TextRank: A Graph-Based Approach to Extractive Summarization

Eleonora Basilico

University of Turin - Stochastics and Data Science - Databases and Algorithms

Contents

1 Introduction

1.1 Problem description

Text Summarization (TS) is a fundamental subtask of Natural Language Processing referring to the automatic creation of a concise and fluent summary that captures the main ideas and topics from a document. TS approaches can be divided into two major categories, namely extractive and abstractive ones. Extractive approaches rank the *top-h* most important sentences in the input document and combine them to form a summary. On the other hand, the abstractive ones generate summaries with potentially new phrases compared to the input document.

In this project we focus on Extractive Text Summarization (ETS). Specificallly, our goal is to extract exactly four sentences from a given document and to combine them to create a brief summary that captures the most significant aspects of the original text.

Extractive Text Summarization is essential for quickly identifying key information from large volumes of text, offering significant benefits in real-world applications like news aggregation, document retrieval and information extraction, where time is critical and users need the essence of a document in a concise form. Given the huge and continuously growing volume of textual data available online, the need for efficient, automatic text summarization has never been more urgent. Every day, millions of articles, research papers, blog posts, and social media updates are produced and manually summarizing this content is neither scalable nor practical.

The primary challenge of extractive summarization is ensuring that the summary is coherent, concise, and adequately representative of the overall content. The selected sentences must cover diverse aspects of the text while avoiding redundancy. However, because these sentences are directly extracted from the text, the resulting summaries may sometimes lack the natural lexical flow typically found in human-produced summaries, making it difficult to ensure smooth transitions between ideas. Nonetheless, certain algorithms handle this challenge better than others.

1.2 State of the art

Various algorithms have been proposed for extractive summarization, each one utilizing different techniques for the sentence ranking and the extraction step, including:

- *Statistical approaches:* Statistical methods rely on word or sentence-level features such as term frequency-inverse document frequency (TF-IDF), sentence length and positional importance to identify key sentences. These approaches typically involve calculating the statistical significance of words or phrases within a document, assuming that higher frequency or contextually significant terms represent important content. Examples include Luhn [Luh58] and Latent Semantic Analysis (LSA) [GL01]. While these approaches are straightforward, they often struggle with capturing deeper semantic relationships, which can lead to summaries that lack coherence or contextual accuracy.

- *Graph-based approaches:* Graph-based methods, such as TextRank [MT04] and LexRank [ER04], model the document as a weighted graph, where sentences represent nodes and edges capture the sentence similarity based on shared terms or semantic overlap. These algorithms use ranking techniques like PageRank [Pag+99] to identify the most central and influential sentences to form the summary.

These approaches are attractive because they do not require pre-labeled training data, making them flexible and applicable across a wide range of domains. However, while they excel in identifying key sentences, the resulting summaries may sometimes lack smooth transitions between ideas, as they treat sentences as independent units.

- *Semantic-based approaches:* Semantic-based approaches aim to go beyond the surface-level word frequency by focusing on understanding the deeper meaning and relationships within the text. Recent advances in these methods leverage pre-trained language models and word embeddings, such as word2vec, GloVe, or more complex architectures like BERT (Bidirectional Encoder Representations from Transformers) [Liu19], to capture the contextual and semantic meaning of words and sentences. These methods outperform purely statistical approaches by understanding relationships at a deeper level, making them more effective in generating summaries that are semantically rich and coherent. However, they require substantial computational resources and may need fine-tuning to perform well in domain-specific contexts.

Each approach has its trade-offs in terms of interpretability, efficiency, and accuracy. Statistical methods are simple and often fast, but they usually miss the deeper relationships between sentences, producing less coherent summaries. Semantic-based techniques offer a superior comprehension of meaning but come with significant computational complexity and require large resources for model training. On the other hand, graph-based methods strike a balance between simplicity and scalability. They work in an unsupervised manner, relying on sentence similarity to form a graph-based structure that effectively captures the text's organization, without requiring labeled data. This makes them highly flexible and efficient, especially for large datasets.

Given these considerations, we decided to solve the extractive summarization problem by using the TextRank algorithm, which is one of the earliest and most prominent graph-based ranking approaches.

2 Modeling

Since we chose a graph-based approach to solve the Extractive Text Summarization problem, we model it using a graph representation. The idea is to represent the document as a graph, where sentences are nodes, and the edges between them capture their relationships or similarities. This graph structure enables us to rank sentences based on their centrality or importance within the document, facilitating the selection of key sentences for the summary. Below, we describe the mathematical framework used to model the problem.

2.1 Mathematical model

Let D be a document consisting of N sentences, denoted as $S_1, ..., S_N$. Our goal is to select the top h sentences, where $h = 4$, to form a concise summary of the document. In order to do so, we model the document as a weighted undirected graph $G = (V, E)$, where:

- $V = \{S_1, ..., S_N\}$ represents the set of nodes, each one corresponding to a sentence in the document;
- E is the set of edges connecting the nodes, where an edge e_{ij} between two nodes S_i and S_j is weighted by the similarity between the sentences. The edge e_{ij} exists if and only if the corresponding weight is non-zero.

We denote the similarity between S_i and S_j by $Sim(S_i, S_j)$ and the weight of the edge e_{ij} by $w_{ij} = Sim(S_i, S_j)$. The similarity score captures how semantically or lexically close the two sentences are. Common measures of sentence similarity in graph-based algorithms include:

- *Cosine similarity:* It measures the cosine of the angle between two sentence vectors in a multi-dimensional space, where each dimension corresponds to a term in the document. Sentences are typically converted into vectors using word frequencies. The formula for cosine similarity between the sentences S_i and S_j is:

  $$Sim_{cos}(S_i, S_j) = \frac{\mathbf{S_i} \cdot \mathbf{S_j}}{\|\mathbf{S_i}\| \|\mathbf{S_j}\|},$$

  where $\mathbf{S_i}$ and $\mathbf{S_j}$ are the vectors corresponding to sentences S_i and S_j, $\|.\|$ indicates the euclidean norm, while $\cdot$ is the inner product between vectors.
  This value ranges from -1 (opposite direction) to 1 (same direction), with higher values indicating greater similarity.

- *Jaccard similarity:* This measure is based on the overlap of words between two sentences. It is defined as the ratio of the intersection of the word sets to their union:

  $$Sim_{Jaccard}(S_i, S_j) = \frac{|W(S_i) \cap W(S_j)|}{|W(S_i) \cup W(S_j)|},$$

  where $W(S_i)$ and $W(S_j)$ represent the sets of the words in sentences S_i and S_j, respectively. The Jaccard similarity ranges from 0 (no overlap) to 1 (complete overlap), indicating how many words two sentences share relative to their combined word sets.

- *Normalized word overlap similarity:* It is defined as the ratio of the number of shared words to the logarithmic sum of the lengths of the two sentences:

  $$Sim_{WO}(S_i, S_j) = \frac{|W(S_i) \cap W(S_j)|}{\log(|W(S_i)|) + \log(|W(S_j)|)}.$$

This normalized measure balances sentence length and overlap, allowing for more effective comparison across sentences of varying lengths.

Notice that the choice of the similarity measure can affect the performance of the summarization, as some measures capture more nuanced relationships than others.

These measures allow us to construct a highly connected weighted graph, where a sentence is linked to another by an edge weighted according to their similarity score, as shown below[1]:

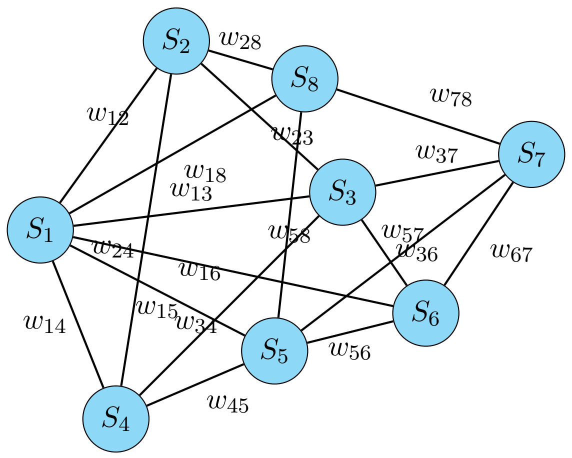


Figure 1: Highly connected and undirected weighted graph with $N = 8$ nodes and weights w_{ij}.

Once the graph is constructed, the next step is to rank sentences based on their importance and centrality within the document. The centrality of a sentence is determined by its connections to other phrases, as reflected in the weighted graph. Sentences that are more connected to other high-scoring sentences are considered more central, making them more likely to contain key information. The ranking algorithm computes an importance score for each sentence based on the graph structure, allowing us to select the top $h = 4$ sentences for the summary. While the specific ranking method may vary depending on the summarization algorithm used, the general principle is that sentences with higher importance scores are chosen to form the final summary.

2.2 Performance measures

The ROUGE metric (Recall-Oriented Understudy for Gisting Evaluation) is a set of measures designed to evaluate the quality of summaries by comparing them to one or more reference texts. There are several variants, each one measuring different aspects of similarity between the generated and the reference summary. To evaluate the algorithm's performance we consider three of the most common ones, ROUGE-1, ROUGE-2 and ROUGE-L, since they provide together a comprehensive evaluation of the quality of the generated summary. All metrics are implemented in the `rouge` package. Below, we explain them in detail.

[1]For a more simple and practical visualization, self-loops representing the similarities w_{ii}, i.e. the similarity of sentences to themselves, are not shown in figure.

2.2.1 ROUGE-N

The ROUGE-N metric evaluates the overlap of N-grams, i.e. sequences of N consecutive words, between the generated and the reference summary. ROUGE-1 and ROUGE-2 are specific cases of ROUGE-N, corresponding to uni-gram (individual word) and bi-gram (pair of consecutive words) matches, respectively.

ROUGE-N evaluates both the extent and quality of overlap through *precision*, *recall*, and *F1-score* measures. More precisely, let S and R be respectively the generated and the reference summary, then:

- *Precision* measures how many of the N-grams in the generated summary S also appear in the reference summary R. It is calculated as:

$$P_N = \frac{\sum_{gram_N \in S} Count_{match}(gram_N)}{\sum_{gram_N \in S} Count_S(gram_N)},$$

  where $Count_S(gram_N)$ is the number of times the $N-$gram $gram_N$ appears in the generated summary, while $Count_{match}(gram_N)$ is the maximum number of times a matching $N-$gram occurs in both the reference and the generated summary.

- *Recall* measures how many of the $N-$grams in the reference summary R are found in the generated summary S. It is defined as:

$$R_N = \frac{\sum_{gram_N \in R} Count_{match}(gram_N)}{\sum_{gram_N \in R} Count_R(gram_N)},$$

  where $Count_R(gram_N)$ is the number of times the $N-$gram $gram_N$ appears in the reference summary.

- *F1-score* combines precision and recall into a single metric, providing a balanced measure that considers both accuracy and completeness of the generated summary. The F1-score is calculated as the harmonic mean of precision and recall:

$$F1_N = 2 \times \frac{P_N \times R_N}{P_N + R_N}.$$

All the above measures fall between 0 and 1, where higher values indicate a greater degree of similarity between the two summaries.
In our evaluation, we will use the F1-score for ROUGE-1 and ROUGE-2 since it balances precision and recall, rewarding summaries that not only capture key information from the reference text (high recall) but also avoid extraneous or irrelevant content (high precision). A high F1-score indicates that the summary is both concise and comprehensive.

2.2.2 ROUGE-L

The ROUGE-L metric measures the longest common subsequence (LCS) between the generated and the reference summary. Unlike ROUGE-N, which is based on exact $N-$gram matches, ROUGE-L captures the order of words and is better suited to evaluate fluency and coherence. This metric reflects the longest shared sequence of words while allowing for possible gaps, making it a flexible measure of similarity that accounts for word order without requiring strict consecutive matches.

As for ROUGE-N, also ROUGE-L calculates *precision*, *recall* and *F1-score* to assess both the extent and quality of the overlap. Again, all the scores range from 0 to 1, with higher values indicating greater similarity between the two summaries.
Precisely, let S and R be respectively the generated and the reference summary and let $LCS(S, R)$ denote the length of their longest common subsequence, then:

- *Precision* measures how much of the generated summary S is made up of the LCS. It is calculated as:
$$P_L = \frac{LCS(S,R)}{|S|},$$
where $|S|$ is the total number of words in the generated summary S.

- *Recall* measures how much of the reference summary R is made up of the LCS:
$$R_L = \frac{LCS(S,R)}{|R|},$$
where $|R|$ is the total number of words in R.

- The *F1-score* is calculated as the harmonic mean of precision and recall:
$$F1_L = 2 \times \frac{P_L \times R_L}{P_L + R_L}.$$

As for ROUGE-N, in our evaluation we use the F1-score for ROUGE-L as it balances precision and recall of the LCS, providing a more comprehensive measure of similarity. This makes ROUGE-L well-suited for evaluating summaries that are both fluent and closely aligned in sequence with the reference summary.

3 Algorithm presentation

In this section, we present the TextRank algorithm and illustrate how it solves the problem of text summarization. We begin with a high-level overview, describing the algorithm's input, output, and main workflow. This is followed by two subsections detailing its two primary stages: the construction of the graph based on sentence similarity, and the application of the PageRank algorithm to identify the most important sentences.

3.1 TextRank algorithm

The TextRank algorithm is one of the earliest and most prominent graph-based ranking approaches. It was designed to identify the most salient sentences in a text for summarization. Its primary goal is to rank sentences by importance, enabling the selection of the top-ranked sentences that capture the key information in the document. Below is the pseudocode illustrating TextRank's main steps:

Algorithm 1 TextRank Algorithm

```
1: procedure TEXTRANK(sentences, damping, thresh_conv, max_iter)
2:     Initialize graph $G$
3:     for each sentence pair $(S_i, S_j)$ do
4:         Compute similarity score $w_{ij} = Sim(S_i, S_j)$ between $S_i$ and $S_j$
5:         if $w_{ij} > 0$ then
6:             Create edge $(S_i \to S_j)$ with weight $w_{ij}$
7:         end if
8:     end for
9:     $scores \leftarrow$ PAGERANK(G, damping, thresh_conv, max_iter)
10:    $ranked\_sentences \leftarrow$ Sort sentences by their scores in descending order
11:    return $ranked\_sentences$
12: end procedure
```

The algorithm can be divided into two main steps:

1. **Graph construction:** It constructs an undirected, weighted graph in which each node represents a sentence from the document. Edges are created between pairs of sentences, with weights representing their degree of similarity. The weight of each edge is determined by a chosen similarity metric that quantifies how closely related the two sentences are in terms of meaning.

2. **PageRank algorithm:** Once the graph is constructed, TextRank applies the PageRank algorithm to assign an importance score to each node. This score indicates how central each sentence is within the context of the entire document. The algorithm then ranks the sentences based on their scores, with the top-ranked sentences representing the most salient points in the document. In our specific case the top $h = 4$ sentences are then selected for summarization.

TextRank takes as input the sentences $S_1, ..., S_N$ of the text and the parameters required by the PageRank algorithm, including the damping factor *damping* (which controls the probability of jumping between nodes), the convergence threshold *thresh_conv* (defining when the algorithm should stop iterating), and the maximum number of iterations *max_iter*. These parameters will be explained in further detail in the subsequent section dedicated to the PageRank method.

3.1.1 Step 1: Graph construction

The first step of the TextRank algorithm involves the construction of an undirected, weighted graph where each node represents a sentence from the document, and each edge between nodes indicates a similarity between two sentences. In this setup, sentences with higher similarity are linked with stronger (higher-weighted) edges, reflecting their closer relationship.

To compute similarity, we use the *normalized word overlap* similarity measure, which was defined in the previous section and is restated here for clarity. This metric, originally presented in [MT04], quantifies the similarity between two sentences based on their shared words, normalized by the length of the sentences. Specifically, given two sentences S_i and S_j, we define the similarity $Sim(S_i, S_j)$ as:

$$Sim(S_i, S_j) = \frac{|W(S_i) \cap W(S_j)|}{\log(|W(S_i)|) + \log(|W(S_j)|)},$$

where $W(S_i)$ and $W(S_j)$ represent the sets of the words in sentences S_i and S_j, respectively. The numerator calculates the count of shared words, and the denominator normalizes this count by the sentence lengths, reducing the similarity for longer sentences with fewer shared words. Notice that the similarity is zero when there are no shared words between the sentences.

While there are various other possible similarity measures, we chose the normalized word overlap metric for this implementation to align with the method proposed in the original TextRank paper [MT04]. This metric is simple yet effective, capturing essential shared content between sentences without requiring additional linguistic processing.

3.1.2 Step 2: PageRank algorithm

The PageRank algorithm was originally introduced by Google founders Sergey Brin and Larry Page to rank web pages by importance [Pag+99]. When applied to the web, PageRank ranks pages by assessing the structure of hyperlinks between them. Web pages with more incoming links are generally ranked higher, as they are considered more significant within the network. This algorithm has then been adapted to rank nodes (or elements) in various types of graphs, making it widely applicable beyond web page ranking.

In the context of TextRank algorithm, PageRank is used to rank sentences based on their similarity with the other sentences in the document. By treating each sentence as a node and representing sentence similarity as edges between nodes, PageRank can assign scores to sentences that reflect their "centrality" or importance within the overall content. These scores help identify the sentences that best summarize the document, capturing the essence of the text.

To describe the PageRank algorithm we use a matrix-based formulation, as it facilitates efficient implementation and computation via linear algebra techniques, particularly through the power method. This formulation is well-suited for iterative computation, making it a popular choice in applications like TextRank. We divide the algorithm's description in three parts:

1. Mathematical formulation and transition matrix P;
2. Damping factor and stationary distribution π;
3. Power method.

1. Mathematical formulation and transition matrix P

Suppose we have a document modeled as an undirected weighted graph G with N nodes, where each node represents a sentence S_i in the document, and edges represent sentence similarities. Let A be the $N \times N$ adjacency matrix, where each entry A_{ij} coincides with the similarity between sentences S_i and S_j. Notice that A is symmetric, since the similarity between sentences is bidirectional.

PageRank's goal is to compute an importance score for each node in the graph. These scores can be represented by a vector $\pi = [\pi_1, ..., \pi_N]^T$ of length N, where each entry π_i denotes the importance score of sentence S_i. This vector represents the stationary distribution of a random walk on G; that is, after sufficient time, the probability of finding the random walk at any node reaches a stable value, captured by π. This stable, or stationary, distribution reflects the long-term likelihood of reaching each node, making π a natural choice for measuring the "importance" of each sentence in the document.

To begin the PageRank calculation, we create from A a transition matrix P. We define P_{ij} as the probability of moving from sentence S_i to sentence S_j in the graph. It is calculated by normalizing each row of A:

$$P_{ij} = \frac{A_{ij}}{\sum_{k=1}^{N} A_{ik}}.$$

Each row of P sums to 1, making it a row-stochastic matrix. This normalization ensures that the total probability of moving out of any node sums to 1, meaning that each entry P_{ij} is a probability value representing a transition from node i to node j.

2. Damping factor and stationary distribution π

The transition matrix P is not the final matrix used to compute the stationary distribution π. To address certain issues, we modify P by introducing a damping factor d which adds a probability of "teleporting" to any other node in the graph at random. Specifically, d represents the probability of following an edge in the graph during each step of a random walk, while the remaining probability $1 - d$ represents the chance that the walker will teleport to a random node instead of following an edge. With the damping factor d, which is often set to 0.85, we define the *adjusted transition matrix* M as follows:

$$M = dP + (1 - d)\frac{\mathbf{1}\mathbf{1}^{\mathbf{T}}}{N},$$

where P is the row-normalized transition matrix, $\mathbf{1}$ is a column vector of ones of length N and $\frac{\mathbf{1}\mathbf{1}^{\mathbf{T}}}{N}$ is a matrix that evenly distributes probability across all nodes, representing the teleportation component. The matrix M thus models a random walker's behavior, where each transition is a combination of following the graph structure and randomly jumping to any node.

The damping factor d was originally introduced in the PageRank algorithm to handle two main issues: *dead-end nodes* (nodes with no outgoing edges) and *spider traps* (subgraphs where nodes link only to each other). Without d, the random walker could get "stuck" in these structures indefinitely, preventing convergence to a stable distribution. Instead, d allows the walker to "teleport" to any node, ensuring that all nodes remain reachable.
In our case, since the graph is undirected, there are no dead-ends or spider traps. However, the damping factor still smooths the distribution of the importance scores, giving even weakly connected sentences a chance of being reached. This adjustment prevents the algorithm from overemphasizing highly connected nodes and allows it to capture relevant but less interconnected sentences.

The final step is to compute the stationary distribution π of M[2]. The vector π satisfies the following equation:

$$\pi = M^T \pi,$$

indicating that π is an eigenvector of the transpose of M with eigenvalue 1. One can prove that 1 is actually the dominant eigenvalue of M^T.

3. Power method

In order to approximate π we use the power method, which is an iterative technique that computes the dominant eigenvector, i.e. the eigenvector corresponding to the dominant eigenvalue of a matrix. Starting with an initial vector $\pi^{(0)}$, which is typically set to a uniform distribution ($\pi_i^{(0)} = \frac{1}{N}$ for each i), we iteratively update π using the following equation:

$$\pi^{(k+1)} = M^T \pi^{(k)}.$$

The iterations continue until the change between successive π values is smaller than a predefined convergence threshold *thresh_conv*, or until we reach a maximum number of iterations *max_iter*.

We report below the pseudocode for the PageRank algorithm, implemented using the power method, as described:

Algorithm 2 PageRank Algorithm using Power Method

```
 1: procedure PAGERANK(G, damping, thresh_conv, max_iter)
 2:     $A \leftarrow$ adjacency matrix of $G$
 3:     Initialize $\pi^{(0)}$ with uniform scores $\frac{1}{N}$ for each of the $N$ sentences
 4:     $P \leftarrow$ transition matrix of $G$ (row-normalized adjacency matrix)
 5:     $M \leftarrow \text{damping} \cdot P + (1 - \text{damping}) \cdot \frac{\mathbf{1}\mathbf{1}^T}{N}$
 6:     $\lambda\_val \leftarrow +\infty$
 7:     $k \leftarrow 0$
 8:     while $\lambda\_val >$ thresh_conv and $k <$ max_iter do
 9:         $\pi^{(k+1)} \leftarrow M^T \pi^{(k)}$
10:         $\lambda\_val \leftarrow \|\pi^{(k+1)} - \pi^{(k)}\|$
11:         $k \leftarrow k + 1$
12:     end while
13:     return $\pi^{(k)}$
14: end procedure
```

The pseudocode follows the power method to calculate the stationary distribution π. Starting with a uniform distribution, it continues iterating, updating π until either the change between iterations falls below *thresh_conv* or the maximum number of iterations *max_iter* is reached. The returned vector π provides the PageRank scores, ranking the nodes (sentences) by their importance. TextRank then selects the top-4 sentences to create the summary.

[2]The existence and uniqueness of π are guaranteed by the irreducibility and positive recurrence of M.

4 Algorithm analysis

In this section, we analyze the TextRank's implementation and performance. We first discuss the preprocessing steps needed to prepare text data before applying the algorithm. Then, we provide a detailed breakdown of the algorithm's implementation in Python, followed by a "toy" example to illustrate its execution. Finally, we analyze the algorithm's computational complexity.

4.1 Text preprocessing

Text preprocessing is essential when working with text data, as it transforms raw text into a clean and structured format, enhancing the effectiveness of algorithms like TextRank. Proper preprocessing helps to eliminate noise, standardize the data, and improve the overall quality of the analysis. The preprocessing steps implemented in this project include:

- **Lowercasing:** All characters are converted to lowercase to ensure uniformity and avoid issues with case sensitivity.
- **Stopwords removal:** Remove common stopwords, such as "and", "the", and "is", that add little meaning to the text. Specifically, for our implementation we used the NLTK library's predefined set of English stopwords. This step reduces noise in the dataset and allows the algorithm to focus on more significant words.
- **Punctuation removal:** Punctuation marks are removed from the text, as they do not carry semantic meaning and can interfere with sentence similarity calculations.
- **Stemming/Lemmatization:** Both techniques aim to reduce words to their base forms, but they work differently. Specifically:
  - *Stemming* reduces words to their root form by stripping affixes (prefixes or suffixes) without considering part of speech (POS). This can result in non-valid words, depending on the stemming algorithm used. For example, the PorterStemmer algorithm, which is the one we use in our implementation, produce the following:

    $$\text{running} \rightarrow \text{run}$$

    $$\text{flies} \rightarrow \text{fli}$$

  - *Lemmatization* reduces words to their dictionary form by considering the context and the part of speech, always resulting in valid words. In the examples above, lemmatization would yield:

    $$\text{running} \rightarrow \text{run}$$

    $$\text{flies} \rightarrow \text{fly}$$

    For our implementation we used WordNetLemmatizer, which accesses WordNet, a large lexical database, to retrieve dictionary forms.

  Choosing between stemming and lemmatization often depends on the specific application. Stemming is typically faster and less resource-intensive, but it can be less precise. In contrast, lemmatization provides more meaningful and accurate results, but is generally slower.

We report below the Python function created to implement these preprocessing steps. The function takes as inputs:

- **sentences**: the list of sentences to preprocess, where each sentence is a string;
- **stemming**: a boolean flag to choose between stemming and lemmatization. If True, the function performs stemming; if False, it applies lemmatization. Default is True.

The function outputs a list of lists of strings, where each inner list contains the preprocessed sentences tokenized into words.

```python
# Initialize stemmer/lemmatizer and stopwords
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))

def get_wordnet_pos(treebank_tag):
    """
    Convert Treebank POS tags to WordNet POS tags.
    """
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN

def preprocess_function(sentences, stemming=True):

    # Regular expression pattern to remove punctuation
    pattern = re.compile(r'[^\w\s]')
    # Initialization of the list that will contain the preprocessed sentences
    preprocessed_sentences = []

    for sent in sentences:
        # Tokenize the sentence into words
        words = nltk.word_tokenize(sent)
        # Remove punctuation, convert to lowercase, and filter out empty strings
        processed_words = [pattern.sub('', word).lower() for word in words if
            pattern.sub('', word).strip()]

        if stemming:
            # Apply stemming while removing stopwords
            preprocessed_sentences.append([
                stemmer.stem(word)
                for word in processed_words
                if word not in stop_words
            ])
        else:
            # Apply lemmatization with POS tagging while removing stopwords
            tagged_words = nltk.pos_tag(processed_words)
            preprocessed_sentences.append([
                lemmatizer.lemmatize(word, get_wordnet_pos(tag))
                for word, tag in tagged_words
                if word not in stop_words
            ])

    return preprocessed_sentences
```

The preprocessing function is structured to ensure flexibility and efficiency. In particular:

- The stopwords list is stored in a set, enabling efficient lookups in constant time (O(1) complexity). This allows the function to efficiently check and exclude non-significant words, enhancing the overall preprocessing speed.

- Punctuation is removed with a compiled regular expression `(re.compile(r'[^\w\s]'))`, which is optimized for repeated use in Python. This approach allows quick punctuation removal without character-by-character checks, making it faster than manual iterations.

- To support accurate lemmatization, the function uses the helper function `get_wordnet_pos` that maps Treebank POS tags to WordNet POS tags. This step is essential for WordNetLemmatizer, which uses WordNet POS tags to produce contextually appropriate word forms.

- The function returns a list of lists, where each inner list contains the preprocessed words of a single sentence. This format is designed for efficient word-level access when computing sentence similarities in the TextRank function, allowing the output to be used directly from the Textrank function without additional transformations.

4.2 TextRank implementation

TextRank is implemented using two functions: the primary function, `textrank`, performs the TextRank algorithm, ranking sentences based on similarity; the helper function, `sentence_similarity`, calculates the normalized word overlap similarity between any two sentences. More precisely:

- `sentence_similarity`: The function takes two inputs, `words1` and `words2`, which are lists of words (tokens) representing the preprocessed sentences. It returns a float which corresponds to the normalized word overlap similarity score between the two sentences.

- `textrank`: The function takes as inputs `sentences_as_words`, a list of lists where each inner list represents a preprocessed sentence as word tokens (coinciding with the output of the preprocessing function[3]); a float, `damping`, representing the damping factor; a float, `threshold`, specifying the convergence threshold for the power method; an optional integer, `max_iter`, specifying the maximum number of iterations for the power method. Default is set to `float('inf')`, which imposes no limit.
  The function outputs two values: a ranked list of tuples, where each tuple contains the index of a sentence and its TextRank score, sorted by score in descending order; the number of iterations it took for the algorithm to converge within the given threshold[4]. To generate the final summary, we retain the top $h = 4$ sentences from the ranked output.

To represent the sentence similarity graph, `textrank` uses an adjacency matrix. This is particularly efficient for relatively highly connected graphs, as in TextRank, where a lot of sentences exhibit some similarity. Additionally, adjacency matrices allow for fast matrix operations (e.g., matrix-vector multiplication), which are crucial for implementing the power method to compute PageRank scores. Overall, this choice allows for efficient convergence and also simplifies the process of normalizing edge weights to construct a stochastic matrix.

[3]Note that we chose not to include the preprocessing step within the textrank function because this function is called multiple times in our simulations, and repeating preprocessing would significantly slow down execution.

[4]We will see in the section dedicated to simulations that we will consider the number of iterations to choose the convergence threshold that provides the best trade-off between speed and accuracy.

The following code blocks provide the implementation of the TextRank algorithm, including the `textrank` and `sentence_similarity` functions:

```python
def sentence_similarity(words1, words2):

    # Create word frequency counters for both sentences
    counter1 = Counter(words1)
    counter2 = Counter(words2)

    # Calculate the number of common words
    common_words = sum((counter1 & counter2).values())

    if common_words == 0:
        return 0.0

    # Calculate the denominator for similarity score
    den = math.log(len(words1)) + math.log(len(words2))

    if np.isclose(den, 0.):
        # This should only happen when words1 and words2 only have a single word.
        # Thus, num can only be 0 or 1.
        return float(common_words)
    else:
        return common_words / den
```

```python
def textrank(sentences_as_words, damping, threshold, max_iter = float('inf')):

    # Compute the number of sentences
    n_sentences = len(sentences_as_words)

    # Initialize the adjacency matrix
    similarity_matrix = np.zeros((n_sentences, n_sentences))

    # Fill the adjacency matrix with similarity scores
    for i, words_i in enumerate(sentences_as_words):
        for j in range(i, n_sentences):
            similarity_ij = sentence_similarity(words_i, sentences_as_words[j])
            # The matrix is symmetric, so similarity_matrix[i, j] is equal to
                similarity_matrix[j, i]
            similarity_matrix[i, j] = similarity_ij
            similarity_matrix[j, i] = similarity_ij

    # Normalize similarity_matrix to make it a stochastic matrix
    row_sums = similarity_matrix.sum(axis = 1) + 1e-8  # Add a constant to prevent
        division by 0
    similarity_matrix = np.divide(similarity_matrix, row_sums[:, np.newaxis])
    # Apply the damping factor to get the adjusted transition matrix
    similarity_matrix = np.full((n_sentences, n_sentences), (1. - damping) /
        n_sentences) + damping * similarity_matrix

    # Apply the power method
    transposed_matrix = similarity_matrix.T
    # Initialize the PageRank vector (all sentences have equal probability
        initially)
    p_vector = np.ones(n_sentences) / n_sentences
    # Initialize the convergence value
    lambda_val = float('inf')

    iter_count = 0
    while lambda_val > threshold and iter_count < max_iter:
        next_p = np.dot(transposed_matrix, p_vector)
        lambda_val = np.linalg.norm(next_p - p_vector)
        # Update the PageRank vector for the next iteration
        p_vector = next_p
        iter_count += 1

    # Rank the sentences based on their PageRank scores
    ranked_sentences = sorted(enumerate(p_vector), key = lambda x: x[1], reverse = True)

    return ranked_sentences, iter_count
```

Notice that the following choices were made to enhance the efficiency and readability of the TextRank implementation:

- In the `sentence_similarity` function, the use of Counter objects allows efficient counting of word frequencies and supports direct calculation of word overlap through the `&` operator, which performs an intersection of word counts between sentences. This approach minimizes the complexity of finding shared words, contributing to faster similarity calculations.
- The `textrank` function leverages numpy arrays for efficient matrix operations. In particular, matrix-vector multiplications in the power method are performed with `np.dot()`, which is optimized for performance with numpy arrays.

4.3 A ”toy” example

To illustrate the application of the TextRank algorithm on a practical case, we use the following sample text. It resembles a brief article (8 sentences) dicussing several aspects of climate change and its impact on agriculture, biodiversity, and ecosystems. This example is small enough for us to break down each step in detail, from preprocessing to summary generation.

Below is the text divided into sentences:

S_1: ”Climate change is causing more intense and frequent droughts in many regions, impacting water availability for various sectors, including agriculture and natural ecosystems.”

S_2: ”Indeed, these environmental changes create major challenges for farmers, who depend on stable water supplies to grow crops, raise livestock, and maintain agricultural productivity.”

S_3: ”Furthermore, natural ecosystems have become increasingly vulnerable to temperature fluctuations and extreme weather, affecting biodiversity.”

S_4: ”Many species, for example, struggle to adapt to the rapid changes and risk extinction as conditions change unpredictably.”

S_5: ”Scientists warn that food production may suffer, as growing seasons become increasingly unpredictable due to climate instability.”

S_6: ”Rising sea levels also pose risks to coastal ecosystems and threaten nearby human communities.”

S_7: ”In addition, agricultural yields are expected to decline if extreme weather events continue to rise in frequency and intensity.”

S_8: ”Therefore, conserving biodiversity and supporting resilient ecosystems is essential for sustaining both agriculture and food security under these rapidly changing conditions.”

We break down the algorithm's execution into the following steps: text preprocessing, similarity calculation and graph contruction, normalization and PageRank computation, summary generation. Moreover, the parameters of the `textrank` function chosen for this example are:

- Damping factor: 0.85. This is a common choice for this parameter, as suggested by the original PageRank algorithm, and it has been widely adopted also in TextRank applications.
- Convergence threshold: 1×10^{-2}. This parameter was chosen to achieve a balance between efficiency and precision in ranking the sentences, as a finer threshold would yield minimal additional insight for this example.
- We do not specify the maximum number of iterations, i.e. there is no limit, since we expect convergence in a few iterations due to the short text.

Step 1: Text preprocessing

As described in the subsection dedicated to text preprocessing, we tokenize the sentences into words by applying lowercasing, stopwords removal, punctuation removal, and lemmatization. In this case, we choose to apply lemmatization because, although it may be slower, the text is short, and prioritizing accuracy is beneficial for this demonstration.

By applying the `preprocess_function` defined earlier, the resulting preprocessed sentences are as follows:

S_1: ['climate', 'change', 'cause', 'intense', 'frequent', 'drought', 'many', 'region', 'impact', 'water', 'availability', 'various', 'sector', 'include', 'agriculture', 'natural', 'ecosystem']

S_2: ['indeed', 'environmental', 'change', 'create', 'major', 'challenge', 'farmer', 'depend', 'stable', 'water', 'supply', 'grow', 'crop', 'raise', 'livestock', 'maintain', 'agricultural', 'productivity']

S_3: ['furthermore', 'natural', 'ecosystem', 'become', 'increasingly', 'vulnerable', 'temperature', 'fluctuation', 'extreme', 'weather', 'affect', 'biodiversity']

S_4: ['many', 'specie', 'example', 'struggle', 'adapt', 'rapid', 'change', 'risk', 'extinction', 'condition', 'change', 'unpredictably']

S_5: ['scientist', 'warn', 'food', 'production', 'may', 'suffer', 'grow', 'season', 'become', 'increasingly', 'unpredictable', 'due', 'climate', 'instability']

S_6: ['rise', 'sea', 'level', 'also', 'pose', 'risk', 'coastal', 'ecosystem', 'threaten', 'nearby', 'human', 'community']

S_7: ['addition', 'agricultural', 'yield', 'expect', 'decline', 'extreme', 'weather', 'event', 'continue', 'rise', 'frequency', 'intensity']

S_8: ['therefore', 'conserve', 'biodiversity', 'support', 'resilient', 'ecosystem', 'essential', 'sustain', 'agriculture', 'food', 'security', 'rapidly', 'change', 'condition']

Step 2: Similarity calculation and graph construction

After preprocessing, we proceed with the core of the TextRank algorithm. The first step involves computing the similarity between each pair of sentences using the `sentence_similarity` function. The computed similarities are then stored in an adjacency matrix A, which serves as the adjacency matrix of the sentence similarity graph, where nodes represent sentences and edge weights represent sentence similarities.

To illustrate similarity calculation, let us compute the similarity between the first two preprocessed sentences. Recall that the `sentence_similarity` function calculates the normalized word overlap similarity, that is:

$$Sim(S_i, S_j) = \frac{|W(S_i) \cap W(S_j)|}{\log(|W(S_i)|) + \log(|W(S_j)|)},$$

where $W(S_i)$ and $W(S_j)$ represent the sets of unique words in sentences S_i and S_j, respectively.

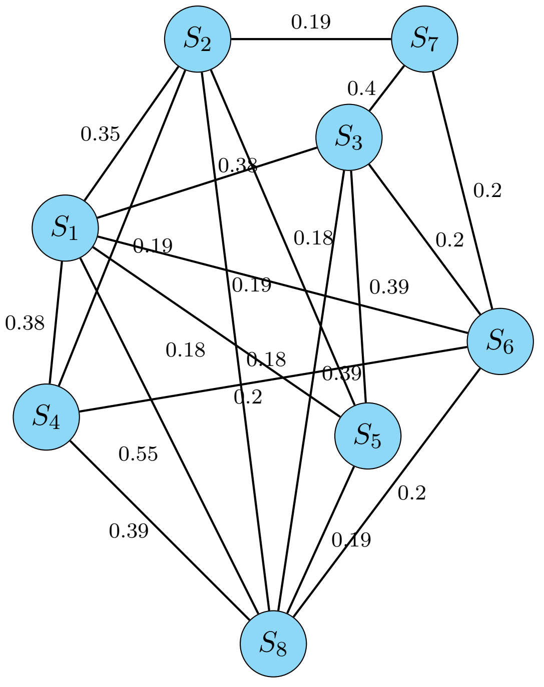
In our case, for the first two preprocessed sentences we have:

- $W(S_1)$ = {'climate', 'change', 'cause', 'intense', 'frequent', 'drought', 'many', 'region', 'impact', 'water', 'availability', 'various', 'sector', 'include', 'agriculture', 'natural', 'ecosystem'}
- $W(S_2)$ = {'indeed', 'environmental', 'change', 'create', 'major', 'challenge', 'farmer', 'depend', 'stable', 'water', 'supply', 'grow', 'crop', 'raise', 'livestock', 'maintain', 'agricultural', 'productivity'}

Hence, $|W(S_1)| = 17$, $|W(S_2)| = 18$ and the shared words are "change" and "water". Therefore:

$$Sim(S_1, S_2) = \frac{2}{\log(17) + \log(18)} \approx 0.35.$$

By repeating these calculations for each pair of sentences, we obtain the following adjacency matrix and corresponding graph representation (we do not show self-loops in the graph for simplicity):



$$A = \begin{bmatrix} 3 & 0.35 & 0.38 & 0.38 & 0.18 & 0.19 & 0 & 0.55 \\ 0.35 & 3.11 & 0 & 0.19 & 0.18 & 0 & 0.19 & 0.18 \\ 0.38 & 0 & 2.41 & 0 & 0.39 & 0.2 & 0.4 & 0.39 \\ 0.38 & 0.19 & 0 & 2.41 & 0 & 0.2 & 0 & 0.39 \\ 0.18 & 0.18 & 0.39 & 0 & 2.65 & 0 & 0 & 0.19 \\ 0.19 & 0 & 0.2 & 0.2 & 0 & 2.41 & 0.2 & 0.2 \\ 0 & 0.19 & 0.4 & 0 & 0 & 0.2 & 2.41 & 0 \\ 0.55 & 0.18 & 0.39 & 0.39 & 0.19 & 0.2 & 0 & 2.65 \end{bmatrix}$$

As we can see, the graph is relatively dense, showing strong similarity connections between many sentences. In particular, sentences S_1, S_3 and S_8 have more connections with higher similarity weights, suggesting they are central in the similarity graph. As a result, we expect these sentences to rank highly and be more likely selected for the summary, as they capture the main themes discussed in the sample text.

Step 3: Normalization and PageRank computation

Before applying the PageRank algorithm, we first need to normalize A by converting it into a transition matrix P, and then apply to P the damping factor to obtain what we called the adjusted transition matrix M.

By dividing each element of A by the sum of the elements in the corresponding row, we normalize A into the following transition matrix P:

$$P = \begin{bmatrix} 0.6 & 0.07 & 0.07 & 0.07 & 0.04 & 0.04 & 0 & 0.11 \\ 0.08 & 0.74 & 0 & 0.04 & 0.04 & 0 & 0.04 & 0.04 \\ 0.09 & 0 & 0.58 & 0 & 0.09 & 0.05 & 0.1 & 0.09 \\ 0.11 & 0.05 & 0 & 0.68 & 0 & 0.06 & 0 & 0.11 \\ 0.05 & 0.05 & 0.11 & 0 & 0.74 & 0 & 0 & 0.05 \\ 0.06 & 0 & 0.06 & 0.06 & 0 & 0.71 & 0.06 & 0.06 \\ 0 & 0.06 & 0.13 & 0 & 0 & 0.06 & 0.75 & 0 \\ 0.12 & 0.04 & 0.09 & 0.09 & 0.04 & 0.04 & 0 & 0.58 \end{bmatrix},$$

where P_{ij} represents the probability of moving from sentence S_i to sentence S_j in the above graph. Let $d = 0.85$ be the damping factor and $N = 8$ be the number of sentences in the sample text. Then, the adjusted transition matrix M is given by

$$M = dP + (1-d)\frac{\mathbf{1}\mathbf{1}^{\mathbf{T}}}{N} = \begin{bmatrix} 0.53 & 0.08 & 0.08 & 0.08 & 0.05 & 0.05 & 0.02 & 0.11 \\ 0.09 & 0.65 & 0.02 & 0.06 & 0.06 & 0.02 & 0.06 & 0.06 \\ 0.1 & 0.02 & 0.51 & 0.02 & 0.1 & 0.06 & 0.1 & 0.1 \\ 0.11 & 0.06 & 0.02 & 0.59 & 0.02 & 0.07 & 0.02 & 0.11 \\ 0.06 & 0.06 & 0.11 & 0.02 & 0.65 & 0.02 & 0.02 & 0.06 \\ 0.07 & 0.02 & 0.07 & 0.07 & 0.02 & 0.62 & 0.07 & 0.07 \\ 0.02 & 0.07 & 0.13 & 0.02 & 0.02 & 0.07 & 0.66 & 0.02 \\ 0.12 & 0.05 & 0.09 & 0.09 & 0.05 & 0.06 & 0.02 & 0.51 \end{bmatrix}.$$

We can now apply the power method to compute PageRank scores. We start with the initial vector $\pi^{(0)} = [\frac{1}{8}, ..., \frac{1}{8}]^T$, which corresponds to a uniform distribution. We then update $\pi^{(0)}$ iteratively using the equation $\pi^{(k+1)} = M^T\pi^{(k)}$, until the euclidean norm between two successive π becomes smaller than the convergence threshold 1×10^{-2}. In our specific case, we have:

- $k = 0 : \pi^{(1)} = M^T\pi^{(0)} = [0.136, 0.126, 0.128, 0.119, 0.12, 0.12, 0.12, 0.13]^T$ and $\|\pi^{(1)} - \pi^{(0)}\| \approx 0.016$, which is grater than the convergence threshold;
- $k = 1 : \pi^{(2)} = M^T\pi^{(1)} = [0.141, 0.127, 0.13, 0.116, 0.118, 0.118, 0.117, 0.133]^T$ and $\|\pi^{(2)} - \pi^{(1)}\| \approx 0.008$, which is lower than the convergence threshold.

Hence, the power method terminates after only two iterations and the PageRank vector is

$$\pi = [0.141, 0.127, 0.13, 0.116, 0.118, 0.118, 0.117, 0.133]^T.$$

Step 4: Summary generation

By ranking sentences in descending order according to their PageRank scores, we identify the sentence order as $S_1, S_8, S_3, S_2, S_6, S_5, S_7, S_4$.
We can now select from this ranked list the top $h = 4$ sentences (S_1, S_8, S_3, S_2) to generate the final

summary. Notice that the summary includes the three sentences (S_1, S_3, S_8) we anticipated would be central, as they showed high similarity with other sentences in the text.

After reordering, the generated summary is the following:

> "Climate change is causing more intense and frequent droughts in many regions, impacting water availability for various sectors, including agriculture and natural ecosystems. Indeed, these environmental changes create major challenges for farmers, who depend on stable water supplies to grow crops, raise livestock, and maintain agricultural productivity. Furthermore, natural ecosystems have become increasingly vulnerable to temperature fluctuations and extreme weather, affecting biodiversity. Therefore, conserving biodiversity and supporting resilient ecosystems is essential for sustaining both agriculture and food security under these rapidly changing conditions."

As we can see, this summary effectively captures the essence of the text, covering key points about the impact of climate change on agriculture, water availability, and ecosystems. It also highlights the importance of biodiversity conservation as a solution to sustain agricultural productivity and food security. Overall, by selecting the sentences with the highest PageRank scores, the algorithm produced a coherent and representative summary that accurately reflects the main ideas of the original text. This confirms both the algorithm's efficacy in extracting relevant content and the robustness of this approach in generating concise, meaningful summaries.

4.4 Computational complexity

In this subsection we analyze the computational complexity of the TextRank algorithm, focusing specifically on the `sentence_similarity` and `textrank` functions. While preprocessing is an important part of the pipeline, it is not considered part of the main algorithm. Therefore, we exclude the computational complexity of the `preprocess_function` from this discussion.

Throughout the whole discussion, the following notation is used:

- N = Number of sentences in the text;
- W = Average number of words per sentence;
- k = Maximum number of iterations for the power method.

The analysis is divided into two parts: time complexity and space complexity, which are evaluated separately below.

4.4.1 Time complexity

We analyze the time complexity of the functions `sentence_similarity` and `textrank` separately. The `sentence_similarity` function requires as input two lists of words, namely `words1` and `words2`. Let W_1 and W_2 be the number of words in `words1` and `words2`, respectively. Then, the noteworthy steps are:

- Constructing the frequency counters `counter1` and `counter2` requires scanning through all the words in `words1` and `words2`. These steps have time complexities of $O(W_1)$ and $O(W_2)$.

- Computing the intersection of two counters, `counter1&counter2`, requires iterating over the smaller counter and checking membership in the larger one. This operation has a time complexity of $O(\min(W_1, W_2))$. Next, `(counter1&counter2).values()` retrieves the counts of the common words, producing a list of size h, where h is the number of common words. The retrieval step contributes an additional $O(h)$ to the complexity, and summing the counts in this list also takes $O(h)$. Since $h \leq \min(W_1, W_2)$, the overall time complexity of this step, including intersection, retrieval, and summation, is $O(\min(W_1, W_2))$.
- Calculating the denominator of the similarity score involves logarithmic and arithmetic computations, which require $O(1)$ time.

Thus, the overall time complexity of the `sentence_similarity` function is $O(W_1+W_2+\min(W_1, W_2))$. Notice that if we consider the case $W_1 = W_2 \approx W$, it simplifies to $O(W)$.

The `textrank` function requires as input a list of the N sentences in the text, each represented as a list of words, along with the damping factor *damping*, the convergence threshold and the maximum number of iterations k for the power method. The time complexity of the function can be analyzed step by step as follows:

- Initializing the adjacency matrix `similarity_matrix` with zeros requires $O(N^2)$ operations.
- To populate the adjacency matrix with similarity scores, a nested loop is implemented. The outer loop iterates over the N sentences, and for each iteration i, the inner loop processes $N - i$ sentences. This ensures that only the upper triangular part of this symmetric matrix (including the diagonal) is computed, amounting to $N(N+1)/2$ similarity calculations. Each similarity calculation involves a call to the `sentence_similarity` function. Assuming each sentence has an average of W words, the complexity of a single call is $O(W)$. Therefore, the time complexity of this step is:
$$O(\tfrac{N(N+1)}{2} \cdot W) = O(N^2 W).$$
- Normalizing the similarity matrix involves two substeps:
  - Calculating the sum of each row, which requires $O(N^2)$, as there are N rows and each row has N elements.
  - Dividing each element by its row sum, which also requires $O(N^2)$.

  Thus, the overall complexity of this step is $O(N^2)$.
- Creating the adjusted transition matrix involves initializing a matrix with a uniform value of $(1 - \textit{damping})/N$, requiring $O(N^2)$, and then modifying it using the normalized similarity matrix, also $O(N^2)$. Consequently, the time complexity of this step is $O(N^2)$.
- Transposing the transition matrix involves swapping its elements, which takes $O(N^2)$. Next, the PageRank vector `p_vector` is initialized as a uniform distribution, requiring $O(N)$ operations, which have a negligible impact on the overall complexity of the algorithm.
- For each iteration of the power method, the PageRank vector is updated by matrix-vector multiplication, which takes $O(N^2)$. Moreover, the value `lambda_val` (used to check convergence) is updated by computing the norm of the difference between two successive PageRank vectors, requiring $O(N)$ operations. The number of iterations depends on the convergence threshold and can be at most k. Hence, the complexity of this step in the worst-case scenario is $O(kN^2 + kN) = O(kN^2)$.

- Sorting the N elements of the PageRank vector takes $O(N\log(N))$ operations.

Hence, the overall time complexity of the `textrank` function is $O(N^2+N^2W+N^2+kN^2+N\log(N))$. Since N^2W and kN^2 dominate the expression, the final time complexity can be simplified to $O((W+k)N^2)$.
This time complexity reflects both strengths and limitations of the `textrank` function. For small to medium-sized texts, where N and W are modest, the algorithm performs well, as the power method often converges in relatively few iterations. However, the quadratic dependency on N becomes a bottleneck when dealing with large documents containing hundreds or thousands of sentences. Similarly, texts with very long sentences introduce overhead in the similarity calculations, further increasing the runtime. Preprocessing can reduce W, mitigating this effect.
In practice, the algorithm is well-suited for tasks such as summarizing articles or ranking sentences in moderately sized texts. For larger texts, optimizations such as sparse matrix representations or graph pruning may be necessary to handle the quadratic growth efficiently.

4.4.2 Space complexity

We now discuss the space complexity of the two algorithms. This includes the memory space used by their inputs, called *input space*, and any other memory they use during execution (including permanent and temporary memory), which is called *auxiliary space.*

As for the `sentence_similarity` function, let W_1 and W_2 denote the number of words in `words1` and `words2`, and U_1 and U_2 be the number of unique words. Then:

- **Input space:** Since the input consists of two lists, `words1` and `words2`, whose sizes are W_1 and W_2, it contributes to the overall space complexity $O(W_1+W_2)$.
- **Auxiliary space:** Key steps include
  - Constructing the frequency counters `counter1` and `counter2` requires space proportional to the number of unique words in each input list, U_1 and U_2. In the worst case, where all words are unique, we have $U_1 = W_1$ and $U_2 = W_2$. Hence, the combined space complexity of the counters is $O(W_1+W_2)$.
  - The intersection `counter1&counter2` creates a temporary new counter containing the common words and their minimum frequencies. This counter occupies $O(h)$, where h is the number of common words. In the worst case, in which all words are common, $h = \min(W_1, W_2)$. Moreover, the temporary list created by `counter1&counter2.values()` stores these counts, requiring the same $O(\min(W_1, W_2))$ space. Thus, the total space complexity of this step is $O(\min(W_1, W_2))$.
  - Calculating the denominator of the similarity score involves arithmetic and logarithmic operations, which use $O(1)$ space. The final output, which is a single similarity score, also requires $O(1)$ space.

Considering both input and auxiliary space, we obtain an overall space complexity of $O(W_1+W_2+\min(W_1, W_2))$. Again, in the specific case where $W_1 = W_2 \approx W$, it simplifies to $O(W)$.

Let us now analyze the space complexity of the `textrank` function step by step:

- **Input space:** The function takes as inputs a list of lists of words, two floats and an integer. Assuming the text contains N sentences, each with an average of W words, the input requires $O(NW)$ space.

- **Auxiliary space:** Key steps include
  - The adjacency matrix `similarity_matrix` is initialized as a dense $N \times N$ matrix filled with zeros. This requires $O(N^2)$ space.
  - Computing the similarity scores involves $O(N^2)$ calls to the `sentence_similarity` function, each requiring $O(W)$ temporary memory. Since this memory is released after each call, the cumulative temporary space at any given moment is $O(W)$. The results of these computations are stored in the already-allocated adjacency matrix, so no additional permanent memory is required.
  - Normalizing the similarity matrix involves computing and storing the sum of each row of the matrix, which requires $O(N)$ space, and then dividing each element by its row sum. This row-wise division is performed in-place, so no additional space is needed.
  - To create the adjusted transition matrix a temporary $N \times N$ matrix with a uniform value of $(1 - \textit{damping})/N$ is initialized, requiring $O(N^2)$ space. Similarly, the scaled similarity matrix *damping* `* similarity_matrix` requires another temporary $O(N^2)$ space. These two matrices are added element-wise to compute the final transition matrix, generating another temporary matrix to hold the result. However, the final result overwrites the original similarity matrix, so the peak space usage for this step remains $O(N^2)$.
  - Storing the transpose of the transition matrix requires $O(N^2)$ permanent space, while the initialization of the PageRank vector `p_vector` $O(N)$.
  - At each iteration of the power method, the PageRank vector is updated and stored in `next_p`, requiring $O(N)$ space. Additionally, computing the difference between successive PageRank vectors for convergence involves a temporary storage of $O(N)$.
  - Sorting and storing the final PageRank vector requires $O(N)$ space.

Thus, we obtain an overall space complexity of $O(NW + N^2)$. Notice that for texts where $N \gg W$, the $O(N^2)$ term dominates, making this a limitation for documents with many sentences. However, for shorter texts with long sentences, also the $O(NW)$ term may become significant.

5 Simulations

In this section, we apply the TextRank algorithm and evaluate its performance on a real-world dataset. The section is organized as follows: we first provide a brief description of the dataset used for simulations. Next, we outline the preprocessing steps and describe the process of selecting key algorithm parameters, such as the convergence threshold and the maximum number of iterations for the power method. Finally, we apply the algorithm to the full dataset and evaluate its performance using ROUGE metrics.

5.1 Dataset description

For our simulations, we used the CNN/DailyMail dataset, an English-language dataset containing over $300,000$ unique news articles written by journalists from CNN and the Daily Mail. Each instance in the dataset consists of the following three features:

- `id`: A string containing the hex-formatted hash of the URL where the story was retrieved.
- `article`: A string containing the body of the news article.
- `highlights`: A string containing a summary of the article as written by the author.

The dataset is provided in three splits: train, validation, and test, reflecting its widespread use in developing and testing text summarization models, particularly neural networks. However, for our purposes, which do not require separate validation or testing phases, we combined all three splits into a single dataset.

To align with the objective of this project, we restricted our analysis to the subset of articles whose *highlights* contain exactly four sentences. The resulting subset includes $105,340$ articles.

We selected the CNN/DailyMail dataset for several reasons. First, it is a well-established benchmark in the field of text summarization, making it a natural choice for evaluating summarization algorithms. Second, its large size provides a substantial amount of data for robust analysis. Third, the articles in this dataset typically do not have an excessively large number of sentences, making it particularly suitable for algorithms like TextRank, which can become computationally expensive on extremely long texts. Finally, it is one of the few open-source datasets that includes human-written summaries, enabling reliable performance evaluation of summarization techniques.

5.2 Data cleaning

To ensure the dataset's quality and consistency, we applied several preprocessing steps to clean the data. This process focused on removing duplicates, eliminating unnecessary metainformation from the articles, and resolving formatting issues that could interfere with the evaluation of the summaries' quality. Below are the main cleaning steps performed.

1. Removing duplicates

The dataset contained duplicate entries with identical articles and highlights but different IDs. These duplicates were removed to prevent redundancy and ensure each article was processed once.

2. Cleaning highlights

For each instance of the dataset the sentences in the *highlights* column were separated by '\n'.

These newline characters were removed to avoid potential issues with the `rouge` evaluation package. Moreover, some highlights began with the word “NEW:” (e.g., ”NEW: State media report...”). This prefix was removed as it is not part of the summary’s semantic content.

3. Removing metainformation in articles

The articles often contained extraneous metainformation, particularly at the beginning, which was irrelevant to the summarization task. Below are the types of metainformation identified and some examples of their original and cleaned forms:

- Articles starting with (CNN), (CNN) −−, or other journal or websites names in parentheses (e.g., (Rolling Stone), (EW.com)), eventually preceded by city and/or country names.

  Examples:

  - (CNN)That’s some rich ”American Pie.”... $\longrightarrow$ That’s some rich ”American Pie.”...
  - (Rolling Stone) −− The digital archivist who... $\longrightarrow$ The digital archivist who...
  - Canberra, Australia (CNN) −− At first glance... $\longrightarrow$ At first glance...

- Articles beginning with CNN −− or names in formats ’City, Country’ followed by −−.

  Examples:

  - CNN −− As one of the most successful... $\longrightarrow$ As one of the most successful...
  - LONDON, England −− Models younger than 16... $\longrightarrow$ Models younger than 16...

- Articles starting with ’By . [Author Name]’, eventually followed by other metadata beginning with ’PUBLISHED: .’, ’UPDATED: .’, ’Follow @@’, ’CREATED: .’, ’Last updated’, or other author names, regardless of case.

  Examples:

  - By . Oliver Todd . Follow @@oliver_todd . The Bundesliga is... $\longrightarrow$ The Bundesliga is...
  - By . Rob Waugh . UPDATED: . 04:54 EST, 28 September 2011 . Apple has made the launch... $\longrightarrow$ Apple has made the launch...
  - By . Chris Parsons . CREATED: . 00:04 EST, 26 June 2012 . A ’wretched and wicked’ primary school... $\longrightarrow$ A ’wretched and wicked’ primary school...

  Some articles started directly with ’CREATED: .’, ’UPDATED: .’, ’PUBLISHED: .’ or ’Last updated’, regardless of case. These sentences were removed.

- Articles beginning with a note from the editor introduced by the expression ’Editor’s note:’. All the notes are made of a single sentence, which was removed.

- Expressions such as (left), (right) or (pictured) indicating images were removed.

- Articles starting with ’Click here’, regardless of case.

  Examples:

  - CLICK HERE to read how the German plans to pip Lewis Hamilton to the post in Abu Dhabi . Nico Rosberg believes he can force... $\longrightarrow$ Nico Rosberg believes he can force...

- Articles beginning with dates in formats ’Month day, year’.

  Examples:

  - April 16, 2014 . A city pays tribute to the... $\longrightarrow$ A city pays tribute to the...
  - December 10, 2013 . The official start of winter... $\longrightarrow$ The official start of winter...

This metainformation was removed using regular expressions. We provide the Python code below.

```python
# List of the journal/websites names appearing in parentheses, regardless of case and eventually followed by .com
names = ['CNN', 'CNN Student News', 'CNN Espanol', 'Rolling Stone', 'RollingStone', 'CNET', 'EW', 'Mental Floss', 'MentalFloss', 'AOL Autos', 'Time', 'Careerbuilder', 'Oprah', 'People', 'Mashable', 'Entertainment Weekly', 'Financial Times', 'The Frisky', 'LifeWire', 'Wired', 'Ars Technica', 'InStyle', 'Real Simple', 'RealSimple', 'Health', 'Parenting', 'Reuters', 'Motor Sport', 'VBS.TV', 'Save The Children', 'Business Insider', 'Upwave', 'Uwire', 'MNN', 'Budget Travel', 'Travel \+ Leisure', 'HLNtv', 'CNNGo', 'Tribune Media Services', 'Southern Living', 'ESSENCE', 'This Old House', 'RS']

for name in names:
    # For each name replace the pattern with empty string in the 'article' column to remove the first type of metainformation
    pattern = r'^.*?\(\s*' + name + r'(?:\.com)?\s*\)(?:\s*--\s*)?'
    df['article'] = df['article'].str.replace(pattern, '', regex = True, flags = re.IGNORECASE)

# Define a list of regular expression patterns to remove all other metainformation except for the last type
patterns = [
    r'^CNN\s*--\s*',
    r'^\w+\s*[.,]+\s*\w+\s*--\s*',
    r'^.*?CREATED: \..*?\.\s*',
    r'^.*?UPDATED: \..*?\.\s*',
    r'^.*?PUBLISHED: \..*?\.\s*',
    r'^.*?Last updated at .*?\.\s*',
    r'^(By\s*\.\s*.*?\s*\.\s*)(?:((Follow|and)\b.*?\.\s*))?(?:(and\b.*?\.\s*))?',
    r'^Editor"s note:.*?\.\s*',
    r'\((left|right|pictured)\)\s*',
    r'^Click here.*?\.\s*'
]
for pattern in patterns:
    # Replace occurrences of the pattern in the 'article' column with an empty string
    df['article'] = df['article'].str.replace(pattern, '', regex = True, flags = re.IGNORECASE)

# List of months
months = ['Jenuary', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']
for month in months:
    # For each month replace the pattern with empty string in the 'article' column to remove the last type of metainformation
    pattern = r'^' + month + r' \d{1,2}\s*, \d{4}\s*\.\s*'
    df['article'] = df['article'].str.replace(pattern, '', regex = True)
```

Overall, these cleaning steps ensured that the dataset was well-prepared for applying the TextRank algorithm, removing extraneous content that could negatively affect performance. Moreover, these patterns were only found at the beginning of the articles, and no other significant recurring pattern was identified in the main body of the articles.

5.3 Sample generation

To optimize the TextRank algorithm efficiently, we used a representative sample of the dataset. Specifically, we needed to determine the optimal convergence threshold and the maximum number of iterations for the power method, which are critical for the algorithm's efficiency and accuracy. Additionally, we had to decide between stemming and lemmatization, as these text preprocessing choices can significantly affect the results. A representative sample was essential because applying the algorithm repeatedly to the entire dataset would be computationally too expensive due to its large size. Hence, by conducting these experiments on a smaller yet representative subset, we ensured computational feasibility while maintaining reliable and generalizable results.

To create the sample, we used *stratified sampling* based on the number of sentences in each article. This method ensures that the sample's proportions of articles with specific sentence counts match those in the full dataset. We chose a stratification fraction of 0.3, meaning that the 30% of the articles from each sentence-count stratum were included in the sample. The resulting sample contains $31,108$ articles.

To confirm that the sample is representative of the entire dataset, we compared the distributions of the following key features between the full dataset and the sample:

1. Number of sentences per article;
2. Number of words per article;
3. Average number of words per sentence.

The following histograms demonstrate that the sample's distributions align closely with those of the full dataset:

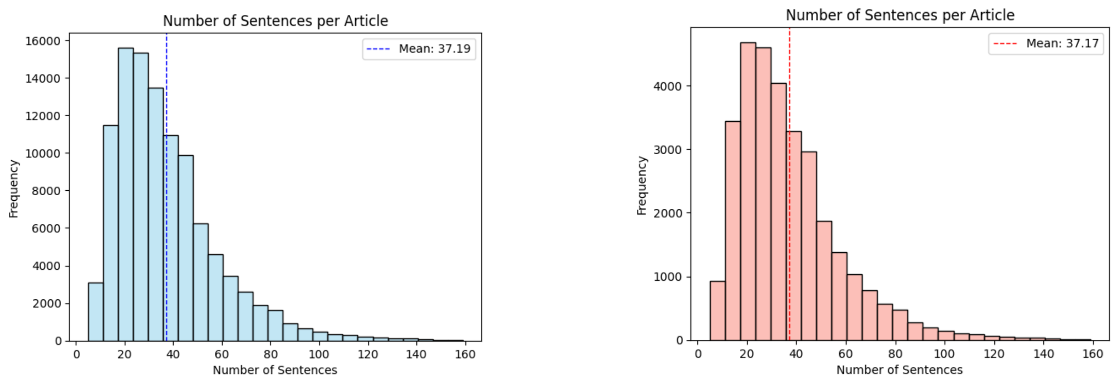


Figure 2: Histograms showing the number of sentences per article in the full dataset (left) and in the sample (right).

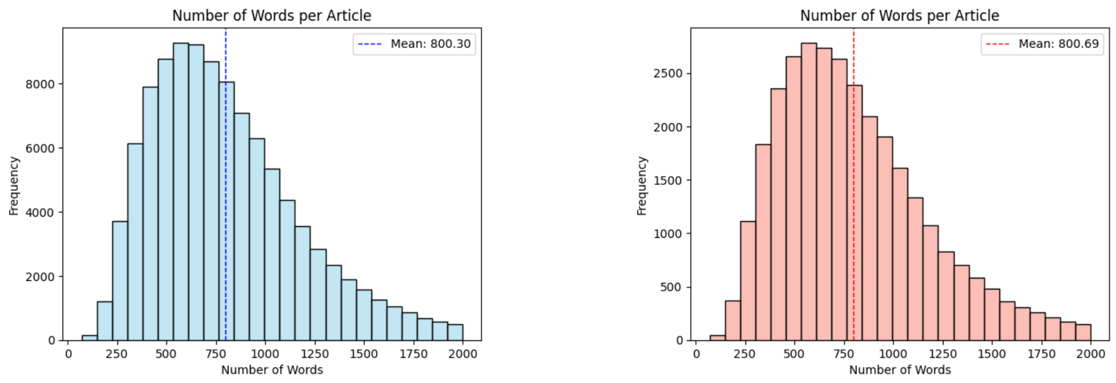
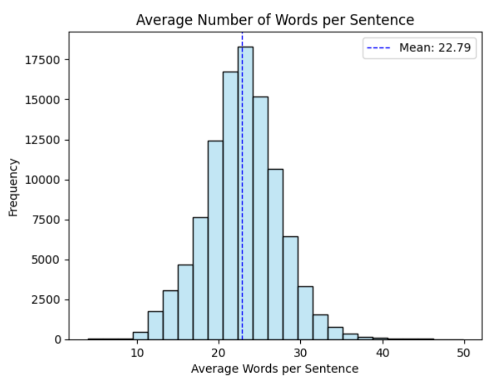


Figure 3: Histograms showing the number of words per article in the full dataset (left) and in the sample (right).



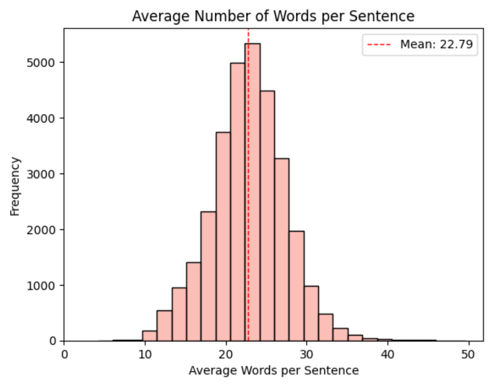


Figure 4: Histograms showing the average number of words per sentence in the full dataset (left) and in the sample (right).

Overall, the consistency between the sample and the full dataset ensures that the conclusions drawn from the sample are applicable to the entire dataset.

5.4 Parameter selection

We now describe the process adopted for tuning the parameters of the TextRank algorithm to ensure optimal performance. Specifically, we optimized two key parameters: the convergence threshold and the maximum number of iterations for the power method. Additionally, we compared stemming and lemmatization as text preprocessing techniques to determine which provided a better trade-off between computational efficiency and output quality.

The damping factor for the power method was fixed at 0.85, which is a common choice for this parameter, as suggested by the original PageRank algorithm. To tune these parameters and compare stemming versus lemmatization, we repeatedly applied the TextRank algorithm to the representative sample described in the previous subsection. The process was divided into two steps: first, we optimized the convergence threshold and selected between stemming and lemmatization; then, we fine-tuned the maximum number of iterations. Below, we present a detailed discussion of each step.

5.4.1 Stemming vs. Lemmatization and convergence threshold

To optimize the convergence threshold and select between stemming and lemmatization, we conducted the following procedure:

- The TextRank algorithm was run on the sample dataset using four convergence thresholds: $10^{-1}, 10^{-2}, 10^{-3}$ and 10^{-4}. These thresholds were chosen to represent a logical progression in precision, balancing computational efficiency and output quality.
- To ensure the algorithm's convergence for all thresholds, the maximum number of iterations was set to $10,000$, as if there was no limit.
- For each threshold we collected the following metrics:
  - Mean execution time per article (it does not include preprocessing time)[5];

[5] The reported execution times were measured on an Intel i5 processor with 8GB RAM. Times may vary depending on the hardware used.

  – Mean ROUGE-1, ROUGE-2, and ROUGE-L scores, which measure the overlap between the generated and the reference summaries;
  – Statistics on the number of iterations required for convergence (mean, median, maximum, 90th, 95th, and 99th percentiles).

- The process was repeated twice, first with stemming and then with lemmatization applied during text preprocessing.

The following two tables summarize the metrics obtained using stemming and lemmatization:

Convergence Threshold	**Mean Execution Time (s)**	**ROUGE-1 Mean**	**ROUGE-2 Mean**	**ROUGE-L Mean**
10^{-1}	0.0151	0.2940	0.0986	0.2708
10^{-2}	0.0151	0.2943	0.0991	0.2713
10^{-3}	0.0152	0.2952	0.0999	0.2722
10^{-4}	0.0159	0.2954	0.1001	0.2724

Mean n_iter	**Median n_iter**	**Max n_iter**	**90th perc. n_iter**	**95th perc. n_iter**	**99th perc. n_iter**
1.0001	1.0	2.0	1.0	1.0	1.0
2.4327	2.0	5.0	3.0	3.0	3.0
7.6881	7.0	109.0	8.0	8.0	50.0
75.0784	13.0	544.0	375.0	490.0	521.0

Table 1: Metrics obtained for each convergence threshold using stemming.

Convergence Threshold	**Mean Execution Time (s)**	**ROUGE-1 Mean**	**ROUGE-2 Mean**	**ROUGE-L Mean**
10^{-1}	0.0148	0.2932	0.0981	0.2700
10^{-2}	0.0147	0.2934	0.0985	0.2704
10^{-3}	0.0148	0.2941	0.0992	0.2711
10^{-4}	0.0155	0.2941	0.0993	0.2712

Mean n_iter	**Median n_iter**	**Max n_iter**	**90th perc. n_iter**	**95th perc. n_iter**	**99th perc. n_iter**
1.0001	1.0	2.0	1.0	1.00	1.0
2.4205	2.0	5.0	3.0	3.00	3.0
7.6796	7.0	109.0	8.0	8.00	50.0
75.0567	13.0	544.0	375.0	490.65	521.0

Table 2: Metrics obtained for each convergence threshold using lemmatization.

As we can see from the above tables:

- Both stemming and lemmatization yielded comparable ROUGE-1, ROUGE-2, and ROUGE-L scores, with stemming having a slight edge at all thresholds. Moreover, for both techniques the ROUGE scores increase slightly with a lower convergence threshold.

- In terms of iterations and convergence speed, both methods show similar trends. As expected, the execution time and the number of iterations grow as the convergence threshold decreases.

Although the performance metrics for stemming and lemmatization are almost identical, we selected stemming as the preprocessing method due to its superior efficiency: the average execution time for text preprocessing on the sample dataset was significantly lower for stemming (0.0125s) compared to lemmatization (0.0610s). The slight improvement in ROUGE scores with stemming further supports this decision.

As for the convergence threshold, we chose 10^{-3} as the optimal value. Indeed, it provides a good trade-off between computational efficiency and performance, with a slight increase in ROUGE scores compared to 10^{-2} and 10^{-1}. Although 10^{-4} gave marginally higher ROUGE scores, it significantly increased the number of iterations, making it impractical for large datasets. Moreover, 10^{-3} offers flexibility for more complex articles compared to higher thresholds like 10^{-1} and 10^{-2}, which often converge prematurely (from 1 to 3 iterations).

5.4.2 Maximum number of iterations

After selecting stemming and a convergence threshold of 10^{-3}, we tuned the maximum number of iterations for the power method. We tested three values: 7, 8, and 50, corresponding to the mean, the 95th percentile (= 90th percentile) and the 99th percentile of the number of iterations observed in the previous experiments.

For each value we collected the mean execution time per article and the mean ROUGE-1, ROUGE-2 and ROUGE-L scores. We report the results in the table below.

Max n_iter	**Mean Execution Time (s)**	**ROUGE-1 Mean**	**ROUGE-2 Mean**	**ROUGE-L Mean**
7.0	0.0148	0.2951	0.0999	0.2722
8.0	0.0147	0.2952	0.0999	0.2722
50.0	0.0148	0.2952	0.0999	0.2722

Table 3: Metrics obtained for each maximum number of iterations using stemming and 10^{-3} as convergence threshold.

As we can see, all three configurations yielded nearly identical ROUGE scores. However, 8 iterations provided, although slightly, the fastest execution time. Choosing 8 strikes a balance between being too conservative (like 7) and unnecessarily extending the runtime (like 50), ensuring quick convergence without sacrificing the summaries' quality.

5.5 Algorithm application and performance evaluation

To evaluate the performance of the TextRank algorithm, we applied it to the entire dataset using the optimized parameters determined in the tuning phase. Specifically, we set the convergence threshold to 10^{-3}, the maximum number of iterations to 8, the damping factor to 0.85 and we employed stemming during text preprocessing. We generated a summary for each article and then collected the following metrics:

- Mean execution time per article;
- Mean ROUGE-1, ROUGE-2, ROUGE-L scores.

We report the results obtained from this evaluation in the table below:

Mean Execution Time (s)	ROUGE-1 Mean	ROUGE-2 Mean	ROUGE-L Mean
0.0153	0.2952	0.0998	0.2721

Table 4: Metrics obtained by applying TextRank on the whole dataset using stemming, 10^{-3} as convergence threshold, 0.85 as damping factor and 8 as maximum number of iterations.

As discussed in Section 2, ROUGE metrics provide a quantitative measure of the quality of text summarization, but the raw scores alone are not sufficient to properly judge the performance. In text summarization, especially with extractive algorithms like TextRank, very high ROUGE scores are rarely achievable since this would require the generated summary to be nearly identical to the reference summary. Therfore, to assess the quality of our results, rather than evaluating how close the ROUGE scores are to 1, we compared them to two baselines:

- **Random baseline** (Lower bound): For each article, this baseline generates a summary by randomly selecting four sentences from the article. The mean ROUGE scores are then computed across all articles. This represents a lower bound for comparison, indicating the expected performance of an entirely random approach.
- **Greedy oracle baseline** (Upper bound): For each article, the oracle baseline evaluates all possible combinations of four sentences to find the combination that maximizes the ROUGE scores compared to the reference summary. Given the large amount of data we have (over 100,000 articles), this approach is infeasible, so we used a greedy approximation. Specifically:
  - For each sentence in the article, the mean of the three ROUGE scores between the sentence and the reference summary is computed. Then, the four sentences providing the highest scores are selected to form the summary.
  - For practical reasons, we limited the candidate sentences to the first half of each article. This is justified by considering that, in general, important information is presented at the beginning of an article.

We summarize the performance of TextRank and the two baselines in the table below:

Method	ROUGE-1 Mean	ROUGE-2 Mean	ROUGE-L Mean
Random Baseline	0.2374	0.0626	0.2184
TextRank	0.2952	0.0998	0.2721
Greedy Oracle Baseline	0.4222	0.1977	0.3983

Table 5: Comparison of the TextRank performance (using optimized parameters) with baseline methods.

The performance of the TextRank algorithm with the optimized parameters lies between the random baseline and the greedy oracle baseline. This outcome indicates that TextRank generates meaningful summaries significantly better than random selection, but does not achieve the theoretical upper bound represented by the oracle. More precisely:

- ROUGE-1: TextRank improves by approximately 24.4% over the random baseline, showing its effectiveness in capturing individual words that match the reference summaries.
- ROUGE-2: The improvement over the random baseline is even more pronounced (59.4%), highlighting the algorithm's ability to capture bigrams that match the reference summaries.

- ROUGE-L: TextRank also outperforms the random baseline by 24.6%, reflecting its capacity to retain the structure of the reference summaries.

While the ROUGE scores for TextRank are lower than those of the greedy oracle baseline, this gap is reasonable. The oracle method relies on an almost perfect selection of sentences to maximize ROUGE scores, a scenario that is not achievable by automatic summarization techniques, especially extractive ones. Moreover, the execution speed of TextRank (mean time of 0.0153s per article) can be considered relatively high.

In conclusion, the results demonstrate that TextRank achieves a strong balance between computational efficiency and summarization quality. Its performance is robust and significantly better than random selection, validating its effectiveness for real-world extractive summarization tasks.

6 Improvements

In this section, we present and evaluate a series of improvements to the TextRank algorithm, aimed at optimizing memory usage and computational speed while preserving the quality of the generated summaries. Specifically, we introduce two key enhancements: sparsifying the similarity graphs by applying a threshold and limiting similarity computations to a fixed window of neighboring sentences. For each approach, we optimize the new parameters based on our dataset, provide a Python implementation, and analyze its computational complexity. Finally, we apply these improvements to the dataset and compare their performance against the original TextRank algorithm.

6.1 Sparsifying the graphs

In this subsection, we introduce an optimization aimed at reducing the memory usage of the TextRank algorithm by sparsifying the similarity graphs. In the standard TextRank algorithm the similarity graph is represented by a dense adjacency matrix, where each element corresponds to the similarity score between two sentences. While this representation is justified in most cases because similarity graphs are relatively dense, it is possible that certain low similarity scores do not significantly impact the ranking process. Hence, sparsifying the graphs by introducing a threshold can help reduce memory usage without substantially affecting the algorithm's output quality.

In order to do so, we introduce a threshold, let us call it *threshold_sim*. Any similarity score below this threshold is set to 0, effectively removing the corresponding edge from the graph. This sparsification reduces the number of non-zero entries in the adjacency matrix, enabling the use of sparse matrix representations, which are significantly more memory-efficient than dense matrices.

6.1.1 Selecting *threshold_sim*

To determine for our dataset a suitable value for *threshold_sim*, we analyzed the mean density of the adjacency matrices for various thresholds using the sample dataset introduced in the previous section. Specifically, the density of an adjacency matrix is computed as the ratio of the number of non-zero entries (in this case non-zero similarity scores) to the total number of entries.

We tested a range of values from 0 to 0.5 with a step size of 0.05, and computed for each the mean density. The results are shown in Figure 5 below:

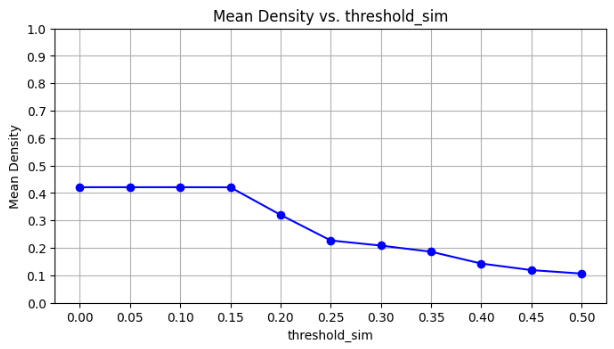


Figure 5: Mean density of the adjacency matrices as *threshold_sim* varies (sample dataset).

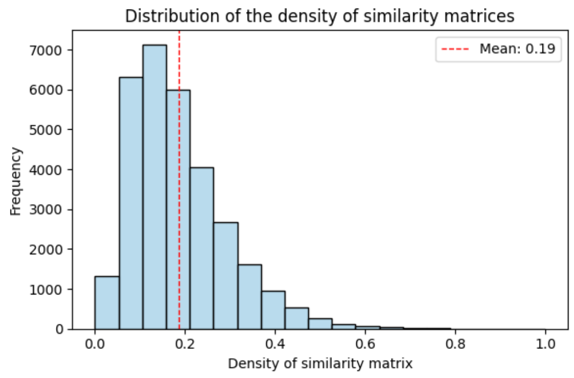


Figure 6: Distribution of the density of adjacency matrices when *threshold_sim* = 0.35 (sample dataset).

As we can see from Figure 5:

- When *threshold_sim*= 0 (which corresponds to the standard case), the mean density is a bit higher than 0.4, indicating that the graphs are relatively dense.
- As *threshold_sim* increases, the mean density decreases, reaching approximately 0.1 when *threshold_sim*= 0.5.

Based on this, a threshold of 0.35 was chosen. At this value, the mean density is approximately 0.19, representing a good balance between sparsity and retaining meaningful connections. Indeed, if we set *threshold_sim* too high we could risk losing important edges and make the algorithm's performance worse.
Additionally, Figure 6 shows the distribution of the density of adjacency matrices when *threshold_sim* = 0.35. As we can see, most graphs are sparsified to densities approximately between 0 and 0.25, supporting our choice.

6.1.2 Python implementation

Below is the updated Python implementation of the TextRank algorithm with sparsification:

```python
def textrank_sparse(sentences_as_words, damping, threshold_conv, threshold_sim,
    max_iter = float('inf')):

    # Compute the number of sentences
    n_sentences = len(sentences_as_words)

    # Initialize a sparse matrix using LIL format
    sparse_sim_matrix = lil_matrix((n_sentences, n_sentences))

    # Construct the sparse adjacency matrix based on sentence similarity
    for i, words_i in enumerate(sentences_as_words):
        for j in range(i, n_sentences):
            similarity_ij = sentence_similarity(words_i, sentences_as_words[j])
            # Only consider similarities above threshold_sim
            if similarity_ij > threshold_sim:
                sparse_sim_matrix[i, j] = similarity_ij
                sparse_sim_matrix[j, i] = similarity_ij

    # Convert to CSR format for efficient arithmetic operations
    sparse_sim_matrix = sparse_sim_matrix.tocsr()

    # Normalize the sparse adjacency matrix to make it stochastic
    row_sums = np.array(sparse_sim_matrix.sum(axis = 1)).flatten() + 1e-8
    sparse_sim_matrix = sparse_sim_matrix.multiply(1 / row_sums[:, np.newaxis])

    # Apply the power method
    transposed_matrix = sparse_sim_matrix.T
    # Initialize the PageRank vector (all sentences have equal probability
        initially)
    p_vector = np.ones(n_sentences) / n_sentences
    # Random jump value (applied dynamically during each iteration)
    random_jump_value = (1.0 - damping) / n_sentences
    # Initialize the convergence value
    lambda_val = float('inf')

    iter_count = 0
    while lambda_val > threshold_conv and iter_count < max_iter:
        next_p = transposed_matrix.dot(p_vector)
        # Apply the random jump as part of the iteration
        next_p = damping * next_p + random_jump_value
        lambda_val = np.linalg.norm(next_p - p_vector)
        # Update the PageRank vector for the next iteration
        p_vector = next_p
        iter_count += 1

    # Rank the sentences based on their PageRank scores
    ranked_sentences = sorted(enumerate(p_vector), key = lambda x: x[1], reverse = True)

    return ranked_sentences
```

The `textrank_sparse` function introduces two key implementation changes compared to the `textrank` function:

- The similarity computation incorporates the threshold *threshold_sim* to sparsify the adjacency matrix. In particular, the function implements the sparse adjacency matrix using the LIL (List of Lists) format, which is efficient for incremental construction. The matrix is then converted to CSR (Compressed Sparse Row) format for efficient arithmetic operations.
- In the standard TextRank algorithm, the random jump factor $(1-damping)/N$ is added before the power method to the transition matrix. In the sparse implementation, this adjustment is applied dynamically during each iteration of the power method, in order to preserve the sparsity of the transition matrix.[6]

6.1.3 Computational complexity

To analyze the computational complexity of the `textrank_sparse` function, we consider the following key parameters: the number N of sentences in the text, the average number of words per sentence W, the maximum number of iterations k for the power method, and the density d of the adjacency matrix after sparsification (which depends on the chosen value of *threshold_sim*).

As for time complexity, we have that:

- The sparse adjacency matrix is initialized in LIL format and populated by calculating pairwise sentence similarities $O(N^2)$ times, each time requiring $O(W)$. This results in a complexity of $O(N^2 \cdot W)$. Edge insertion into the LIL matrix takes $O(1)$ per non-zero entry ; since the number of non-zero entries is proportional to $d \cdot N^2$, this requires $O(d \cdot N^2)$. Converting the matrix from LIL to CSR adds another $O(d \cdot N^2)$. Together, these steps have a total complexity of $O(N^2 \cdot W + d \cdot N^2)$. Since $d \ll W$, it simplifies to $O(N^2 \cdot W)$.
- Constructing the stochastic transition matrix involves summing the rows of the sparse matrix and normalizing each row. This step has a complexity of $O(d \cdot N^2)$.
- Each iteration of the power method involves a sparse matrix-vector multiplication, which operates only on the non-zero entries of the matrix. The time complexity per iteration is $O(d \cdot N^2)$. Over a maximum of k iterations, this becomes $O(k \cdot d \cdot N^2)$.

[6]It can be mathematically proven that, in terms of the final result, applying the jump dynamically during the power method is equivalent to pre-adjusting the transition matrix.

Hence, the total time complexity is $O(N^2 \cdot W + k \cdot d \cdot N^2)$. Notice that in the worst-case scenario, which corresponds to $d = 1$ (fully connected graph), we obtain the same time complexity as the `textrank` function. However, when $d \ll 1$, there is a noticeable improvement.

As for space complexity, we have that:

- The input space is the same as the one of `textrank` function, $O(N \cdot W)$: the two functions have the same inputs except for *threshold_sim*, which is a float, hence it requires $O(1)$ space.
- As for the auxiliary space:
  - The LIL matrix initially stores approximately $d \cdot N^2$ non-zero entries, requiring $O(d \cdot N^2)$ space. After the conversion to CSR format, additional row and column pointers are required, but the memory usage remains proportional to the number of non-zero entries. Hence, the space complexity here is $O(d \cdot N^2)$.
  - The transposition step requires space proportional to the number of non-zero entries at a cost of $O(d \cdot N^2)$.
  - The power method maintains two vectors of size N for each iteration, `next_p` and `p_vector`. Temporary vectors used in computations also scale linearly with N. Hence, the space complexity for this step is $O(N)$.

The overall space complexity is $O(N \cdot W + d \cdot N^2)$. As for time complexity, in the worst-case scenario, this matches the `textrank` function. However, for sparsified graphs we have a significant reduction.

When applying this version of the TextRank algorithm to our dataset, we expect the use of sparse matrices to significantly reduce the average peak memory usage. Indeed, for our dataset, using a sparsification threshold of $threshold\_sim = 0.35$ results in adjacency matrices with an average density of $\hat{d} \approx 0.19 \ll 1$. Moreover, since the input space remains identical across the two versions, it will be excluded from the comparison.
In terms of execution time, while sparsification theoretically reduces time complexity, we do not expect a significant reduction for our dataset. This is because the adjacency matrix construction step ($O(N^2 \cdot W)$) dominates for our dataset, where the average number of words per sentence is $\hat{W} \approx 23$, which far exceeds the fixed number of iterations in the power method ($k = 8$). As a result, the time savings from sparsification are outweighed by the cost of calculating pairwise similarities.

Overall, this version of the algorithm is more scalable, especially for datasets with many long articles, as the memory usage grows slower compared to the dense matrix implementation. However, sparsification is not without risks. If the chosen threshold eliminates edges representing meaningful connections, it could negatively affect the quality of rankings. Careful selection of the threshold is critical to balancing sparsity and the algorithm's performance.

6.2 Limiting sentence similarity computations

In this subsection, we describe a significant improvement to the TextRank algorithm's efficiency by restricting sentence similarity computations to a fixed window size w. Specifically, we compute similarity scores only for pairs of sentences that are at most w positions apart. This restriction reduces the computational cost associated with constructing the adjacency matrix without significantly affecting the algorithm's performance. Importantly, this optimization is implemented in conjunction with sparsification, ensuring that only meaningful similarities (above a given threshold) are retained.

The idea of limiting similarity computations to a fixed window size arises from the observation that sentence similarity decreases with distance. To verify this for our dataset, we analyzed how the similarity scores between sentences change as a function of their positional distance in the text. Specifically, using the representative sample, we computed the mean similarity scores for sentences separated by increasing distances and plotted the results:

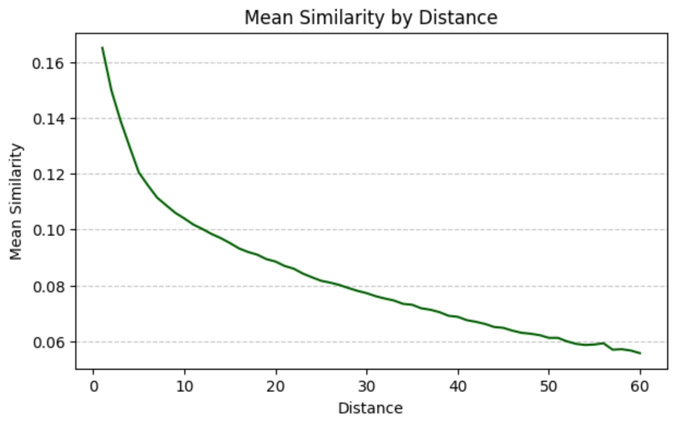


Figure 7: Mean similarity scores as distance between sentences varies (sample dataset).

The above line plot clearly shows that similarity scores decrease rapidly as the distance increases, confirming that closer sentences tend to have stronger semantic connections. This trend suggests that focusing similarity computations on close sentences is both computationally efficient and consistent with the characteristics of textual data.

6.2.1 Python implementation

We present below the Python implementation of the adjacency matrix construction for the improved algorithm, which incorporates the window size w. The rest of the algorithm remains identical to the `textrank_sparse` function and is therefore omitted for brevity.

```python
def textrank_sparse_w(sentences_as_words, damping, threshold_conv, threshold_sim, w
    , max_iter = float('inf')):
    ...
    # Construct the sparse adjacency matrix based on sentence similarity
    for i, words_i in enumerate(sentences_as_words):
        end_idx = min(n_sentences, i + w + 1)  # Ensure we don't go beyond the last
            index
        for j in range(i, end_idx):
            similarity_ij = sentence_similarity(words_i, sentences_as_words[j])
            # Only consider similarities above threshold_sim
            if similarity_ij > threshold_sim:
                sparse_sim_matrix[i, j] = similarity_ij
                sparse_sim_matrix[j, i] = similarity_ij
    ...
```

6.2.2 Computational complexity

Below, we briefly discuss the time and space complexity of `textrank_sparse_w`, highlighting the main differences from `textrank_sparse`. Let N be the number of sentences, W the average number of words per sentence, k the maximum number of iterations for the power method, w the window size and d_w the density of the adjacency matrix after sparsification restricted to the computed similarities (it depends on the chosen values for *threshold_sim* and w).

The time complexity of the algorithm is determined by two main components:

- In `textrank_sparse`, pairwise similarity is computed for all pairs of sentences, resulting in $O(N^2)$ comparisons. By contrast, `textrank_sparse_w` computes similarities only within a window of size w around each sentence. Specifically, the algorithm compares each sentence with approximately $2w$ neighboring sentences (the actual comparisons are approximately w as similarities are symmetric). Since each similarity computation costs $O(W)$, the overall time complexity for this is $O(N \cdot w \cdot W)$. Moreover, the number of non-zero entries of the adjacency matrix after sparsification is proportional to $N \cdot w \cdot d_w$, so adding edges to the LIL matrix costs an additional $O(N \cdot w \cdot d_w)$. Since $W \gg d_w$, the total cost simplifies to $O(N \cdot w \cdot W)$.

- Each iteration of the power method involves a sparse matrix-vector multiplication, which operates only on the non-zero entries of the matrix. Since the matrix contains approximately $N \cdot w \cdot d_w$ non-zero entries, the cost of one iteration is $O(N \cdot w \cdot d_w)$. Over k iterations, the total cost is $O(k \cdot N \cdot w \cdot d_w)$.

Hence, the overall time complexity is $O(N \cdot w \cdot W + k \cdot N \cdot w \cdot d_w)$. Notice that when $w = N$, this coincides with the time complexity of `textrank_sparse`, but for $w \ll N$, the performance gain is significant, particularly for longer texts.

As for space complexity:

- The input space remains $O(N \cdot W)$, as the two functions have the same input except for w, which is an integer requiring contant space.

- The auxiliary space for `textrank_sparse_w` is similar to `textrank_sparse`, except for the adjacency matrix, which now depends on w. Indeed, the LIL matrix stores approximately $d_w \cdot w \cdot N$ non-zero entries and after the conversion to CSR format, the memory usage remains proportional to the number of non-zero entries. This also applies during the transposition step. Hence, the space complexity here becomes $O(d_w \cdot w \cdot N)$.

The total space complexity is $O(N \cdot W + d_w \cdot w \cdot N)$, which improves the complexity of `textrank_sparse` especially when $w \ll N$. Again, if $w = N$, the space complexity matches that of `textrank_sparse`.

When applying this function to our dataset with optimized parameters, we expect improvements in both execution time and peak memory usage. Indeed, restricting the similarity computation to a window of size w reduces the number of comparisons from $O(N^2)$ to $O(Nw)$, and sparsifying the adjacency matrix further reduces the overhead for matrix operations. Of course, these optimizations are particularly effective for large texts, where $w \ll N$ and sparsification produces a low d_w.

6.2.3 Choosing the window size w

To determine the optimal value of w for our dataset, we evaluated the performance of the function `textrank_sparse_w` on the sample dataset using various window sizes, specifically $w \in \{3, 5, 8, 10, 12, 15\}$. We stopped at $w = 15$ as this value is less than half the mean number of sentences per article in our dataset (≈ 37.19). For each value of w, we applied the algorithm using the previously optimized parametrs, and measured both the mean execution time and the mean ROUGE-1, ROUGE-2, and

ROUGE-L scores. The results are summarized in the table below:

w	**Mean Execution Time (s)**	**ROUGE-1 Mean**	**ROUGE-2 Mean**	**ROUGE-L Mean**
3	0.0042	0.2676	0.0823	0.2456
5	0.0053	0.2730	0.0855	0.2508
8	0.0069	0.2779	0.0883	0.2554
10	0.0078	0.2805	0.0898	0.2578
12	0.0087	0.2828	0.0913	0.2602
15	0.0097	0.2854	0.0931	0.2626

Table 6: Metrics obtained for different window sizes (w) using `textrank_sparse_w` on the sample dataset with the following parameters: $damping = 0.85$, $threshold\_conv = 10^{-3}$, $threshold\_sim = 0.35$ and $max\_iter = 8$.

The results reveal a clear trade-off between execution time and summarization quality: as w increases, the ROUGE scores gradually improve, reflecting a more comprehensive consideration of sentence similarities. However, this comes at the cost of increased execution time.

We chose as optimal window size $w = 12$ since it balances efficiency and accuracy: the ROUGE scores are close to the highest values observed, indicating a robust summarization performance. Additionally, the mean execution time ($0.0087s$) remains manageable and only slightly higher than the one corresponding to $w = 10$.

6.2.4 Dynamic w

To further enhance computational efficiency, we propose a slight variation that makes the parameter w 'dynamic', meaning its value decreases as we progress through the text. This approach is motivated by the observation that the most important sentences tend to be located at the beginning of the text, especially in articles. To verify this for our dataset, we analyzed the positional frequencies of sentences selected by the functions `textrank` and `textrank_sparse`. The results, illustrated in the bar plots below, indicate that both algorithms consistently select more sentences from the initial positions of the text.

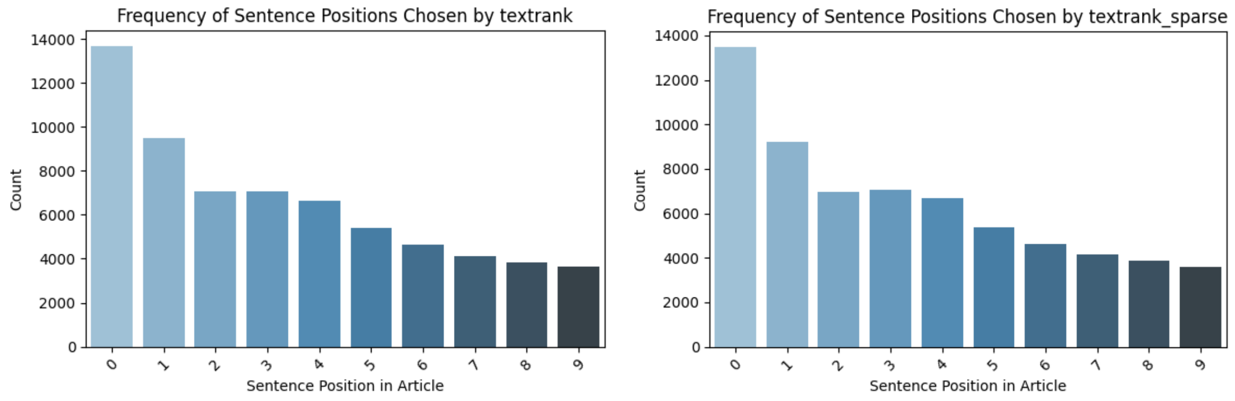


Figure 8: Barplots showing the frequencies of the positions of the sentences selected by `textrank` (left) and `textrank_sparse` (right), using the sample dataset.

The plots suggest that sentences near the beginning of a text are often the most relevant. Consequently, it is essential to compute almost all similarities involving these initial sentences. On the contrary, as we move further into the text, sentences tend to be less important for summarization. Hence, by gradually reducing w, we can skip computing similarities between less relevant sentences, improving efficiency while maintaining a good performance.

The dynamic w is computed using a simple linear interpolation function that adjusts w based on the sentence's relative position in the text. The implementation is as follows:

```python
def compute_dynamic_w(n_sentences, i, max_w, min_w = 1):
    """
    Compute a dynamic value of w based on the sentence position.

    Parameters:
    - n_sentences: int
       Total number of sentences in the article
    - i: int
       Current sentence index
    - max_w: int
       Maximum value of w
    - min_w: int
       Minimum value of w

    Returns:
    - dynamic_w: int
       Dynamic value of w for the current sentence
    """
    # Calculate the relative position of the sentence in the article
    relative_position = i / (n_sentences - 1)

    # Compute w using a linear interpolation between max_w and min_w
    dynamic_w = int(max_w - (max_w - min_w) * relative_position)

    return max(dynamic_w, min_w)
```

This function is used when building the adjacency matrix to update w dynamically before determining the *end_idx* variable:

```python
def textrank_sparse_w_dynamic(sentences_as_words, damping, threshold_conv,
   threshold_sim, w, max_iter = float('inf')):
    ...
    # Store the initial w value
    max_w = w
    # Construct the sparse adjacency matrix based on sentence similarity
    for i, words_i in enumerate(sentences_as_words):
        w = compute_dynamic_w(n_sentences, i, max_w)
        end_idx = min(n_sentences, i + w + 1)  # Ensure we don't go beyond the last
             index
        for j in range(i, end_idx):
            similarity_ij = sentence_similarity(words_i, sentences_as_words[j])
            # Only consider similarities above threshold_sim
            if similarity_ij > threshold_sim:
                sparse_sim_matrix[i, j] = similarity_ij
                sparse_sim_matrix[j, i] = similarity_ij
    ...
```

This approach allows the algorithm to focus on computing similarities for earlier sentences, which are more likely to be important for the summary, significantly reducing the computation time for longer texts.

6.3 Comprehensive evaluation

To evaluate the performance of the introduced improvements (`textrank_sparse`, `textrank_sparse_w`, and `textrank_sparse_w_dynamic`) and compare them with the original `textrank` function, we applied all the algorithms to the entire dataset. The parameters were kept consistent with those optimized in Section 5: a convergence threshold of 10^{-3}, a damping factor of 0.85, and a maximum of $k = 8$ iterations for the power method. Additionally, the new optimized parameters were used: a sparsification threshold of 0.35 and a window size of $w = 12$ to limit similarity computations (for both the fixed and dynamic versions).

We evaluated the algorithms based on the following metrics: mean execution time per article, mean ROUGE-1, ROUGE-2, ROUGE-L scores, and mean peak memory usage. Peak memory usage was measured using the `tracemalloc` library. It is important to note that this measurement does not include the input space, focusing solely on the memory required during algorithm execution. Moreover, due to computational constraints, memory usage was measured on the sample dataset. The results are summarized in the table below.

Algorithm	**Mean Time (s)**	**ROUGE-1**	**ROUGE-2**	**ROUGE-L**	**Memory Usage (MB/article)**
textrank	0.0153	0.2952	0.0998	0.2721	0.0577
textrank_sparse	0.0164	0.2971	0.1005	0.2738	0.0187
textrank_sparse_w	0.0088	0.2829	0.0914	0.2601	0.0134
textrank_sparse_w_dynamic	0.0068	0.2884	0.0954	0.2655	0.0134

Table 7: Performance metrics for different TextRank variants.

As we can see, the results confirm a significant reduction in memory usage from the standard `textrank` algorithm to its sparsified versions. In particular:

- For `textrank_sparse`, memory usage decreases by approximately 68%, mainly due to the reduced number of connections stored in the adjacency matrix after sparsification.
- The `textrank_sparse_w` and `textrank_sparse_w_dynamic` algorithms further reduce memory usage, as limiting similarity computations within a window w lowers the number of entries in the adjacency matrix.

As for execution time, `textrank` and `textrank_sparse` exhibit comparable results, which is due to the fact that sparsification does not impact the initial similarity computations. As expected, the introduction of the window size w in `textrank_sparse_w` significantly reduces execution time by limiting similarity computations. Finally, the dynamic approach further improves efficiency, with the average time decreasing by over 56% compared to `textrank`.
As for ROUGE scores, surprisingly `textrank_sparse` yields slightly higher ROUGE scores than `textrank`. This improvement may result from eliminating weak connections that could interfere with the ranking process, enhancing the coherence of the extracted summaries. On the contrary, introducing the window size w leads to a small reduction (approximately 0.01) in ROUGE scores, with the dynamic approach reaching slightly better results. This is expected, as limiting comparisons to a neighborhood can exclude potentially relevant connections. However, the reduction in ROUGE scores is low compared to the gains in efficiency.

In conclusion, the results demonstrate that sparsification and window-based optimizations significantly improve both memory efficiency and execution time, with a minimal impact on ROUGE scores. In particular, the dynamic window approach strikes a balance between computational efficiency and summary quality, making it particularly suitable for longer texts.

7 Conclusions

In this project, we explored the problem of text summarization, focusing on the extractive approach, which involves selecting and combining sentences directly from the original text to create a summary. Specifically, the problem we wanted to solve was the following: extract exactly four sentences from a given document, ensuring that the resulting summary captured the most significant aspects of the original text.
To solve this problem, we used the TextRank algorithm, which is a graph-based method. TextRank consists of two key phases: graph construction and PageRank algorithm. In the graph construction step, sentences are represented as nodes, and their edges are weighted by sentence similarity. PageRank is then applied to rank the sentences based on their relevance to the document. The $top - 4$ sentences are then selected to form the summary.

In Section 5, we implemented TextRank in Python and applied it on the subset of the CNN/-DailyMail dataset including only the articles whose highlights contained exactly four sentences. Moreover, we evaluated its performance in terms of mean execution time, and mean ROUGE-1, ROUGE-2, and ROUGE-L scores, providing both a measure of computational efficiency and a quantitative evaluation of summarization quality. By comparing the ROUGE scores with two baselines - a random selection of sentences and a greedy oracle baseline (which provide a lower and an upper bound for the scores, respectively) - we demonstrated that TextRank strikes a strong balance between computational efficiency and summarization quality. It significantly outperformed the random baseline, confirming its effectiveness for extractive summarization and, although its ROUGE scores were lower than those of the oracle baseline, the gap was reasonable.

We conducted a detailed analysis of TextRank's computational complexity, which is $O((W + k)N^2)$ in time and $O(NW + N^2)$ in space, where N is the number of sentences in the text, W the average number of words per sentence, and k the maximum number of iterations for the power method. These complexities ensure robustness and high-quality results especially for small to medium-sized texts, where N and W are modest, making it particularly suited for tasks such as summarizing articles. However, the quadratic dependence on N can make TextRank computationally expensive and memory-intensive for long documents, which is a notable limitation.

To address this limitation, in Section 6 we proposed and implemented two optimizations:

1. The first one involves sparsifying the similarity graphs by applying a threshold: any similarity score below this threshold is set to 0, effectively removing the corresponding edge from the graph. This sparsification reduces the number of non-zero entries in the adjacency matrix, enabling the use of sparse matrix representations, which are significantly more memory-efficient than dense matrices.

2. The second one involves limiting the similarity computations to a fixed window of size w around each sentence, to reduce both execution time and memory usage. We also introduced a dynamic window strategy in which the window size w decreases as we progress through the text. Both cases were implemented in conjunction with sparsification.

These optimizations were implemented in the variants `textrank_sparse`, `textrank_sparse_w` and `textrank_sparse_w_dynamic`. By applying them to the same dataset, we evaluated their performance and compared it to `textrank`'s in terms of mean execution time, mean ROUGE scores, and mean peak memory usage (the last one computed on a representative sample for computational reasons). The results, summarized in the bar plots below, highlight the trade-offs between efficiency and accuracy:

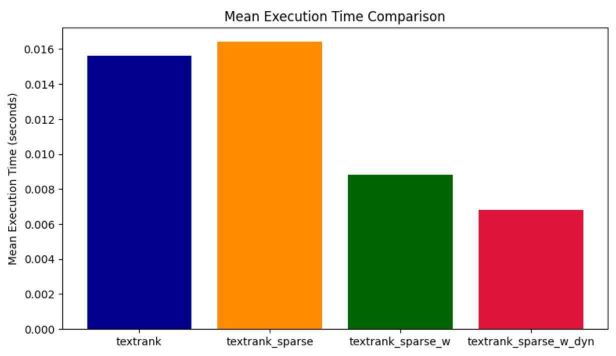


Figure 9: Comparison of mean execution times across all TextRank variants.

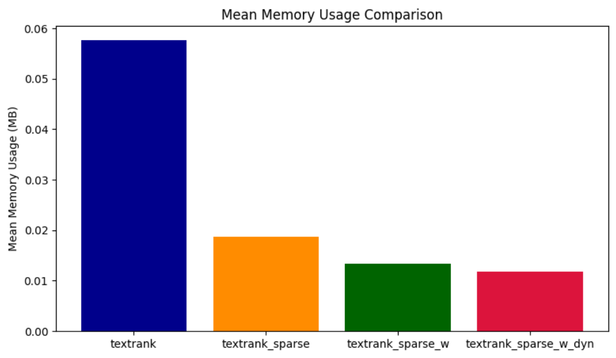


Figure 10: Comparison of mean peak memory usage across all TextRank variants.

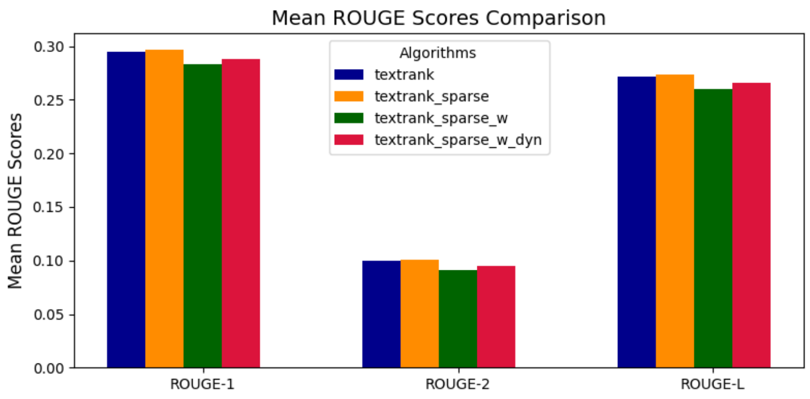


Figure 11: Comparison of mean ROUGE scores across all TextRank variants.

We can notice that:

- The windowed approaches (`textrank_sparse_w` and `textrank_sparse_w_dynamic`) achieved substantial reductions in execution time compared to the original `textrank` algorithm, with the dynamic window approach achieving the fastest performance.
- Sparsification and windowing significantly lowered memory usage compared to the original algorithm.
- While `textrank_sparse_w` and `textrank_sparse_w_dynamic` showed slight reductions in ROUGE scores compared to the original `textrank` algorithm, the dynamic window approach partially mitigated this loss, demonstrating a robust balance between summarization quality and computational efficiency.

Overall, the proposed optimizations significantly improved the efficiency of TextRank algorithm while maintaining competitive summarization quality. These enhancements make TextRank more practical for large-scale and resource-constrained applications. In particular, the dynamic window approach suggests opportunities for further exploration and refinement in future work, including exploring data-driven or adaptive window-sizing techniques tailored to specific text structures.

References

[ER04] G. Erkan and D. R. Radev. “LexRank: Graph-based Lexical Centrality as Salience in Text Summarization”. In: *Journal of Artificial Intelligence Research* 22 (Dec. 2004), pp. 457–479. URL: `http://dx.doi.org/10.1613/jair.1523`.

[GL01] Yihong Gong and Xin Liu. “Generic text summarization using relevance measure and latent semantic analysis”. In: *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, 2001, pp. 19–25. URL: `https://doi.org/10.1145/383952.383955`.

[GMK23] Nikolaos Giarelis, Charalampos Mastrokostas, and Nikos Karacapilidis. “Abstractive vs. Extractive Summarization: An Experimental Review”. In: *Applied Sciences* 13.13 (2023), p. 7620. URL: `https://doi.org/10.3390/app13137620`.

[Lin04] Chin-Yew Lin. “ROUGE: A Package for Automatic Evaluation of Summaries”. In: *Annual Meeting of the Association for Computational Linguistics*. 2004. URL: `https://api.semanticscholar.org/CorpusID:964287`.

[Liu19] Yang Liu. *Fine-tune BERT for Extractive Summarization*. 2019. URL: `https://arxiv.org/abs/1903.10318`.

[Luh58] H. P. Luhn. “The Automatic Creation of Literature Abstracts”. In: *IBM Journal of Research and Development* 2.2 (1958), pp. 159–165.

[MT04] Rada Mihalcea and Paul Tarau. “TextRank: Bringing Order into Text”. In: *Conference on Empirical Methods in Natural Language Processing*. 2004. URL: `https://api.semanticscholar.org/CorpusID:577937`.

[Pag+99] Lawrence Page et al. “The PageRank Citation Ranking : Bringing Order to the Web”. In: *The Web Conference*. 1999.

[Wik24] Wikipedia. *PageRank — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=PageRank&oldid=1258385219`. 2024.