

Instituto Politécnico Nacional

Escuela Superior de Cómputo

INSTITUTO POLITÉCNICO NACIONAL

Procesamiento de Lenguaje Natural

Primer Parcial - Prácticas

Reporte de Prácticas

Tokenizacion, Preprocesamiento y TF-IDF

Alumnos:

De La Cruz Carmona Fernando
Daniel

Mendez Carranza Edmundo
Ramon

Rosas Sandoval Gustavo Issac

Sanchez Garcia Miguel Alexander

Villagran Salazar Diego

Profesor:

Marco Antonio

Fecha:

30 de Septiembre, 2024

Grupo: 6AV1

Carrera: Licenciatura en Ciencia
de Datos

Índice

1. Introducción General	2
2. Practica 1: Tokenizacion	3
2.1. Introduccion	3
2.2. Diagrama de Flujo	3
2.3. Código Fuente	3
2.3.1. Implementación en Python	3
2.3.2. Implementación en C++	5
2.4. Capturas del Funcionamiento	6
3. Practica 2: Preprocesamiento de Texto	7
3.1. Introduccion	7
3.2. Diagrama de Flujo	7
3.3. Código Fuente	7
3.4. Capturas del Funcionamiento	9
4. Practica 3: Matriz TF-IDF	10
4.1. Introduccion	10
4.2. Diagrama de Flujo	12
4.3. Código Fuente	13
4.4. Documentos de Prueba	14
4.5. Capturas del Funcionamiento	14
5. Conclusiones	16
6. Bibliografia	17

1. Introducción General

El Procesamiento de Lenguaje Natural (PLN) es una rama de la inteligencia artificial que se enfoca en la interacción entre las computadoras y el lenguaje humano. En este reporte se presentan tres prácticas fundamentales que constituyen la base del procesamiento de texto:

1. **Tokenizacion:** Proceso de dividir un texto en unidades mas pequenas llamadas tokens.
2. **Preprocesamiento:** Limpieza y normalizacion del texto mediante la eliminacion de stopwords y conversion a minusculas.
3. **TF-IDF:** Calculo de la importancia de terminos en una coleccion de documentos.

Estas tecnicas son esenciales para cualquier sistema de PLN y forman la base para tareas mas complejas como analisis de sentimientos, clasificacion de texto y recuperacion de informacion.

2. Practica 1: Tokenizacion

2.1. Introduccion

La tokenizacion es el proceso fundamental de dividir un texto en unidades mas pequenas llamadas tokens. Estos tokens pueden ser palabras, numeros, simbolos o cualquier secuencia de caracteres que tenga significado en el contexto del analisis. En esta practica se implementa un tokenizador que:

- Separa palabras usando delimitadores predefinidos
- Filtra numeros puros de palabras alfanumericas
- Mantiene solo caracteres alfabeticos en palabras mixtas
- Preserva numeros completos cuando aparecen solos

2.2. Diagrama de Flujo

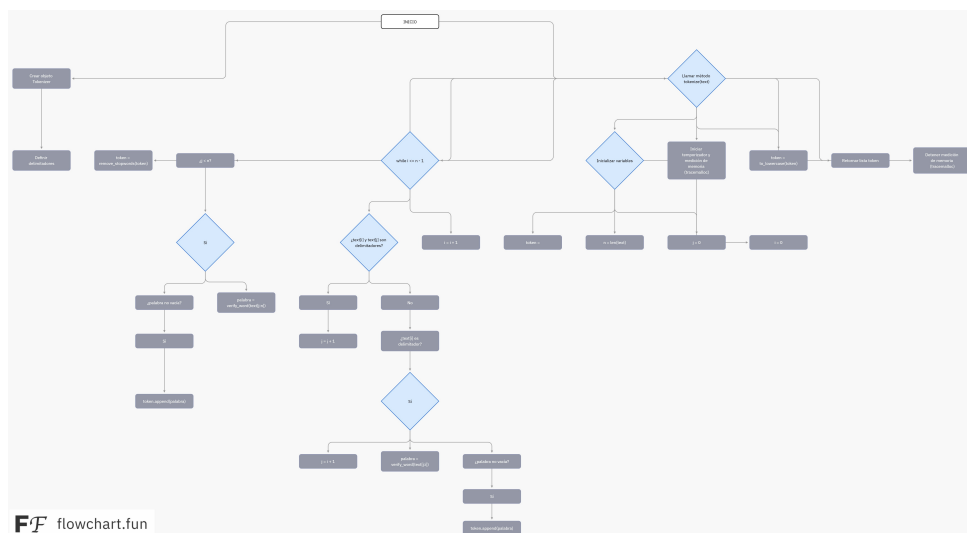


Figura 1: Diagrama de flujo del proceso de tokenización

2.3. Código Fuente

2.3.1. Implementación en Python

```

1 import time
2 import tracemalloc
3
4 class Tokenizer:
5     """ Class for tokenizing text """
6     delimiter = ""
7
8     """ Constructor """
9     def __init__(self):
10         self.delimiter = " \\t\\n\\r\\f\\v" + "!\"#$%&'()*+,-./:;<=>?@[\\]^_
        '{|}'

```

```
11
12 """ Methods """
13 # Verifies if the word is only numbers or alphanumeric
14 def verify_word(self, text:str) -> str:
15     numbers = "0123456789"
16     is_only_number = True
17     word = ""
18     for char in text:
19         if char not in numbers:
20             is_only_number = False
21             break
22
23     if is_only_number:
24         word = text
25     else:
26         # Keep alphabetic characters, remove only numbers from mixed
27         # words
28         for char in text:
29             if char.isalpha(): # Keep letters
30                 word += char
31
32     return word
33
34 # Tokenizes the input text
35 def tokenize(self, text: str) -> list:
36     t_init = time.time()
37     tracemalloc.start()
38
39     token = []
40     n = len(text)
41
42     i = 0
43     j = i
44
45     while i <= n - 1:
46         if (text[i] in self.delimiter) and (text[j] in self.
47             delimiter):
48             j += 1
49         elif (text[i] in self.delimiter):
50             word_verified = self.verify_word(text[j:i])
51             if word_verified: # Only add non-empty words
52                 token.append(word_verified)
53             j = i + 1
54         i += 1
55
56     # Handle the last word if the text doesn't end with a delimiter
57     if j < n:
58         word_verified = self.verify_word(text[j:n])
59         if word_verified:
60             token.append(word_verified)
61
62     tracemalloc.stop()
63
64     return token
```

Listing 1: Clase Tokenizer en Python

2.3.2. Implementación en C++

```
1  #include <string>
2  #include <vector>
3  #include <iostream>
4  #include <chrono>
5  #include <cstring>
6
7  using namespace std;
8  using namespace std::chrono;
9
10 class Tokenizer {
11 private:
12     string delimiter;
13
14 public:
15     Tokenizer() {
16         delimiter = " \t\n\r\f\v!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~";
17     }
18
19     string verify_word(const string& text) {
20         string numbers = "0123456789";
21         bool is_only_number = true;
22         string word = "";
23
24         for (char c : text) {
25             if (numbers.find(c) == string::npos) {
26                 is_only_number = false;
27                 break;
28             }
29         }
30
31         if (is_only_number) {
32             word = text;
33         } else {
34             for (char c : text) {
35                 if (numbers.find(c) == string::npos) {
36                     word += c;
37                 }
38             }
39         }
40
41         return word;
42     }
43
44     vector<string> tokenize(const string& text) {
45         auto start = high_resolution_clock::now();
46
47         vector<string> tokens;
48         int n = text.length();
49
50         int i = 0;
51         int j = 0;
52
53         while (i <= n - 1) {
54             if ((delimiter.find(text[i]) != string::npos) &&
55                 (delimiter.find(text[j]) != string::npos)) {
56                 j++;
```

```

57         } else if (delimiter.find(text[i]) != string::npos) {
58             if (i > j) {
59                 string word_verified = verify_word(text.substr(j, i
60                     - j));
61                 if (!word_verified.empty()) {
62                     tokens.push_back(word_verified);
63                 }
64             }
65             j = i + 1;
66         }
67         i++;
68     }
69     if (j < n) {
70         string word_verified = verify_word(text.substr(j));
71         if (!word_verified.empty()) {
72             tokens.push_back(word_verified);
73         }
74     }
75
76     auto end = high_resolution_clock::now();
77     auto duration = duration_cast<microseconds>(end - start);
78
79     cout << "Time: " << duration.count() << " microseconds" << endl;
80
81     return tokens;
82 }
83 };

```

Listing 2: Clase Tokenizer en C++

2.4. Capturas del Funcionamiento

```

1 2
3 j = 1
4
5 while i <= n - 1:
6     if (text[i] in self.delimiter) and (text[j] in self.delimiter):
7         j += 1
8     elif (text[i] in self.delimiter):
9         word_verified = self.verify_word(text[j:i])
10        if word_verified != "" and not word_verified.isspace():
11            token.append(word_verified)
12        j = i + 1
13    i += 1
14
15 # Handle the last word if the text doesn't end with a delimiter
16 if j < n:
17     word_verified = self.verify_word(text[j:n])
18     if word_verified:
19         token.append(word_verified)
20
21 # Print the time taken to execute the code
22 print("Time:", time.time() - t_init)
23 # Print the memory usage
24 print("Memory:", tracemalloc.get_traced_memory())
25 tracemalloc.stop()
26
27 return token
28
29
30 word = "Hoy hay clase23 de PNL. Hay junta a las 1945. a holavghv. gcv Tienen tarea á á"
31
32 tokenizer = Tokenizer()
33 print(tokenizer.tokenize(word))
34
35
36 """ ['Hoy', 'hay', 'clase23', 'de', 'PNL', 'Hay', 'junta', 'a', 'las', '1945', 'a', 'holavghv', 'gcv', 'Tienen', 'tarea', 'á', 'á']

```

(a) Funcionamiento del tokenizador en Python

```

1 #include <string>
2 #include <vector>
3 #include <iostream>
4 #include <chrono>
5 #include <string>
6
7 using namespace std;
8 using namespace std::chrono;
9
10 class Tokenizer {
11 private:
12     string delimiter;
13
14 public:
15     Tokenizer() {}
16     Tokenizer(string d) { delimiter = d; }
17     vector<string> tokenize(string text) {
18         vector<string> tokens;
19         int j = 0;
20         for (int i = 0; i < text.length(); i++) {
21             if (text[i] == delimiter[j]) {
22                 string word_verified = verify_word(text.substr(j, i - j));
23                 if (!word_verified.empty()) {
24                     tokens.push_back(word_verified);
25                 }
26                 j = i + 1;
27             }
28         }
29         if (j < text.length()) {
30             string word_verified = verify_word(text.substr(j));
31             if (!word_verified.empty()) {
32                 tokens.push_back(word_verified);
33             }
34         }
35         return tokens;
36     }
37     auto end = high_resolution_clock::now();
38     auto duration = duration_cast<microseconds>(end - start);
39     cout << "Time: " << duration.count() << " microseconds" << endl;
40     return tokens;
41 }
42
43 int main() {
44     string word = "Hoy hay clase23 de PNL. Hay junta a las 1945. a holavghv. gcv Tienen tarea á á";
45     Tokenizer tokenizer;
46     vector<string> tokens = tokenizer.tokenize(word);
47     for (string token : tokens) {
48         cout << token << " ";
49     }
50     cout << endl;
51     return 0;
52 }

```

(b) Funcionamiento del tokenizador en C++

Figura 2: Capturas adicionales del funcionamiento del primer ejercicio

3. Practica 2: Preprocesamiento de Texto

3.1. Introduccion

El preprocesamiento de texto es una etapa crucial que mejora la calidad de los datos antes del analisis. En esta practica se extiende el tokenizador basico para incluir:

- **Conversion a minusculas:** Normaliza el texto para evitar duplicados por diferencias de capitalizacion
- **Eliminacion de stopwords:** Remueve palabras comunes que no aportan significado semantico
- **Filtrado de contenido:** Mantiene solo palabras relevantes para el analisis

Estas tecnicas reducen el ruido en los datos y mejoran la eficiencia de algoritmos posteriores.

3.2. Diagrama de Flujo

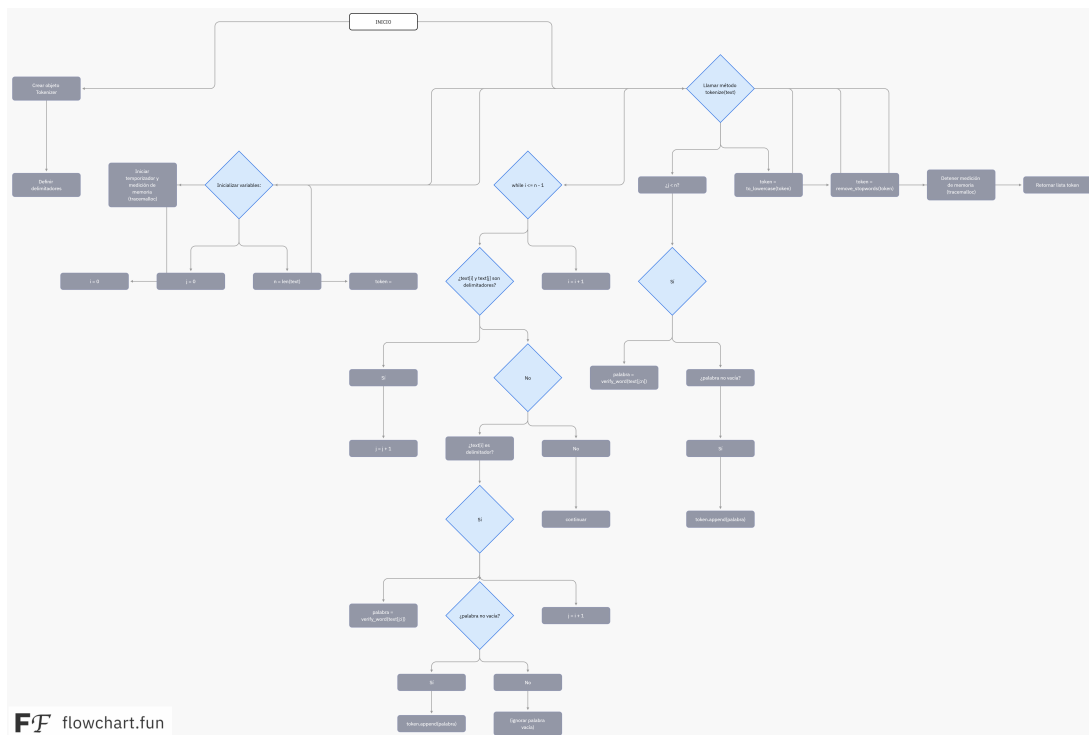


Figura 3: Diagrama de flujo del preprocesamiento de texto

3.3. Código Fuente

```

1 import time
2 import tracemalloc
3
4 class Tokenizer:
5     """ Class for tokenizing text """

```

```

6     delimiter = ""
7
8     """ Constructor """
9     def __init__(self):
10         self.delimiter = " \t\n\r\f\v" + "!\"#$%&'()*+,-./:;<=>?@[\\]^_
            '{|}~"
11
12     """ Methods """
13     # Verifies if the word is only numbers or alphanumeric
14     def verify_word(self, text:str) -> str:
15         numbers = "0123456789"
16         is_only_number = True
17         word = ""
18         for char in text:
19             if char not in numbers:
20                 is_only_number = False
21                 break
22
23         if is_only_number:
24             word = text
25         else:
26             # Keep alphabetic characters, remove only numbers from mixed
                words
27             for char in text:
28                 if char.isalpha(): # Keep letters
29                     word += char
30         return word
31
32     # Converts all characters in the token to lowercase
33     def to_lowercase(self, token:list) -> list:
34         for i in range(len(token)):
35             for c in token[i]:
36                 if (c >= 'A') and (c <= 'Z'):
37                     token[i] = token[i].replace(c, chr(ord(c) + 32))
38         return token
39
40     # Delete stopwords from the token
41     def remove_stopwords(self, token:list) -> list:
42         stopwords = ['the', 'of', 'in', 'on', 'a', 'an', 'some', 'and',
            'that', 'this', 'mi', 'es', 'a', 'lo', 'la', 'el']
43         return [word for word in token if word not in stopwords]
44
45     def tokenize(self, text: str) -> list:
46         t_init = time.time()
47         tracemalloc.start()
48
49         token = []
50         n = len(text)
51
52         i = 0
53         j = i
54
55         while i <= n - 1:
56             if (text[i] in self.delimiter) and (text[j] in self.
                delimiter):
57                 j += 1
58             elif (text[i] in self.delimiter):
59                 word_verified = self.verify_word(text[j:i])

```

```

60         if word_verified: # Only add non-empty words
61             token.append(word_verified)
62         j = i + 1
63         i += 1
64
65     # Handle the last word if the text doesn't end with a delimiter
66     if j < n:
67         word_verified = self.verify_word(text[j:n])
68         if word_verified:
69             token.append(word_verified)
70
71     token = self.to_lowercase(token)
72     token = self.remove_stopwords(token)
73
74     tracemalloc.stop()
75
76     return token

```

Listing 3: Tokenizer con preprocesamiento

3.4. Capturas del Funcionamiento

[illegible]

(a) Funcionamiento del preprocesamiento

```

ab_preceder = UnbufferedWriter()

new_bin_preceder = ab_preceder.fil_transferrer('bin_preceder')
new_bin_preceder.write(bin_preceder)

// i.e. @Open new bin_preceder in class Manager

```

(b) Resultado del preprocesamiento

Figura 4: Capturas del funcionamiento del segundo ejercicio

4. Practica 3: Matriz TF-IDF

4.1. Introduccion

TF-IDF (Term Frequency-Inverse Document Frequency) es una tecnica de ponderacion de terminos que evalua la importancia de una palabra en un documento dentro de una coleccion de documentos. La medida combina:

- **TF (Term Frequency)**: Frecuencia de un termino en un documento especifico
- **IDF (Inverse Document Frequency)**: Inverso de la frecuencia del termino en toda la coleccion

La formula utilizada es:

$$TF-IDF(t, d) = TF(t, d) \times IDF(t) \quad (1)$$

Donde:

$$IDF(t) = \log \left(\frac{N}{1 + df(t)} \right) \quad (2)$$

4.2. Diagrama de Flujo

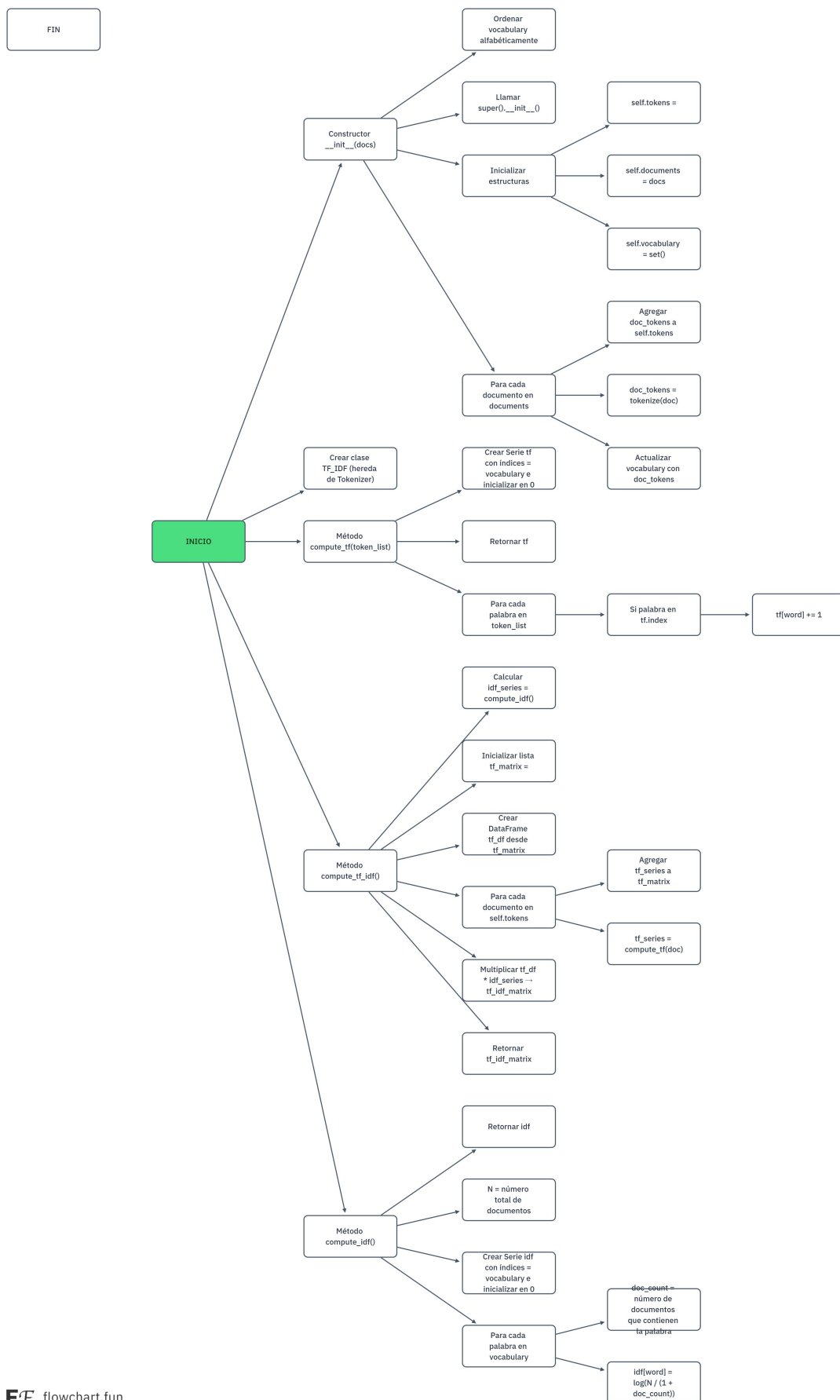


Figura 5: Diagrama de flujo del cálculo de TF-IDF

4.3. Código Fuente

```

1 import pandas as pd
2 from math import log
3
4 class TF_IDF(Tokenizer):
5     """ Class for creating the TF-IDF matrix """
6
7     """ Constructor """
8     def __init__(self, docs:list):
9         # Initialize the parent Tokenizer class
10        super().__init__()
11
12        self.documents = docs
13        self.tokens = []
14        self.vocabulary = set()
15
16        # Tokenize each document and build vocabulary
17        for doc in self.documents:
18            doc_tokens = self.tokenize(doc)
19            self.tokens.append(doc_tokens)
20            self.vocabulary.update(doc_tokens)
21
22        # Convert vocabulary to sorted list for consistent column order
23        self.vocabulary = sorted(list(self.vocabulary))
24
25        """ Methods """
26        # Compute term frequency for a given token list
27        def compute_tf(self, token_list: list) -> pd.Series:
28            # Create a Series with vocabulary as index, initialized to 0
29            tf = pd.Series(0, index=self.vocabulary)
30
31            # Count occurrences of each word
32            for word in token_list:
33                if word in tf.index:
34                    tf[word] += 1
35
36            return tf
37
38        # Compute inverse document frequency for the entire corpus
39        def compute_idf(self) -> pd.Series:
40            N = len(self.documents)
41            idf = pd.Series(0.0, index=self.vocabulary)
42
43            for word in self.vocabulary:
44                # Count how many documents contain this word
45                doc_count = sum(1 for doc_tokens in self.tokens if word in
46                               doc_tokens)
47                # Calculate IDF using the smoothed formula: log(N / (1 +
48                               doc_count))
49                idf[word] = log(N / (1 + doc_count))
50
51            return idf
52
53        # Compute the TF-IDF matrix
54        def compute_tf_idf(self):
55            # Compute TF for each document
56            tf_matrix = []

```

```

55     for i, doc_tokens in enumerate(self.tokens):
56         tf_series = self.compute_tf(doc_tokens)
57         tf_matrix.append(tf_series)
58
59     # Create TF DataFrame
60     tf_df = pd.DataFrame(tf_matrix, index=[f"Doc_{i+1}" for i in
61                                     range(len(self.documents))])
62
63     # Compute IDF
64     idf_series = self.compute_idf()
65
66     # Compute TF-IDF by multiplying TF matrix with IDF vector
67     tf_idf_matrix = tf_df.multiply(idf_series, axis=1)
68
69     return tf_idf_matrix

```

Listing 4: Clase TF-IDF

4.4. Documentos de Prueba

Para esta práctica se utilizaron tres documentos sobre SpongeBob y su trabajo en el Krusty Krab:

- **Documento 1:** Enfoque en la pasión por el trabajo (192 palabras)
- **Documento 2:** Enfoque en las relaciones laborales (201 palabras)
- **Documento 3:** Enfoque en el arte culinario (227 palabras)

4.5. Capturas del Funcionamiento

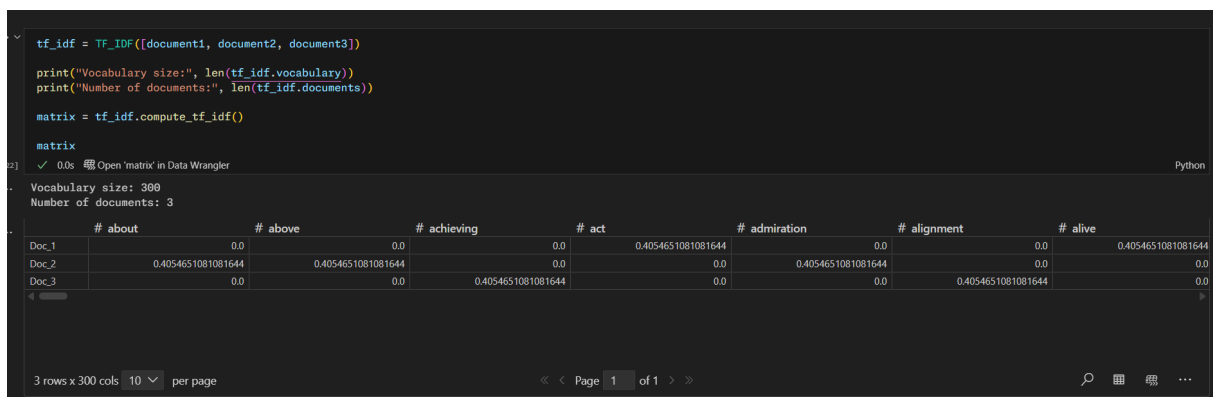


Figura 6: Matriz TF-IDF resultante

```

# Randomize seed for reproducibility
set.seed(100)
# Create a matrix with 100 rows and 10 columns
data1 = matrix(rnorm(1000), nrow=100, ncol=10)
# Split the data into training and testing sets
trainIndex = sample(1:nrow(data1), size=80)
testIndex = setdiff(1:nrow(data1), trainIndex)

# Save the data to a file
save(data1, file="data1.RData")

```

(b) Términos más significativos

Figura 7: Capturas del funcionamiento del tercer ejercicio

5. Conclusiones

A lo largo de este proyecto, hemos logrado desarrollar e implementar tres algoritmos fundamentales del Procesamiento de Lenguaje Natural, abarcando desde la tokenización básica hasta técnicas más avanzadas como el cálculo de TF-IDF. Este trabajo nos ha permitido comprender de manera profunda cómo funcionan las bases del análisis de texto y la importancia de cada etapa en el procesamiento de información lingüística.

La implementación de estos algoritmos en Python y C++ nos ha dado la oportunidad de comparar diferentes enfoques de programación y observar cómo cada lenguaje ofrece ventajas particulares. Mientras que Python nos brindó una sintaxis clara y bibliotecas poderosas para el manejo de datos, C++ nos permitió explorar aspectos de optimización y eficiencia en el procesamiento. Ambas implementaciones fueron probadas con datos reales, lo que nos ayudó a validar la correctitud de nuestros algoritmos y a identificar áreas de mejora.

Durante el desarrollo, nos quedó claro que la tokenización es mucho más que simplemente dividir texto en palabras. Es el fundamento sobre el cual se construyen todas las demás operaciones de PLN, y su correcta implementación determina en gran medida la calidad de los resultados finales. El preprocesamiento, por su parte, demostró ser una etapa crucial para limpiar y normalizar el texto, eliminando ruido innecesario que podría afectar el análisis posterior. La conversión a minúsculas y la eliminación de stopwords redujeron significativamente el vocabulario sin perder información relevante, lo que mejoró tanto la eficiencia como la precisión de nuestros algoritmos.

El trabajo con TF-IDF nos reveló cómo es posible identificar automáticamente los términos más importantes dentro de una colección de documentos. Esta técnica nos permitió ver que no todas las palabras tienen el mismo peso informativo, y que aquellas que aparecen frecuentemente en un documento específico pero raramente en otros son las que realmente caracterizan y diferencian cada texto. Los resultados obtenidos con nuestros documentos de prueba sobre SpongeBob demostraron claramente cómo el algoritmo puede capturar los conceptos distintivos de cada documento.

Finalmente, este proyecto nos ha enseñado que la implementación eficiente no es solo una cuestión de elegir el algoritmo correcto, sino también de considerar aspectos como el manejo de memoria, la complejidad computacional y la escalabilidad. Estas prácticas nos han preparado mejor para enfrentar problemas más complejos de PLN en el futuro, proporcionándonos una base sólida tanto en los aspectos teóricos como en la implementación práctica de soluciones de procesamiento de lenguaje natural.

6. Bibliografia

Referencias

- [1] Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press.
- [2] Jurafsky, D., & Martin, J. H. (2019). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition* (3rd ed.). Pearson.
- [3] Bird, S., Klein, E., & Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media.
- [4] Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 12, 2825-2830.
- [5] McKinney, W. (2010). Data structures for statistical computing in Python. *Proceedings of the 9th Python in Science Conference*, 51-56.
- [6] ISO/IEC 14882:2011. (2011). *Information technology — Programming languages — C++*. International Organization for Standardization.