

Algorithm design & Analysis Project

CS315 – Group:338

Semester:461

Text Similarity

Muhannad Majed Al-Fawzan 431108107

Mohammed Mansour Al-Bulihy 431107867

Mohammed Saleh Al-Saif 431108031

Eyad Fahad Al-Arfaj 431107974

Supervisor: Prof. Dr. Mohammed Al-Hagery

Table of Contents

1	Overview	
1.1	Introduction.....	3
1.2	Problem Description	4
2	Theoretical Analysis	
2.1	Naïve Algorithm	5
2.1.1	Pseudocode	6
2.1.2	Analysis.....	7
2.1.3	Example	10
2.2	Optimized Algorithm.....	12
2.2.1	Pseudocode	13
2.2.2	Analysis.....	14
2.2.3	Example	17
2.3	Comparison	19
3	Empirical Analysis	
3.1	Naïve Algorithm Implementation	20
3.2	Optimized Algorithm Implementation.....	21
3.3	Performance Analysis	22
4	Conclusion	
4.1	Comparison	23
4.2	Final Thoughts	23

Overview

1.1 Introduction:

In this project, we are trying to find how similar two texts are by using the Jaccard similarity. This is important for things like checking for plagiarism, comparing documents, or finding similar content.

Jaccard similarity works by treating each text as a set of unique words. It then compares these sets by looking at how many words the two texts have in common (intersection) and divides that by the total number of different words in both texts (union). The result is a number between 0 and 1, where 0 means the texts are completely different, and 1 means they are exactly the same.

The main goal of this project is to create an algorithm that calculates the Jaccard similarity between two texts. This method is simple and helps us easily measure how alike two texts are.

$$Jaccard = \frac{|A \cap B|}{|A \cup B|}$$

Figure 1.1(Jaccard Similarity equation where is A&B are the sets of unique words in text1 & text2 respectively)

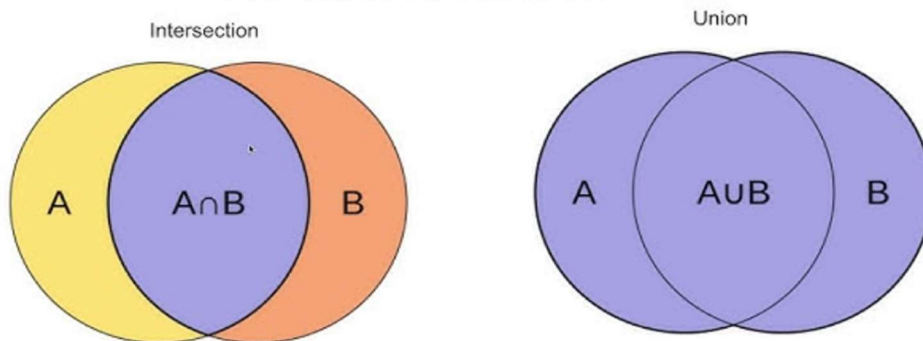


Figure 1.2(Intersection & union illustration)

1.2 Problem Description

The problem is to compare two given texts and calculate their similarity using the Jaccard coefficient. The algorithm should take in two text strings, break them into sets of unique words, and calculate the similarity based on the size of the intersection and union of the two sets.

Input	Output
Text 1: "apple banana orange" Text 2: "banana orange grape"	Jaccard Similarity: 0.5
Text 1: "dog cat fish" Text 2: "dog cat fish"	Jaccard Similarity: 1.0
Text 1: "hello world" Text 2: "hello there"	Jaccard Similarity: 0.33


Figure 1.3 (Input output Examples)

Theoretical Analysis

2.1 Naive Algorithm

Description(Rationale):

The naive algorithm of Jaccard similarity works by comparing two input texts and counting the number of common unique words (intersection) and the total number of unique words across both texts (union). The algorithm proceeds as follows:

- 
1. **Input Processing:** It accepts two text strings, converts them to lowercase, and splits them into lists of words.
 2. **Union List Construction:** It constructs a union list containing unique words from the first text, ensuring that duplicates are eliminated.
 3. **Intersection Count:** It counts how many words from the first text are also present in the second text.
 4. **Completing the Union:** The algorithm adds any unique words from the second text that are not already in the union list.
 5. **Jaccard Similarity Calculation:** Finally, it computes the Jaccard similarity by dividing the intersection count by the union count, returning this value as the result.

2.1.1 Naive Algorithm: Pseudocode

Algorithm 1: Naive Jaccard Similarity Algorithm

Inputs: *text1*: First input text

text2: Second input text

Output: *similarity*: Jaccard similarity between the two texts

Function naive_jaccard_similarity(*text1*, *text2*):

```
words1 ← split(text1.lower());
words2 ← split(text2.lower());
union_list ← empty list;
union_count ← 0;
for each word in words1 do
    flag_comp ← TRUE;
    for each unique_word in union_list do
        if word == unique_word then
            flag_comp ← FALSE;
            break;
        end
    end
    if flag_comp == TRUE then
        union_list.append(word);
        union_count ← union_count + 1;
    end
end
intersection_count ← 0;
for each word in union_list do
    for each compare_word in words2 do
        if word == compare_word then
            intersection_count ← intersection_count + 1;
            break;
        end
    end
end
for each word in words2 do
    flag_comp ← TRUE;
    for each unique_word in union_list do
        if word == unique_word then
            flag_comp ← FALSE;
            break;
        end
    end
    if flag_comp == TRUE then
        union_list.append(word);
        union_count ← union_count + 1;
    end
end
if union_count == 0 then
    return 0.0;
end
return intersection_count / union_count;
```



Figure 2.1(Naive Pseudocode Algorithm)

2.1.2 Naive Algorithm: Analysis

	#operations
1. words1 \leftarrow split(text1.lower())	$C1+N+1$
2. words2 \leftarrow split(text2.lower())	$C2+M+1$
3. union_list \leftarrow empty list	1
4. union_count \leftarrow 0	1
5. for each word in words1 do	N
6. flag_comp \leftarrow TRUE	$1*N$
7. for each unique_word in union_list do	$N*N$
8. if word == unique_word then	$1*N*N$
9. flag_comp \leftarrow FALSE	$1*N*N$
10. break	$1*N*N$
11. end if	
12. end for	
13. if flag_comp == TRUE then	$1*N$
14. union_list.append(word)	$1*N$
15. union_count \leftarrow union_count + 1	$2*N$
16. end if	
17. end for	
18. intersection_count \leftarrow 0	1
19. for each word in union_list do	N
20. for each compare_word in words2 do	$N*M$
21. if word == compare_word then	$1*N*M$
22. intersection_count \leftarrow intersection_count + 1	$2*N*M$
23. break	$1*N*M$
24. end if	
25. end for	
26. end for	
27. for each word in words2 do	M
28. flag_comp \leftarrow TRUE	$1*M$
29. for each unique_word in union_list do	$(N+M)*M$
30. if word == unique_word then	$1*(N+M)*M$
31. flag_comp \leftarrow FALSE	$1*(N+M)*M$
32. break	$1*(N+M)*M$
33. end if	
34. end for	

35.	if flag_comp == TRUE then	1*M
36.	union_list.append(word)	1*M
37.	union_count ← union_count + 1	2*M
38.	end if	
39.	end for	
40.	if union_count == 0 then	1
41.	return 0.0	1
42.	end if	
43.	return intersection_count / union_count	2

Time Complexity Analysis of the Naive Algorithm:

The naive algorithm consists of several key operations: splitting the input texts into words, building a union of unique words, and calculating the intersection of common words. Below is a breakdown of the time complexity for each operation:

Splitting the Texts:

The texts are split into words using the split function, which operates in linear time with respect to the number of words.

Time Complexity: $O(C1+N+C2+M)$, where $C1$ is the number of characters of text1, $C2$ is the number of characters of text2, N is the number of words of text1, and M is the number of words of text2.

Building the Union List:

For each word in text1, we check if it is already in the union_list. This involves a nested loop, leading to a worst-case scenario where each word in text1 is compared with all unique words in union_list.

Time Complexity: $O(N^2)$ in the worst case.

Calculating the Intersection:

For each word in the union_list, we compare it with every word in text2 to count the common words, resulting in a quadratic complexity.

Time Complexity: $O(N \times M)$.

Adding Remaining Words from text2 to union_list:

Similar to building the union list from text1, we iterate through text2 and add unique words.

Time Complexity: $O(M^2)$.

Overall Time Complexity:

The overall time complexity of the naive algorithm is determined by the most significant terms. Therefore, it can be written as: $O(N^2 + M^2 + N \times M)$

In the end, this simplifies to: $O(N^2)$.

Space Complexity Analysis of the Naive Algorithm:

The space complexity of the algorithm reflects the amount of memory used relative to the size of the input. Below, we provide a detailed breakdown of how memory is utilized throughout the algorithm.

words1 and words2 Storage:

The words from text1 and text2 are stored in two lists. If text1 has N words and text2 has M , the space required is:

Space Complexity: $O(N + M)$.

union_list:

This list holds the unique words from both texts. In the worst case, it stores all words, so:

Space Complexity: $O(N + M)$.

Miscellaneous Variables:

Variables like counters and flags require constant memory:

Space Complexity: $O(1)$.

Total Space Complexity

The total space complexity is $O(N + M)$, as the memory usage is dominated by storing the words and unique words from both texts. This ensures efficient memory usage, even for large inputs.

Algorithm Scenarios:

The time complexity of the naive Jaccard similarity algorithm varies based on the characteristics of the input texts:

Best Case:

When both texts have few unique words, the complexity is linear: $O(n)$

Average Case:

Under typical conditions with moderate uniqueness, the complexity is quadratic: $O(n^2)$

Worst Case:

When both texts contain many unique words, the complexity remains quadratic: $O(n^2)$

Summary:

Best Case: $O(n)$

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

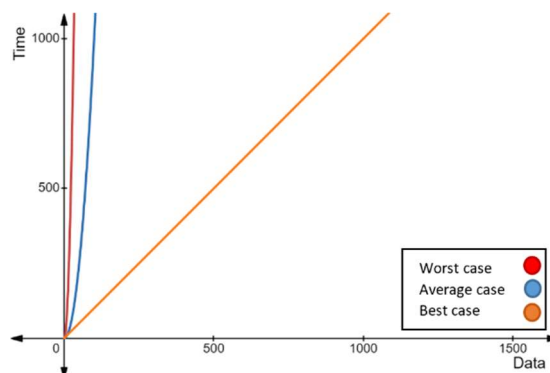


Figure 2.2 (Naive Time complexity graph)

2.1.3 Naive Algorithm: Example

Input:

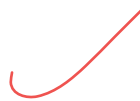
- Text 1: "The quick brown fox jumps over the lazy dog"
- Text 2: "The quick blue hare jumps over the lazy dog"

Step 1: Preprocessing

- Convert both texts to lowercase and split them into lists of words:
 - Words from Text 1:
 - ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
 - Words from Text 2:
 - ["the", "quick", "blue", "hare", "jumps", "over", "the", "lazy", "dog"]

Step 2: Create Union Set from Text 1

- Initialize an empty union list for Text 1.
- Iterate through each word in Text 1:
 - Add "the" to the union list.
 - Add "quick" to the union list.
 - Add "brown" to the union list.
 - Add "fox" to the union list.
 - Add "jumps" to the union list.
 - Add "over" to the union list.
 - Add "lazy" to the union list.
 - Add "dog" to the union list.
- Union List After Processing Text 1:
 - ["the", "quick", "brown", "fox", "jumps", "over", "lazy", "dog"]
- Union Count for Text 1: 8



Step 3: Count Intersection with Text 2

- Initialize the intersection count to zero.
- Iterate through each word in the Union List created from Text 1:
 - "the" → exists in Text 2 (increment intersection count)
 - "quick" → exists in Text 2 (increment intersection count)
 - "brown" → exists in Text 1 only (skip)
 - "fox" → exists in Text 1 only (skip)
 - "jumps" → exists in Text 2 (increment intersection count)
 - "over" → exists in Text 2 (increment intersection count)
 - "lazy" → exists in Text 2 (increment intersection count)
 - "dog" → exists in Text 2 (increment intersection count)

- Final Intersection Count: 6

Step 4: Create Union Set from Text 2

- Initialize a union list for Text 2.
- Iterate through each word in Text 2:
 - "the" → already exists in the union list (skip)
 - "quick" → already exists in the union list (skip)
 - Add "blue" to the union list.
 - Add "hare" to the union list.
 - "jumps" → already exists in the union list (skip)
 - "over" → already exists in the union list (skip)
 - "lazy" → already exists in the union list (skip)
 - "dog" → already exists in the union list (skip)
- Final Union List After Processing Text 2:
 - ["the", "quick", "brown", "fox", "jumps", "over", "lazy", "dog", "blue", "hare"]
- Final Union Count: 10

Step 5: Calculate Jaccard Similarity

- Use the formula to compute the Jaccard similarity:
$$\text{Jaccard Similarity} = \text{Intersection Count} / \text{Union Count} = 6 / 10 = 0.6$$

Output:

- The Jaccard similarity between Text 1 and Text 2 is 0.6 or 60%.

2.2 Optimized Algorithm

Description (Rationale):

The optimized algorithm for Jaccard similarity improves efficiency by using dictionaries to check for word existence and directly calculating the union and intersection of the two input texts. This approach minimizes redundant comparisons and reduces the overall complexity. The algorithm proceeds as follows:

6. **Input Processing:** It accepts two text strings, converts them to lowercase, and splits them into lists of words.
7. **Word Dictionaries:** It constructs two dictionaries, one for each text, that store the words as keys. For each word in the texts, the algorithm either skips to the next word if the word has already been added in the dictionary or adds the word to the dictionary with (None) as a value.
8. **Initial Union Calculation:** The union count is initialized as the combined size of the two dictionaries, representing the total number of unique words in both texts.
9. **Intersection and Union:** The algorithm iterates over the words in the first dictionary. For each word found in both dictionaries, it increments the intersection count. After calculating the intersection count, we update the union count value to be the initial union count value minus the intersection count.
10. **Jaccard Similarity Calculation:** If there are no words in the union (i.e., if the union count is zero), the algorithm returns 0 to avoid division by zero. Otherwise, it computes the Jaccard similarity by dividing the intersection count by the union count and returns this value as the result.

2.2.1 Optimized Algorithm: Pseudocode

Algorithm 2: Optimized Jaccard Similarity Algorithm

text1: First input text *text2*: Second input text

similarity: Jaccard similarity between the two texts

Function `optimized_jaccard_similarity(text1, text2):`

```
words1 ← SPLIT(LOWERCASE(text1));
words2 ← SPLIT(LOWERCASE(text2));
word_count1 ← DICTIONARY();
word_count2 ← DICTIONARY();
foreach word in words1 do
    if word ∉ word_count1 then
        | word_count1[word] ← None;
    end
end
foreach word in words2 do
    if word ∉ word_count2 then
        | word_count2[word] ← None;
    end
end
intersection_count ← 0;
union_count ← SIZE(word_count1) + SIZE(word_count2);
foreach word in word_count1 do
    if word ∈ word_count2 then
        | intersection_count ← intersection_count + 1;
    end
end
union_count ← union_count - intersection_count;
if union_count = 0 then
    | return 0.0;
end
return intersection_count / union_count;
```




Figure 2.3(Optimized Pseudocode Algorithm)

2.2.2 Optimized Algorithm: Analysis

	#Operations
1. words1 \leftarrow SPLIT(LOWERCASE(text1))	$C1+N+1$
2. words2 \leftarrow SPLIT(LOWERCASE(text2))	$C2+M+1$
3. word_count1 \leftarrow DICTIONARY()	1
4. word_count2 \leftarrow DICTIONARY()	1
5. FOR EACH word IN words1 DO	N
6. IF word \notin word_count1 THEN	$1*N$
7. word_count1[word] \leftarrow None	$2*N$
8. END IF	
9. END FOR EACH	
10. FOR EACH word IN words2 DO	M
11. IF word \notin word_count2 THEN	$1*M$
12. word_count2[word] \leftarrow None	$2*M$
13. END IF	
14. END FOR EACH	
15. intersection_count \leftarrow 0	1
16. union_count \leftarrow SIZE(word_count1) + SIZE(word_count2)	2
17. FOR EACH word IN word_count1 DO	N
18. IF word \in word_count2 THEN	$1*N$
19. intersection_count \leftarrow intersection_count + 1	$2*N$
20. END IF	
21. END FOR EACH	
22. union_count \leftarrow union_count - intersection_count	1
23. IF union_count = 0 THEN	1
24. RETURN 0.0	1
25. END IF	
26. RETURN intersection_count / union_count	2

Time Complexity Analysis of the Optimized Algorithm:

The optimized algorithm consists of several key operations: splitting the input texts into words, building word dictionaries, calculating the intersection of common words, and determining the union count. Below is a breakdown of the time complexity for each operation:

Splitting the Texts:

The texts are split into words using the SPLIT function and lowercased, which operates in linear time with respect to the number of characters in the texts.

Time Complexity: $O(C1+N+C2+M)$, where $C1$ is the number of characters of text1, $C2$ is the number of characters of text2, N is the number of words of text1, and M is the number of words of text2.

Building the Word Dictionaries:

For each word in words1, we check if it exists in word_count1 and either skip to the next one or add it. This operation takes $O(1)$ time on average due to the use of a dictionary. This process is repeated for words2 to build word_count2.

In the worst case, the number of unique words in both words1 and words2 can be equal to the total number of words.

Time Complexity: $O(N + M)$.

Calculating the Intersection:

We iterate through the unique words in word_count1 (up to N unique words in the worst case) and check if each exists in word_count2. Since lookups in dictionaries are $O(1)$ on average, this results in linear time complexity.

Time Complexity: $O(N)$.

Calculating the Union Count:

The union count is derived from the sizes of the two dictionaries minus intersection count. This step involves simple arithmetic and thus operates in constant time.

Time Complexity: $O(1)$.

Returning the Result:

The final division operation and returning the result also operate in constant time.

Time Complexity: $O(1)$.

Overall Time Complexity:

The overall time complexity of the optimized algorithm is determined by the most significant terms. Therefore, it can be written as: $O(N + M + N + C1 + C2 + 1 + 1)$.

In the end, this simplifies to: $O(N)$.

Space Complexity Analysis of the Optimized Algorithm:

The space complexity of the algorithm reflects the amount of memory used relative to the size of the input. Below is a detailed breakdown of how memory is utilized throughout the algorithm:

words1 and words2 Storage:

The words from text1 and text2 are stored in two lists. If text1 has N words and text2 has M words, the space required is:

Space Complexity: $O(N + M)$.

Word Dictionaries:

The dictionaries word_count1 and word_count2 store all unique words from text1 and text2, respectively. In the worst case, each dictionary may contain all unique words, which can be equal to N and M :

Space Complexity: $O(N + M)$.

Miscellaneous Variables:

Variables like counters and flags require constant memory:

Space Complexity: $O(1)$.

Total Space Complexity:

The total space complexity is $O(N + M)$, as the memory usage is dominated by storing the words and unique words from both texts. This ensures efficient memory usage, even for large inputs.

Algorithm Scenarios:

The time complexity of the optimized Jaccard similarity algorithm varies based on the characteristics of the input texts:

Best Case:

When both texts have very few unique words, the algorithm will quickly count occurrences with minimal checks, resulting in linear complexity: $O(N)$.

Average Case:

Under typical conditions with a moderate number of unique words, the algorithm efficiently uses dictionaries for counting. The complexity remains linear: $O(N)$.

Worst Case:

When both texts contain a maximum number of unique words, the algorithm will still efficiently handle counting through dictionaries. Therefore, the complexity remains linear: $O(N)$.

Summary:

Best Case: $O(N)$

Average Case: $O(N)$

Worst Case: $O(N)$

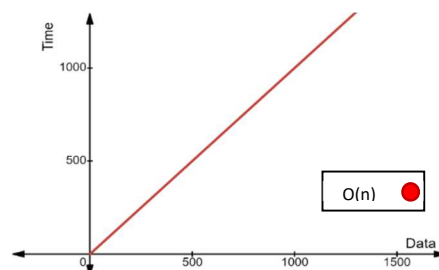


Figure 2.4(Optimized Time complexity graph)

2.2.3 Optimized Algorithm: Example


Input:

- Text 1: "The quick brown fox jumps over the lazy dog"
- Text 2: "The quick blue hare jumps over the lazy dog"


Step 1: Preprocessing

- Convert both texts to lowercase and split them into lists of words:
 - Words from Text 1:
 - ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
 - Words from Text 2:
 - ["the", "quick", "blue", "hare", "jumps", "over", "the", "lazy", "dog"]

Step 2: Create Word Dictionary for Text 1

- Initialize an empty dictionary word_count1.
 - Iterate through each word in Text 1:
 - Add "the" to word_count1 with (None) as a value.
 - Add "quick" to word_count1 with (None) as a value.
 - Add "brown" to word_count1 with (None) as a value.
 - Add "fox" to word_count1 with (None) as a value.
 - Add "jumps" to word_count1 with (None) as a value.
 - Add "over" to word_count1 with (None) as a value.
 - Skip "the" since it has already been added.
 - Add "lazy" to word_count1 with (None) as a value.
 - Add "dog" to word_count1 with (None) as a value.
 - Final Word Count Dictionary for Text 1:
 - {"the": None, "quick": None, "brown": None, "fox": None, "jumps": None, "over": None, "lazy": None, "dog": None}
- 

Step 3: Create Word Dictionary for Text 2

- Initialize an empty dictionary word_count2.
 - Iterate through each word in Text 2:
 - Add "the" to word_count2 with (None) as a value.
 - Add "quick" to word_count2 with (None) as a value.
 - Add "blue" to word_count2 with (None) as a value.
 - Add "hare" to word_count2 with (None) as a value.
 - Add "jumps" to word_count2 with (None) as a value.
 - Add "over" to word_count2 with (None) as a value.
 - Skip "the" since it has already been added.
 - Add "lazy" to word_count2 with (None) as a value.
 - Add "dog" to word_count2 with (None) as a value.
- 

- Final Word Count Dictionary for Text 2:

- {"the": None, "quick": None, "blue": None, "hare": None, "jumps": None, "over": None, "lazy": None, "dog": None}

Step 4: Calculate Initial Union Count

- Calculate the union count as the sum of the sizes of the two word dictionaries:

- Initial Union Count = $\text{size}(\text{word_count1}) + \text{size}(\text{word_count2}) = 8 + 8 = 16$

Step 5: Calculate Intersection Count

- Initialize the intersection count to zero.

- Iterate through each unique word in word_count1:

- "the" → exists in word_count2 (increment intersection count to 1).

- "quick" → exists in word_count2 (increment intersection count to 2).

- "brown" → does not exist in word_count2 (skip).

- "fox" → does not exist in word_count2 (skip).

- "jumps" → exists in word_count2 (increment intersection count to 3).

- "over" → exists in word_count2 (increment intersection count to 4).

- "lazy" → exists in word_count2 (increment intersection count to 5).

- "dog" → exists in word_count2 (increment intersection count to 6).

- Final Intersection Count: 6

Step 6: Update (Recalculate) Union Count

- Final Union Count = Initial Union Count - Final Intersection Count

- Final Union Count = $16 - 6$

- Final Union Count: 10

Step 7: Calculate Jaccard Similarity

- Use the formula to compute the Jaccard similarity:

Jaccard Similarity = $\text{Intersection Count} / \text{Union Count} = 6 / 10 = 0.6$

Output:

- The Jaccard similarity between Text 1 and Text 2 is 0.6 or 60%.

2.3 Theoretical Analysis: Comparison

Naive Algorithm:

Time Complexity:

Worst-case scenario: $O(N^2)$, where N is the number of unique words in the texts.

The algorithm compares every word from one text with every other word, leading to quadratic complexity.

Space Complexity:

$O(N + M)$, where N and M are the number of words in the two texts. It requires storage for the union list and the words in both texts.

Steps:

- Constructs the union list by iterating through each word and ensuring no duplicates.
- Counts intersections by comparing each word from one text with the other.
- Involves multiple nested loops for union and intersection calculations.

Advantages:

Simplicity in implementation.

Disadvantages:

Inefficient for large texts due to quadratic time complexity.

Optimized Algorithm:

Time Complexity:

$O(N)$, where N is the total number of words across both texts. It achieves linear complexity by leveraging dictionaries (hash tables) for faster lookups.

Space Complexity:

$O(N + M)$, similar to the naive approach, but dictionaries are used for storing unique words from both texts.

Steps:

- Uses dictionaries for both texts, which allows for constant-time word lookups.
- Calculates union and intersection more efficiently by counting unique words and directly finding matches between the two sets.
- Avoids nested loops, improving efficiency.

Advantages:

Much more efficient for larger texts due to linear time complexity.

Disadvantages:

Slightly more complex to implement compared to the naive version.

Summary of Comparison:

-Naive Algorithm: Quadratic time complexity makes it slower for large datasets. It is simple but inefficient.

-Optimized Algorithm: Offers significant performance improvement using dictionaries, making it better suited for large inputs.

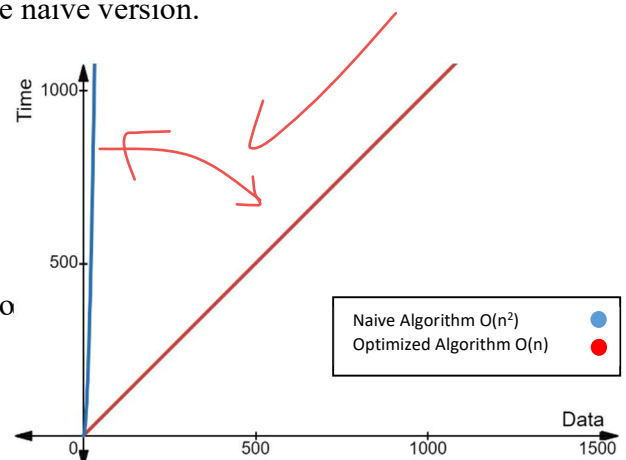


Figure 2.4(The worst case for naive and optimized algorithms graph)

Empirical Analysis

3.1 Naive Algorithm Implementation:

Here is the implementation of the naive algorithm in python. It compares each word from the two texts by iterating through them in a nested loop. While simple, this approach can be slow for large inputs due to its quadratic time complexity. The algorithm builds a union of unique words and calculates the intersection to determine the Jaccard Similarity. This implementation is straightforward and written in Python, utilizing simple loops to process input strings and maintain clarity, though it may not perform optimally for large datasets.

Figure 3.1 (Naive Algorithm in python)

```
# Naive algorithm

def naive_jaccard_similarity(text1, text2):

    # This to split the text into list of words
    words1 = text1.lower().split()
    words2 = text2.lower().split()

    # Union list will contain the union of both texts
    union_list = []
    # To count the number of words in union
    union_count = 0

    # This to remove duplicated words and making union set
    for word in words1:
        flag_comp = True
        for unique_word in union_list:
            if word == unique_word: # If it is already in union then don't put it in the list
                flag_comp = False
                break;
        if flag_comp == True: # If it's not in union list then put it and increase the counter
            union_list.append(word)
            union_count += 1

    # To count the common words in text1 and text2 ignoring the duplicated words
    intersection_count = 0
    for word in union_list:
        for compare_word in words2:
            if word == compare_word:
                intersection_count += 1
                break

    # To count the complete union by finding only words that unique from text2
    for word in words2:
        flag_comp = True
        for unique_word in union_list:
            if word == unique_word: # If it's already in union then skip it
                flag_comp = False
                break;
        if flag_comp == True: # If it's not found in union list then put it in union list and increase
            counter
            union_list.append(word)
            union_count += 1

    if union_count == 0: # To handel run time error in case of union set is empty
        return 0.0
    return intersection_count / union_count # This is the Jaccar Simillirity law

# naive function Finish here -----
```

3.2 Optimized Algorithm Implementation

Below is the implementation of the optimized algorithm. By using dictionaries to track unique words, the algorithm avoids redundant comparisons and significantly improves performance. The optimized approach reduces the time complexity to linear, making it much more efficient for large datasets. This implementation also emphasizes Python's readability, employing data structures that enhance performance while remaining accessible to understand the underlying logic.

Figure 3.2 (Optimized Algorithm in python)

```
# Optimized Jaccard Similarity Algorithm

def optimized_jaccard_similarity(text1, text2):

    # Tokenize the texts and convert to lowercase for case insensitivity
    words1 = text1.lower().split()
    words2 = text2.lower().split()

    # Use dictionaries to count occurrences of each word in both texts
    word_count1 = {} # Dictionary for word counts in text1
    word_count2 = {} # Dictionary for word counts in text2

    # Appends only unique words of text1 into word_count1 {}
    for word in words1:
        if word not in word_count1:
            # Word has not been appended yet, so append it
            word_count1[word] = None

    # Appends only unique words of text2 into word_count2 {}
    for word in words2:
        if word not in word_count2:
            # Word has not been appended yet, so append it
            word_count2[word] = None

    # Initialize intersection and union sizes
    intersection_count = 0 # To store the number of common words between both texts
    union_count = len(word_count1) + len(word_count2) # Initialize union with sum of unique words from
    both texts

    # Calculate the intersection then calculating union
    for word in word_count1: # Loop through words in text1's word counts
        if word in word_count2: # If word is common between both texts
            intersection_count += 1 # Increase intersection count

    # Union count is (sum of unique words from both texts) - intersection_count
    union_count -= intersection_count

    # Handle case where both sets are empty, preventing division by zero
    if union_count == 0:
        return 0.0

    # Calculate and return the Jaccard similarity
    return intersection_count / union_count

#Optimized algorithm ends here-----
```

3.3 Performance Analysis

To evaluate the performance of both the naive and optimized algorithms, we ran a series of tests using different input sizes. The input texts varied in length, and we recorded the time taken by each algorithm to compute the Jaccard similarity. Below are the results:

Figure 3.4(Execution Results)

Sample Size (Words)	Naive (ms)	Optimized (ms)	Results (Jaccard Similarity)
Small - 1K	2.6	1	33.65%
Medium - 40k	2633	5.1	29.27%
Large - 180k	24110	37.7	33.62%

Note: Execution time results may differ from machine to another.

The results clearly demonstrate that the optimized algorithm is significantly faster than the naive algorithm, especially as the input size increases. The optimized algorithm, which leverages dictionary lookups, scales more efficiently compared to the nested loops used in the naive algorithm.

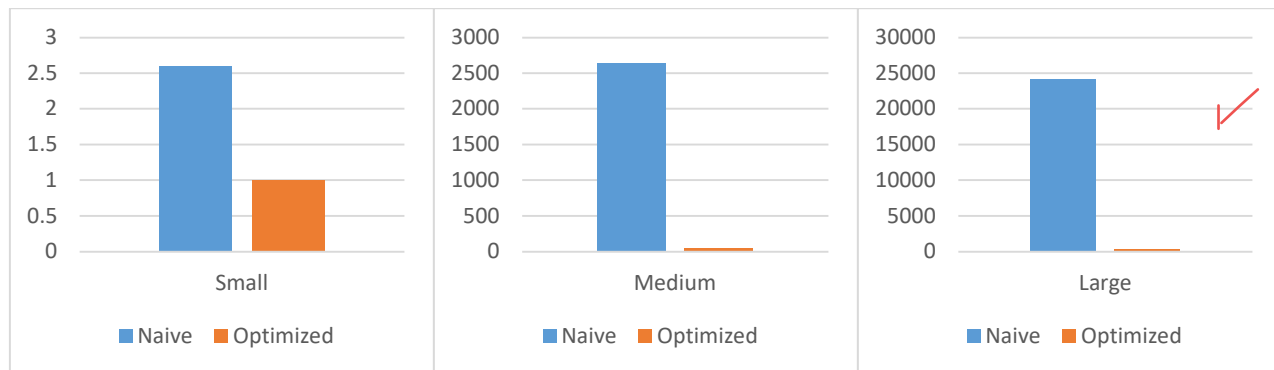


Figure 3.5 (Execution Times charts)

Conclusion

4.1 Comparison

Theoretical analysis suggests that the naive algorithm has a time complexity of $O(N^2)$, while the optimized algorithm has a linear time complexity of $O(N)$. This is confirmed by the empirical data, where the runtime of the naive algorithm increases quadratically as the input size grows, whereas the optimized algorithm increases in a linear fashion.

For example:

At an input size of 1,000 words, the naive takes 2.6 ms, while the optimized takes only 1 ms.

At an input size of 180,000 words, the naive takes 24110 ms, compared to the optimized 37.7 ms.

This performance gap widens as input sizes increase, confirming the theoretical expectations.

The discrepancies between theoretical and empirical results are minimal, and they primarily arise due to hardware limitations and varying system loads during testing. However, both the empirical and theoretical analyses agree that the optimized algorithm is much more efficient for large datasets.

4.2 Final thoughts

In this project, we implemented and analyzed two approaches to calculating the Jaccard similarity between two texts: a naive algorithm and an optimized algorithm. The naive approach, with a time complexity of $O(N^2)$, is simple but inefficient for large inputs. The optimized algorithm, which uses dictionaries for faster lookups, achieves a time complexity of $O(N)$, making it much more suitable for large datasets.

Empirical testing confirmed the theoretical analyses, showing that the optimized algorithm significantly outperforms the naive one as input sizes grow. This project illustrates the importance of algorithmic optimization in real-world applications, where performance can make a significant difference.



5. Ref—
1
2