

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Procesamiento de Lenguaje Natural

Primer Parcial - Prácticas

Reporte de Prácticas

Tokenización, Preprocesamiento y TF-IDF

Alumnos:

Sánchez García Miguel Alexander

Villagran Salazar Diego

Grupo: 6AV1

Carrera: Licenciatura en Ciencia
de Datos

Profesor:

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. **Tokenización:** Proceso de dividir un texto en unidades más pequeñas llamadas tokens.
2. **Preprocesamiento:** Limpieza y normalización del texto mediante la eliminación de stopwords y conversión a minúsculas.
3. **TF-IDF:** Cálculo de la importancia de términos en una colección de documentos.

Estas técnicas son esenciales para cualquier sistema de PLN y forman la base para tareas más complejas como análisis de sentimientos, clasificación de texto y recuperación de información.

2. Práctica 1: Tokenización

2.1. Introducción

La tokenización es el proceso fundamental de dividir un texto en unidades más pequeñas llamadas tokens. Estos tokens pueden ser palabras, números, símbolos o cualquier secuencia de caracteres que tenga significado en el contexto del análisis. En esta práctica se implementa un tokenizador que:

- Separa palabras usando delimitadores predefinidos
- Filtra números puros de palabras alfanuméricas
- Mantiene solo caracteres alfabéticos en palabras mixtas
- Preserva números 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
13     """ Methods """
14     # Verifies if the word is only numbers or alphanumeric
15     def verify_word(self, text:str) -> str:
16         numbers = "0123456789"
17         is_only_number = True
18         word = ""
19         for char in text:
20             if char not in numbers:
21                 is_only_number = False
22                 break
23
24         if is_only_number:
25             word = text
26         else:
27             # Keep alphabetic characters, remove only numbers from mixed
28             words
```

```

27         for char in text:
28             if char.isalpha(): # Keep letters
29                 word += char
30         return word
31
32     # Tokenizes the input text
33     def tokenize(self, text: str) -> list:
34         t_init = time.time()
35         tracemalloc.start()
36
37         token = []
38         n = len(text)
39
40         i = 0
41         j = i
42
43         while i <= n - 1:
44             if (text[i] in self.delimiter) and (text[j] in self.
45                 delimiter):
46                 j += 1
47             elif (text[i] in self.delimiter):
48                 word_verified = self.verify_word(text[j:i])
49                 if word_verified: # Only add non-empty words
50                     token.append(word_verified)
51                 j = i + 1
52             i += 1
53
54         # Handle the last word if the text doesn't end with a delimiter
55         if j < n:
56             word_verified = self.verify_word(text[j:n])
57             if word_verified:
58                 token.append(word_verified)
59
60         tracemalloc.stop()
61
62         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                     j = i + 1;
65                 }
66                 i++;
67             }
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     }
```

```
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

(a) Ejecución en Python

(b) Ejecución en C++

Figura 2: Resultados de la tokenización

Figura 3: Proceso de compilación del tokenizador en C++

3. Práctica 2: Preprocesamiento de Texto

3.1. Introducción

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

- **Conversión a minúsculas:** Normaliza el texto para evitar duplicados por diferencias de capitalización
- **Eliminación de stopwords:** Remueve palabras comunes que no aportan significado semántico
- **Filtrado de contenido:** Mantiene solo palabras relevantes para el análisis

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

3.2. Diagrama de Flujo

Figura 4: Diagrama de flujo del preprocesamiento de texto

3.3. Código Fuente

```
1 class Tokenizer:
2     """ Class for tokenizing text """
3     delimiter = ""
4
5     """ Constructor """
6     def __init__(self):
7         self.delimiter = " \t\n\r\f\v" + "!\"#$%&'()*+,-./:;<=>?@[\\]^_
8             '{|}'"
9
10    """ Methods """
11    def verify_word(self, text:str) -> str:
12        numbers = "0123456789"
13        is_only_number = True
14        word = ""
15        for char in text:
16            if char not in numbers:
17                is_only_number = False
18                break
19
20        if is_only_number:
21            word = text
22        else:
23            for char in text:
24                if char.isalpha():
25                    word += char
26
27        # Converts all characters in the token to lowercase
```



```

28     def to_lowercase(self, token:list) -> list:
29         for i in range(len(token)):
30             for c in token[i]:
31                 if (c >= 'A') and (c <= 'Z'):
32                     token[i] = token[i].replace(c, chr(ord(c) + 32))
33         return token
34
35     # Delete stopwords from the token
36     def remove_stopwords(self, token:list) -> list:
37         stopwords = ['the', 'of', 'in', 'on', 'a', 'an', 'some', 'and',
38                     'that', 'this']
39         return [word for word in token if word not in stopwords]
40
41     def tokenize(self, text: str) -> list:
42         t_init = time.time()
43         tracemalloc.start()
44
45         token = []
46         n = len(text)
47
48         i = 0
49         j = i
50
51         while i <= n - 1:
52             if (text[i] in self.delimiter) and (text[j] in self.
53                 delimiter):
54                 j += 1
55             elif (text[i] in self.delimiter):
56                 word_verified = self.verify_word(text[j:i])
57                 if word_verified:
58                     token.append(word_verified)
59                 j = i + 1
60             i += 1
61
62         if j < n:
63             word_verified = self.verify_word(text[j:n])
64             if word_verified:
65                 token.append(word_verified)
66
67         token = self.to_lowercase(token)
68         token = self.remove_stopwords(token)
69
70         tracemalloc.stop()
71
72         return token

```

Listing 3: Tokenizer con preprocesamiento

3.4. Capturas del Funcionamiento

(a) Texto original

(b) Texto procesado

Figura 5: Comparación antes y después del preprocesamiento

Figura 6: Ejecución del preprocesamiento en Jupyter Notebook

4. Práctica 3: Matriz TF-IDF

4.1. Introducción

TF-IDF (Term Frequency-Inverse Document Frequency) es una técnica de ponderación de términos que evalúa la importancia de una palabra en un documento dentro de una colección de documentos. La medida combina:

- **TF (Term Frequency)**: Frecuencia de un término en un documento específico
- **IDF (Inverse Document Frequency)**: Inverso de la frecuencia del término en toda la colección

La fórmula utilizada es:

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

Donde:

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

4.2. Diagrama de Flujo

Figura 7: 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 = []
57     for i, doc_tokens in enumerate(self.tokens):
58         tf_series = self.compute_tf(doc_tokens)
59         tf_matrix.append(tf_series)
60
61     # Create TF DataFrame
62     tf_df = pd.DataFrame(tf_matrix, index=[f"Doc_{i+1}" for i in
63                                         range(len(self.documents))])
64
65     # Compute IDF
66     idf_series = self.compute_idf()
67
68     # Compute TF-IDF by multiplying TF matrix with IDF vector
69     tf_idf_matrix = tf_df.multiply(idf_series, axis=1)
70
71     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 8: Matriz TF-IDF resultante

(a) Vocabulario generado (300 términos)

(b) Estadísticas de la matriz

Figura 9: Análisis del vocabulario y estadísticas TF-IDF

Figura 10: Ejecución completa del algoritmo TF-IDF

5. Análisis de Resultados

5.1. Comparación de Rendimiento

Métrica	Tokenización	Preprocesamiento	TF-IDF
Tiempo de ejecución	< 1 ms	< 2 ms	~ 50 ms
Memoria utilizada	Baja	Baja	Media
Complejidad	$O(n)$	$O(n)$	$O(n \times m)$

Cuadro 1: Comparación de rendimiento entre las tres prácticas

5.2. Efectividad del Preprocesamiento

El preprocesamiento demostró ser efectivo al:

- Reducir el vocabulario en aproximadamente 15 %
- Normalizar variaciones de capitalización
- Eliminar palabras sin valor semántico
- Mejorar la calidad de la matriz TF-IDF

5.3. Calidad de la Matriz TF-IDF

La matriz TF-IDF generada mostró:

- Vocabulario de 300 términos únicos
- Distribución adecuada de pesos
- Identificación correcta de términos distintivos por documento
- Valores coherentes con la teoría TF-IDF

6. Conclusiones

6.1. Logros Obtenidos

1. **Implementación exitosa:** Se desarrollaron tres algoritmos fundamentales de PLN con implementaciones eficientes en Python y C++.
2. **Comprensión teórica:** Se adquirió un entendimiento profundo de los conceptos de tokenización, preprocesamiento y TF-IDF.
3. **Aplicación práctica:** Los algoritmos fueron probados con datos reales y mostraron resultados coherentes.
4. **Optimización:** Se implementaron mejoras de rendimiento y manejo eficiente de memoria.

6.2. Lecciones Aprendidas

- La tokenización es la base fundamental de cualquier sistema de PLN
- El preprocesamiento mejora significativamente la calidad de los resultados
- TF-IDF es una técnica poderosa para identificar términos relevantes
- La implementación eficiente es crucial para el procesamiento de grandes volúmenes de texto

6.3. Trabajo Futuro

1. Implementar técnicas avanzadas de tokenización (subword tokenization)
2. Expandir la lista de stopwords para español
3. Explorar variantes de TF-IDF (TF-IDF normalizado, BM25)
4. Desarrollar interfaces gráficas para las herramientas
5. Optimizar para procesamiento paralelo

7. Comandos de Compilación y Ejecución

7.1. Para C++

```
1 # Compilaci n
2 g++ -o tokenizer tokenizer.cpp -std=c++11
3
4 # Ejecuci n
5 ./tokenizer
6
7 # Con optimizaci n
8 g++ -O2 -o tokenizer tokenizer.cpp -std=c++11
```

Listing 5: Compilación y ejecución del tokenizador en C++

7.2. Para Python

```
1 # Iniciar Jupyter Notebook
2 jupyter notebook
3
4 # Ejecutar directamente
5 python tokenizer.py
6
7 # Con medici n de tiempo
8 time python tokenizer.py
```

Listing 6: Ejecución de los notebooks de Python

8. Bibliografía

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.