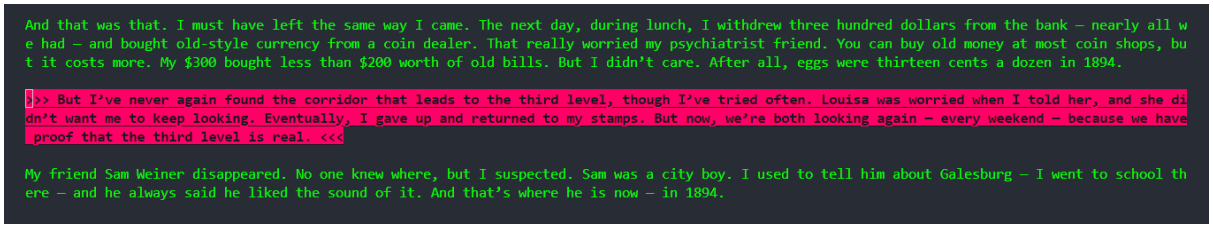


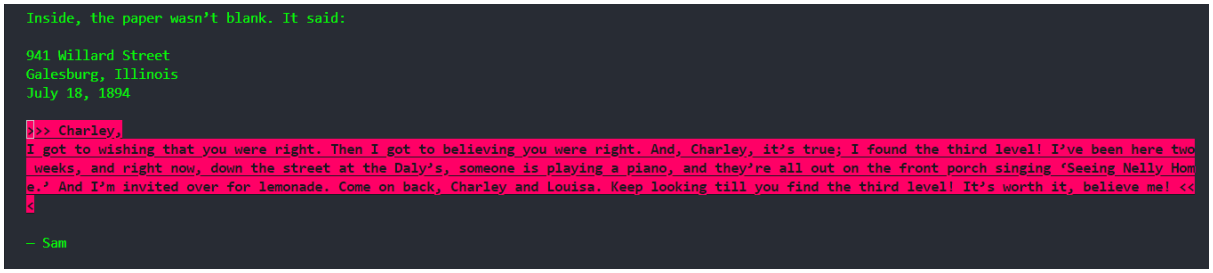
5.VISUALIZATION OF THE OUTPUT/RESULTS AND MAPPING TO THE ORIGINAL TEXT:





(Above are the top 2 closest pairs of sentence match- highlighted in red (the green coloured text are the other potential candidates as we chose "all" top k matches)





(Above are the top 2 closest pairs of paragraph match- highlighted in red (the green coloured text are the other potential candidates as we chose “all” top k matches)

6. Performance Evaluation

Evaluating the performance of a semantic search system is fundamentally different from traditional information retrieval due to the inherently subjective nature of what constitutes a "relevant" result. Unlike keyword-based systems where matches are binary, semantic systems rely on contextual closeness, which is harder to quantify without human annotation. To address this, the following evaluation strategies are employed in this project:

1. Semantic Similarity (Cosine Score):
The core of the system's evaluation rests on semantic similarity, computed using cosine similarity between TF-IDF-based vector representations of queries and documents. Cosine similarity produces a score between -1 and 1, with values closer to 1 indicating stronger contextual alignment. This score is used to rank documents relative to a query, effectively prioritizing results that share similar term importance and distribution. Since the vectors reflect TF-IDF weighting, even simple term-based overlaps can yield useful semantic approximations.

2. Precision@K:
Precision at rank K (Precision@K) measures the proportion of relevant documents within the top-K retrieved results. For example, if 5 out of the top 10 results are relevant, Precision@10 is 0.5. This metric highlights the system's ability to retrieve useful documents early in the ranking. It is particularly important in real-world applications where users typically only examine the top few results. Higher Precision@K values indicate better prioritization of meaningful content.

3. Recall@K:
Recall at rank K (Recall@K) evaluates the system's ability to capture all relevant documents in the top-K results. If there are 10 relevant documents overall and the system retrieves 6 of them within the top 15, then Recall@15 is 0.6. While precision focuses on the quality of retrieved items, recall emphasizes completeness. High recall ensures that fewer relevant results are missed, which is critical in use cases where full information coverage is necessary.

4. Qualitative Analysis:
Due to the lack of a labeled dataset with predefined relevance judgments, the evaluation process also incorporates manual inspection. By examining the top-ranked documents for a variety of queries and comparing them against expected results, users can assess how well the system captures semantic intent. The combination of similarity scores and human review provides valuable feedback on performance, especially in small-scale or domain-specific corpora.

Cosine Similarity Scores: (sentences and paragraphs)

```
Top sentence matches:
1. ID: id102, Similarity: 0.4821
   Text: And, Charley, it’s true; I found the third level!...
2. ID: id2, Similarity: 0.4366
   Text: But I say there are three, because I’ve been on the third level of the Grand Central Station....
3. ID: id105, Similarity: 0.4005
   Text: Keep looking till you find the third level!...
4. ID: id87, Similarity: 0.3645
   Text: But now, we’re both looking again – every weekend – because we have proof that the third level is re...
5. ID: id7, Similarity: 0.3337
   Text: Everybody I know wants to escape, but they don’t wander into a third level at Grand Central....
6. ID: id84, Similarity: 0.3182
   Text: But I’ve never again found the corridor that leads to the third level, though I’ve tried often....
7. ID: id4, Similarity: 0.3094
   Text: I told him about the third level, and he said it was a "waking-dream wish fulfillment." He claimed I...
8. ID: id38, Similarity: 0.2704
   Text: The tunnel turned sharply left; I went down a short flight of stairs and came out on the third level...
```

```
Top paragraph matches:
1. ID: id13, Similarity: 0.3479
   Text: But I’ve never again found the corridor that leads to the third level, though I’ve tried often. Loui...
2. ID: id1, Similarity: 0.2539
   Text: The presidents of the New York Central and the New York, New Haven and Hartford railroads will swear...
3. ID: id18, Similarity: 0.2503
   Text: Charley,
I got to wishing that you were right. Then I got to believing you were right. And, Charley,...
4. ID: id6, Similarity: 0.1344
   Text: The corridor I found myself in began angling left and slanting downward. That felt wrong, but I kept...
5. ID: id5, Similarity: 0.1236
   Text: Sometimes, I think Grand Central is growing like a tree – pushing out new corridors and staircases l...
6. ID: id2, Similarity: 0.1095
   Text: Well, who doesn’t? Everybody I know wants to escape, but they don’t wander into a third level at Gra...
7. ID: id4, Similarity: 0.0900
   Text: I entered Grand Central from Vanderbilt Avenue and went down the steps to the first level, where lon...
8. ID: id7, Similarity: 0.0433
   Text: At first, I thought I was back on the second level, but I quickly realized this was different. The r...
9. ID: id3, Similarity: 0.0000
```

7. Conclusion and Future Work

The **Intelligent Text Search System** demonstrates that semantic search can be both effective and lightweight—achievable without large neural models or cloud infrastructure. Built using classical NLP techniques, TF-IDF-based embeddings, and cosine similarity, it delivers meaningful, context-aware retrieval in a modular, transparent pipeline.

Each module—from rule-based lemmatization and TF-IDF vectorization to in-memory querying and terminal-based result navigation—has been designed for clarity, flexibility, and real-time use. This system proves that:

- **Semantic Search** is achievable using traditional techniques
- **Multi-level Indexing** supports sentence- and paragraph-level retrieval
- **Interactive Exploration** engages users via terminal navigation
- **Transparent Architecture** allows complete visibility into processing
- **Local Execution** ensures accessibility without resource-heavy infrastructure

Ideal for researchers and students, this system is fully understandable and extendable.

Future Work

1. **Improved NLP Capabilities**:
   Add NER, phrase detection, synonym expansion, and language detection to enhance semantic understanding.
2. **Better Embeddings**:
   Explore alternatives like word2vec-lite, LSA, or BM25 for richer vector representations.
3. **Visual Dashboards**:
   Introduce UI elements like similarity maps, concept graphs, and cluster views to aid exploration.
4. **Scalability Enhancements**:
   Implement ANN search (e.g., FAISS), persistent vector storage, and parallel processing to handle larger corpora efficiently.
5. **Extended Format Support**:
   Add parsing for PDF, HTML, Word, and Markdown with semantic structure retention.