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

Grupo: 6AV1
Carrera: Licenciatura en Ciencia de Datos

Profesor:
Ortiz Castillo Marco Antonio

Fecha:
30 de Septiembre, 2024

Índice

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

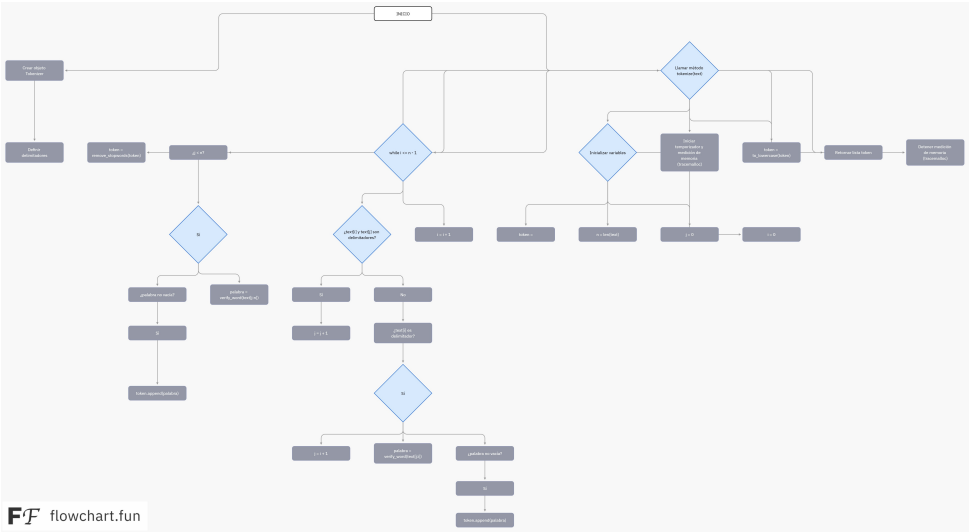


Figura 1: Diagrama de flujo del proceso de tokenización

2.3. Código Fuente

2.3.1. Implementación en Python

```
import time
import tracemalloc

class Tokenizer:
    """ Class for tokenizing text """
    delimiter = ""

    """ Constructor """
    def __init__(self):
        self.delimiter = " \t\n\r\f\v" + "!\"#$%&'()*+,-./:;<=>?@[\\]^_
            `{|}"
```

```
    """ Methods """
    # Verifies if the word is only numbers or alphanumeric
    def verify_word(self, text:str) -> str:
        numbers = "0123456789"
        is_only_number = True
        word = ""
        for char in text:
            if char not in numbers:
                is_only_number = False
                break

        if is_only_number:
            word = text
        else:
            # Keep alphabetic characters, remove only numbers from mixed words
            for char in text:
                if char.isalpha():  # Keep letters
                    word += char
        return word

    # Tokenizes the input text
    def tokenize(self, text: str) -> list:
        t_init = time.time()
        tracemalloc.start()

        token = []
        n = len(text)

        i = 0
        j = i

        while i <= n - 1:
            if (text[i] in self.delimiter) and (text[j] in self.delimiter):
                j += 1
            elif (text[i] in self.delimiter):
                word_verified = self.verify_word(text[j:i])
                if word_verified:  # Only add non-empty words
                    token.append(word_verified)
                j = i + 1
            i += 1

        # Handle the last word if the text doesn't end with a delimiter
        if j < n:
            word_verified = self.verify_word(text[j:n])
            if word_verified:
                token.append(word_verified)

        tracemalloc.stop()

        return token
```

Listing 1: Clase Tokenizer en Python

2.3.2. Implementación en C++

```cpp
#include <string>
#include <vector>
#include <iostream>
#include <chrono>
#include <cstring>

using namespace std;
using namespace std::chrono;

class Tokenizer {
private:
    string delimiter;

public:
    Tokenizer() {
        delimiter = " \t\n\r\f\v!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~";
    }

    string verify_word(const string& text) {
        string numbers = "0123456789";
        bool is_only_number = true;
        string word = "";

        for (char c : text) {
            if (numbers.find(c) == string::npos) {
                is_only_number = false;
                break;
            }
        }

        if (is_only_number) {
            word = text;
        } else {
            for (char c : text) {
                if (numbers.find(c) == string::npos) {
                    word += c;
                }
            }
        }

        return word;
    }

    vector<string> tokenize(const string& text) {
        auto start = high_resolution_clock::now();

        vector<string> tokens;
        int n = text.length();

        int i = 0;
        int j = 0;

        while (i <= n - 1) {
            if ((delimiter.find(text[i]) != string::npos) &&
                (delimiter.find(text[j]) != string::npos)) {
                j++;
```

```
            } else if (delimiter.find(text[i]) != string::npos) {
                if (i > j) {
                    string word_verified = verify_word(text.substr(j, i
                        - j));
                    if (!word_verified.empty()) {
                        tokens.push_back(word_verified);
                    }
                }
                j = i + 1;
            }
            i++;
        }

        if (j < n) {
            string word_verified = verify_word(text.substr(j));
            if (!word_verified.empty()) {
                tokens.push_back(word_verified);
            }
        }

        auto end = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(end - start);

        cout << "Time: " << duration.count() << " microseconds" << endl;

        return tokens;
    }
};
```

Listing 2: Clase Tokenizer en C++

2.4. Capturas del Funcionamiento

(a) Funcionamiento del tokenizador en Python

(b) Funcionamiento del tokenizador en C++

Figura 2: Capturas adicionales del funcionamiento del primer ejercicio

3. Practica 2: Preprocesamiento de Texto, Lectura de PDF y One Hot Encoding

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
- **Lectura de archivos PDF**: Extraccion de texto desde documentos PDF
- **One Hot Encoding**: Creacion de matrices de representacion binaria del vocabulario

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

3.2. Diagrama de Flujo

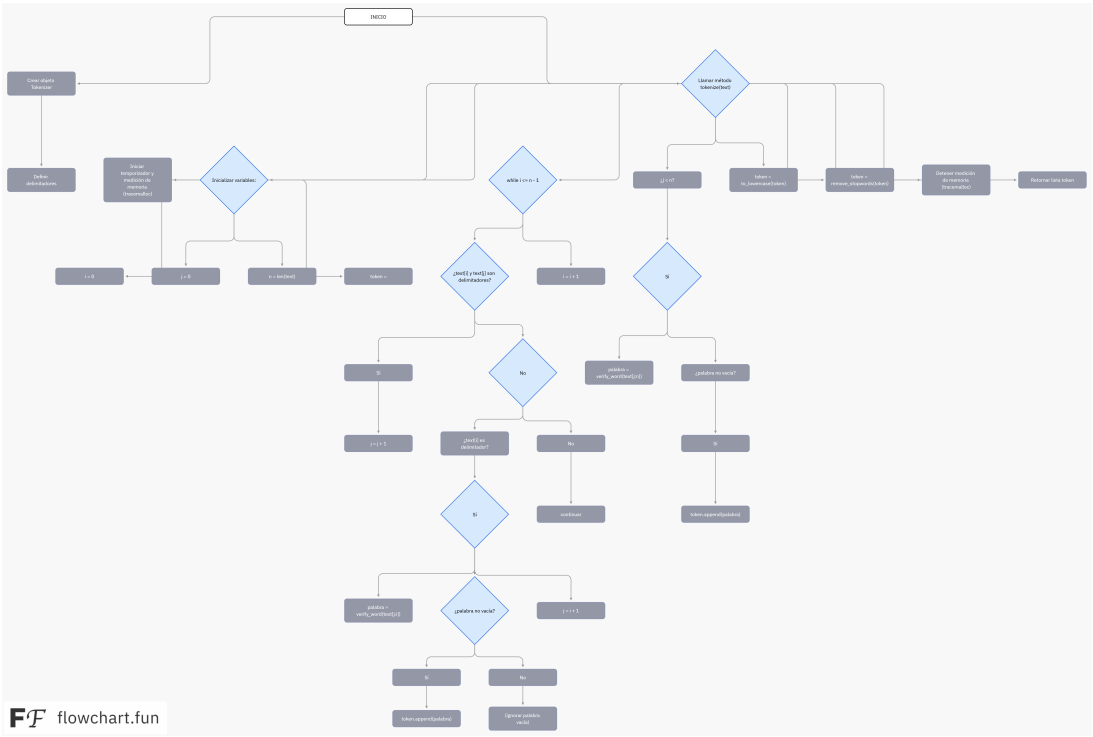


Figura 3: Diagrama de flujo del preprocesamiento de texto

3.3. Código Fuente

```python
import time
import tracemalloc

class Tokenizer:
    """ Class for tokenizing text """
    delimiter = ""

    """ Constructor """
    def __init__(self):
        self.delimiter = " \t\n\r\f\v" + "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"

    """ Methods """
    # Verifies if the word is only numbers or alphanumeric
    def verify_word(self, text:str) -> str:
        numbers = "0123456789"
        is_only_number = True
        word = ""
        for char in text:
            if char not in numbers:
                is_only_number = False
                break

        if is_only_number:
            word = text
        else:
            # Keep alphabetic characters, remove only numbers from mixed words
            for char in text:
                if char.isalpha():  # Keep letters
                    word += char
        return word

    # Converts all characters in the token to lowercase
    def to_lowercase(self, token:list) -> list:
        for i in range(len(token)):
            for c in token[i]:
                if (c >= 'A') and (c <= 'Z'):
                    token[i] = token[i].replace(c, chr(ord(c) + 32))
        return token

    # Delete stopwords from the token
    def remove_stopwords(self, token:list) -> list:
        stopwords = ['the', 'of', 'in', 'on', 'a', 'an', 'some', 'and', 'that', 'this', 'mi', 'es', 'a', 'lo', 'la', 'el']
        return [word for word in token if word not in stopwords]

    def tokenize(self, text: str) -> list:
        t_init = time.time()
        tracemalloc.start()

        token = []
        n = len(text)

        i = 0
        j = i

```

```
        while i <= n - 1:
            if (text[i] in self.delimiter) and (text[j] in self.
               delimiter):
                j += 1
            elif (text[i] in self.delimiter):
                word_verified = self.verify_word(text[j:i])
                if word_verified:  # Only add non-empty words
                    token.append(word_verified)
                j = i + 1
            i += 1

        # Handle the last word if the text doesn't end with a delimiter
        if j < n:
            word_verified = self.verify_word(text[j:n])
            if word_verified:
                token.append(word_verified)

        token = self.to_lowercase(token)
        token = self.remove_stopwords(token)

        tracemalloc.stop()

        return token
```

Listing 3: Tokenizer con preprocesamiento

3.4. Lectura de Archivo PDF y Aplicacion del Tokenizador

3.4.1. Codigo para Lectura de PDF

```
import fitz

# Get the text from a PDF file
doc = fitz.open("el principito.pdf")

# Extract text from each page since third page
text = "\n".join([page.get_text() for page in doc[2:]])

# Print the first 100 characters of the extracted text
print(text[:100])
```

Listing 4: Extracción de texto desde PDF

3.4.2. Tokenizacion del Documento Extraido

```
tokenizer = Tokenizer()

token_text = tokenizer.tokenize(text)
print(len(token_text))
print(token_text[:100])

# Get the unique words from the tokenized text
unique_words = set(token_text)
print(len(unique_words))
print(unique_words)
```

Listing 5: Tokenización del texto extraído

3.5. One Hot Encoding

3.5.1. Implementacion de la Clase OneHotEncoder

```
import pandas as pd

class OneHotEncoder:
    """ Class for One Hot Encoding """
    def __init__(self):
        pass

    def fit_transform(self, token:list) -> dict:
        unique_words = set(token)
        # Order the set alphabetically
        unique_words = sorted(unique_words)
        one_hot_df = pd.DataFrame(0, index=range(len(unique_words)),
                                  columns=list(unique_words))
        for i, word in enumerate(unique_words):
            one_hot_df.at[i, word] = 1
        return one_hot_df
```

Listing 6: Clase OneHotEncoder

3.5.2. Aplicacion del One Hot Encoding

```
oh_encoder = OneHotEncoder()

one_hot_token = oh_encoder.fit_transform(token_text)

one_hot_token
```

Listing 7: Generación de la matriz One Hot

3.6. Descripcion del Proceso de One Hot Encoding

El proceso de One Hot Encoding transforma cada palabra unica del vocabulario en un vector binario donde:

- Cada fila representa una palabra unica del vocabulario
- Cada columna corresponde a una palabra del vocabulario ordenado alfabeticamente
- El valor 1 aparece en la posicion correspondiente a la palabra
- Todos los demas valores son 0

Esta representacion es util para algoritmos de machine learning que requieren entrada numerica, aunque puede ser ineficiente en terminos de memoria para vocabularios grandes.

3.7. Capturas del Funcionamiento

(a) Funcionamiento del preprocesamiento

(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:

$$TF\text{-}IDF(t,d) = TF(t,d) \times IDF(t) \tag{1}$$

Donde:

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

4.2. Diagrama de Flujo

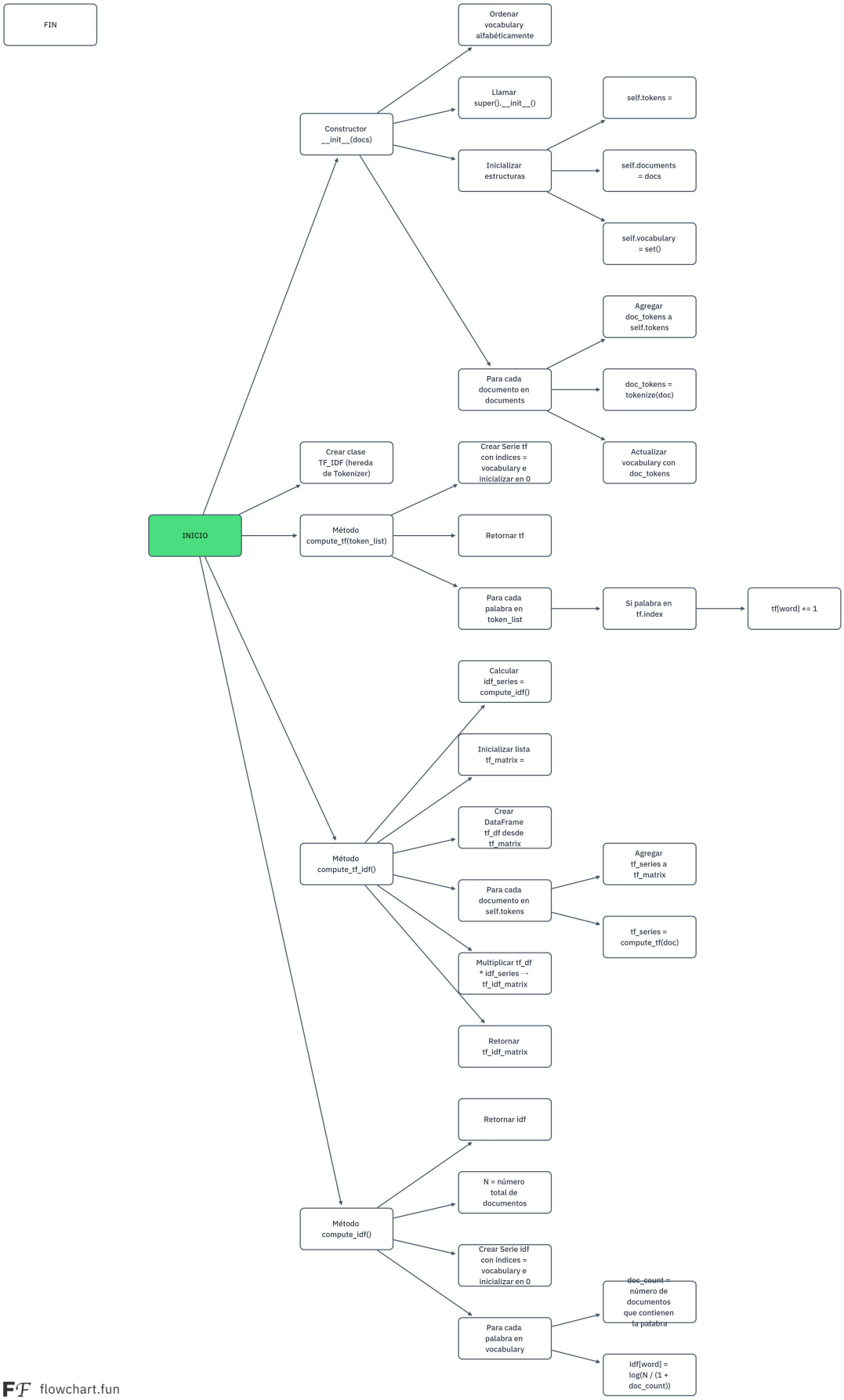


Figura 5: Diagrama de flujo del cálculo de TF-IDF

4.3. Código Fuente

```
import pandas as pd
from math import log

class TF_IDF(Tokenizer):
    """ Class for creating the TF-IDF matrix """

    """ Constructor """
    def __init__(self, docs:list):
        # Initialize the parent Tokenizer class
        super().__init__()

        self.documents = docs
        self.tokens = []
        self.vocabulary = set()

        # Tokenize each document and build vocabulary
        for doc in self.documents:
            doc_tokens = self.tokenize(doc)
            self.tokens.append(doc_tokens)
            self.vocabulary.update(doc_tokens)

        # Convert vocabulary to sorted list for consistent column order
        self.vocabulary = sorted(list(self.vocabulary))

    """ Methods """
    # Compute term frequency for a given token list
    def compute_tf(self, token_list: list) -> pd.Series:
        # Create a Series with vocabulary as index, initialized to 0
        tf = pd.Series(0, index=self.vocabulary)

        # Count occurrences of each word
        for word in token_list:
            if word in tf.index:
                tf[word] += 1

        return tf

    # Compute inverse document frequency for the entire corpus
    def compute_idf(self) -> pd.Series:
        N = len(self.documents)
        idf = pd.Series(0.0, index=self.vocabulary)

        for word in self.vocabulary:
            # Count how many documents contain this word
            doc_count = sum(1 for doc_tokens in self.tokens if word in
                doc_tokens)
            # Calculate IDF using the smoothed formula: log(N / (1 +
                doc_count))
            idf[word] = log(N / (1 + doc_count))

        return idf

    # Compute the TF-IDF matrix
    def compute_tf_idf(self):
        # Compute TF for each document
        tf_matrix = []
```

```
        for i, doc_tokens in enumerate(self.tokens):
            tf_series = self.compute_tf(doc_tokens)
            tf_matrix.append(tf_series)

        # Create TF DataFrame
        tf_df = pd.DataFrame(tf_matrix, index=[f"Doc_{i+1}" for i in
            range(len(self.documents))])

        # Compute IDF
        idf_series = self.compute_idf()

        # Compute TF-IDF by multiplying TF matrix with IDF vector
        tf_idf_matrix = tf_df.multiply(idf_series, axis=1)

        return tf_idf_matrix
```

Listing 8: 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

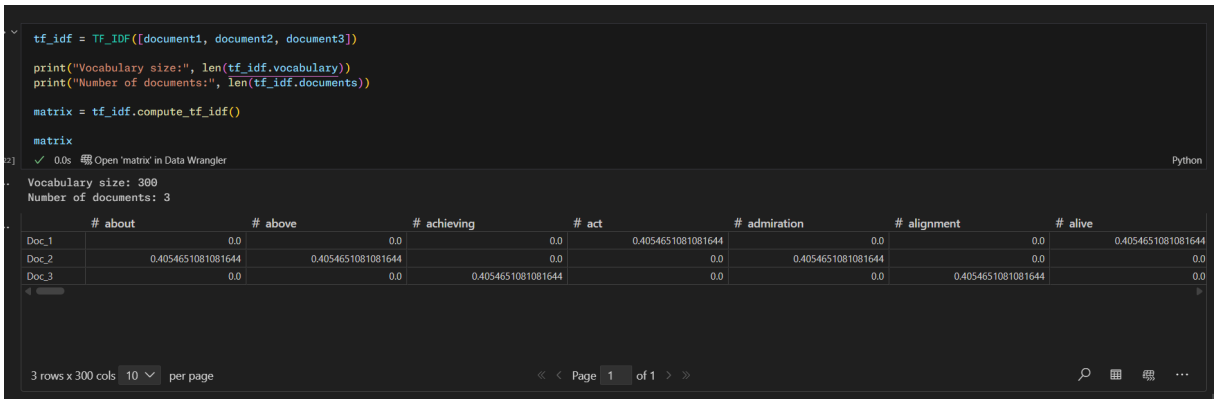
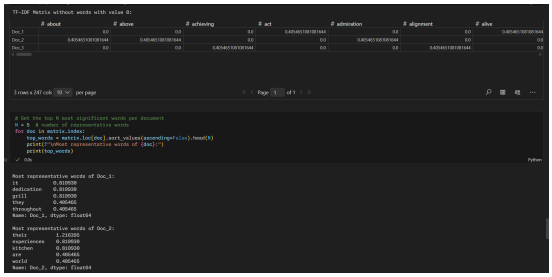
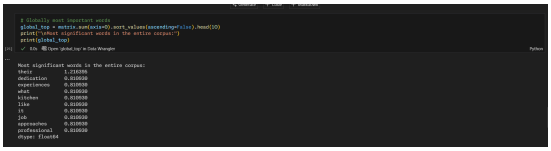


Figura 6: Matriz TF-IDF resultante



(a) Análisis de términos representativos



(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.