

Desarrollo de un Tokenizador Mínimo para la Analítica de Voz en Interacciones Telefónicas en Español

Elías Cristaldo¹

¹ Ingeniería en Informática / Facultad Politécnica / Universidad Nacional de Asunción
Campus, San Lorenzo – Paraguay, Teléfono (+595-21) 588 7000

27 de Junio del 2024

Abstract. Este informe propone una solución para la implementación de los analizadores de voz. Utilizando conocimientos fundamentales del análisis léxico, se construye un tokenizador mínimo que funciona como una herramienta de analítica de voz. Este tokenizador se encargará de identificar los lexemas presentes en un texto que representa la interacción de una llamada telefónica y asignará las etiquetas correspondientes. La solución se desarrolla en tres etapas principales: la primera etapa corresponde al análisis del personal de atención. En esta fase, el texto correspondiente al personal de atención es analizado, y como resultado, se obtiene una puntuación relativa al personal. Como segunda etapa tenemos el análisis del cliente, de vuelta se analiza el texto ingresado, pero en este caso realtiva al cliente, obteniendo así una puntuación que refleja el nivel relativo de satisfacción del cliente respecto a la solución de sus necesidades. Como tercera y última etapa tenemos el análisis global de la llamada, en esta etapa se realiza un análisis combinando las puntuaciones obtenidas en las etapas anteriores, ofreciendo una evaluación integral de la llamada. El programa propuesto demuestra un buen desempeño en el análisis léxico del texto recibido como entrada. Además, puede ser ampliado para mejorar su capacidad de detección de patrones y, al ser combinado con otros algoritmos, alcanzar niveles óptimos de rendimiento.

Keywords: tokenizador mínimo, analítica de voz, análisis léxico, interacción telefónica, satisfacción del cliente, desempeño del personal, token.

1. INTRODUCCIÓN

En el ámbito de la analítica de conversaciones, la capacidad de procesar y analizar grandes volúmenes de interacciones telefónicas se ha vuelto una necesidad crítica para muchas organizaciones. Estas interacciones contienen una gran cantidad de información valiosa que puede ser utilizada para mejorar la calidad del servicio, entender mejor las necesidades de los clientes y optimizar procesos internos.

Para abordar esta necesidad, presentamos el desarrollo de un tokenizador mínimo, conocido como MNLPTK (Minimal Natural Language Processing Tokenizer). Este sistema está diseñado para actuar como una solución de analítica de voz (speech analytics), con el objetivo de identificar y pro-

cesar palabras (lexemas) en textos en idioma español que son el resultado de conversaciones telefónicas.

El **MNLPTK** se enfoca en la identificación precisa de lexemas en los textos de entrada, lo cual es un paso fundamental en el procesamiento del lenguaje natural (NLP). Una vez identificados los lexemas, el sistema los procesará para generar una ponderación sobre la llamada en general. Esta evaluación se basará en varios criterios que, en cierta medida, reflejan la calidad y el contenido de la interacción.

2. OBJETIVO, ALCANCE Y MEJORAS

El objetivo de este proyecto es implementar un sistema de **Speech Analytics** que aproveche los conocimientos adquiridos en el análisis léxico de un compilador. Este sistema se diseñará para procesar y analizar datos de audio convertidos a texto, extrayendo información valiosa para la organización.

Aplicando principios de análisis léxico, se busca desarrollar un programa capaz de identificar patrones en las conversaciones y asignar puntuaciones coherentes a las llamadas. Este enfoque permitirá obtener un análisis detallado de la calidad del servicio y la atención al cliente, proporcionando información que pueda ayudar a mejorar continuamente el servicio ofrecido.

En cuanto al alcance del proyecto, se espera que el análisis se realice únicamente a nivel léxico. No se considerarán aspectos sintácticos ni semánticos, ya que el objetivo es desarrollar un tokenizador mínimo.

El programa podría potenciar su capacidad de aprendizaje mediante la implementación de enfoques más avanzados. Actualmente, no está diseñado para reconocer variantes de género y número en las palabras de entrada. Sin embargo, esta funcionalidad podría integrarse en futuras actualizaciones para mejorar la precisión y utilidad del análisis léxico. Además, es posible mejorar la ponderación de las palabras; actualmente, todas reciben el mismo peso, siendo asignadas como 1, -1 o 0 según su tipo.

3. METODOLOGÍA

Para la aplicación del **MNLPTK** se establecieron los siguientes puntos.

3.1. TOKENS DEFINIDOS PARA EL LENGUAJE DE ENTRADA

El MNLPTK contará con los siguientes tokens: **EXP_MALA**, **EXP_NEUTRA**, **EXP_BUENA** que irán destinados para el análisis de la interacción del cliente y **ATC_MALA**, **ATC_NEUTRA**, **ATC_BUENA** que corresponderán a la interacción del personal de atención al cliente.

3.2. DEFINICIÓN DE LA HERRAMIENTA A UTILIZAR PARA LA DEFINICIÓN DE LOS PATRONES

Para la representación de los patrones se opta por utilizar el algoritmo de simulación de un **AFD** [1] para obtener los lexemas. En la Figura 1 podemos observar una categorización de los tokens por colores de acuerdo si es una palabra buena, neutra o mala. Esto último para entender a través de un ejemplo, el funcionamiento del algoritmo propuesto.



Figura 1: Tokens.

A continuación, en la Figura 2 se muestra a través de un ejemplo, la estructura que se utilizará en la implementación del AFD. En esta representación, cada nodo que corresponda a un estado final estará asociado una categoría específica. Es importante destacar que habrán dos estructuras similares: una para la atención al cliente, que manejará los tokens ATC_MALA, ATC_NEUTRA y ATC_BUENA. Otra otra para la experiencia del cliente, que incluirá los tokens EXP_MALA, EXP_NEUTRA y EXP_BUENA.

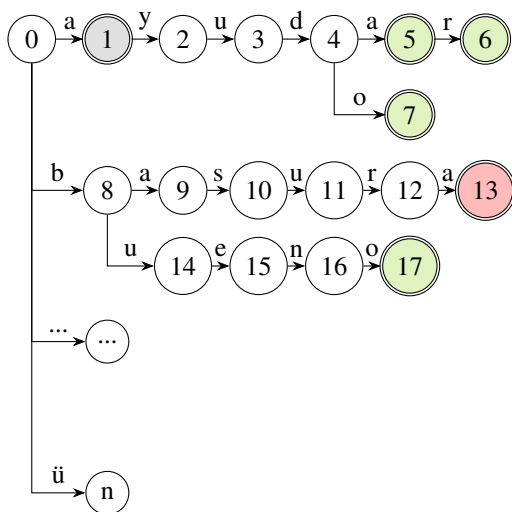


Figura 2: Grafo dirigido con tres nodos.

La idea es que todas las letras del abecedario tengan un único estado inicial. Esto permitirá que al recorrer la es-

tructura, solo se necesite una función, la función **mover**. Esta función tomará un carácter de entrada y, en función del nodo en el que se encuentre, se moverá a un nuevo nodo.

3.3. ESTRUCTURA DE LOS NODOS DEL AFD

Para la construcción del AFD, se definió la estructura **Nodo** junto con sus atributos correspondientes. En la Figura 3 podremos apreciar con más detalles la mencionada estructura.

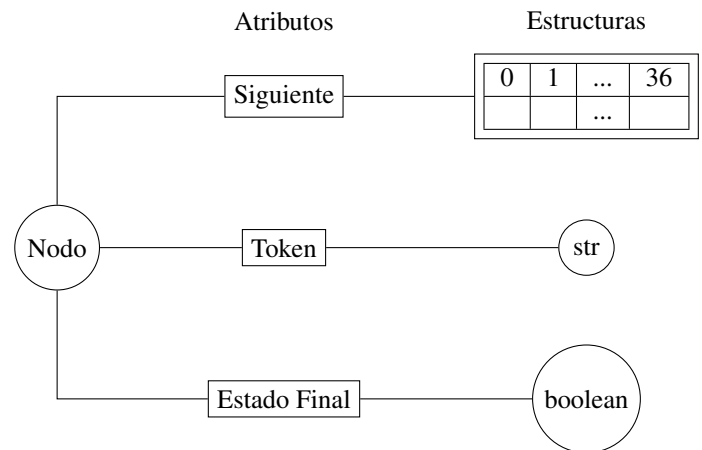


Figura 3: Modelo de un Nodo en el AFD.

El atributo **Siguiente** es un array con 37 elementos, donde cada celda contiene vacío o una referencia a otro nodo. El atributo **Token** es una cadena de texto de tipo string que describe el token al que pertenece el nodo. Por último, el atributo **Estado Final** es un valor booleano que indica si el nodo es un estado final o no. En la siguiente sección, se presentará la función hash que mapea los valores del array **Siguiente**.

3.4. FUNCIÓN HASH

En la sección anterior se presentó la estructura de un nodo, destacando que el atributo **Siguiente** es un array de 37 elementos. ¿Por qué se eligió esta estructura?. La decisión se basa en la necesidad de un acceso más rápido a los datos, donde cada posición del array representa un carácter del alfabeto, como se muestra en la Figura 4.

a	b	...	ñ	...	z	...	á	...	ú	ä	...	ü
0	1	...	14	...	26	...	27	...	31	32	...	36

Figura 4: Función hash para el array de 37 elementos.

Se ha investigado que el diccionario de la RAE contiene aproximadamente 93.000 entradas [2]. Se calculó que cada nodo ocupa aproximadamente 464 bytes. Con una media de 6 letras por palabras. Por lo tanto se tiene que

N : Cantidad aproximada de nodos

C : Capacidad requerida

$$\begin{aligned} N &= 93.000 \text{ [palabras]} * 6 \left[\frac{\text{letras}}{\text{palabras}} \right] \\ &= 558.000 \text{ [letras]} \end{aligned}$$

como cada letra representa un nodo, entonces...

$$\begin{aligned} C &= 558.000 \text{ [nodo]} * 464 \left[\frac{\text{bytes}}{\text{nodo}} \right] \\ &= 258.912.000 \text{ [bytes]} \end{aligned}$$

pasando a MB tenemos que ...

$$\begin{aligned} 258.912.000 \text{ [bytes]} &\equiv \frac{258.912.000}{2^{20}} \\ &\approx 258,912 \text{ [MB]} \end{aligned}$$

por lo tanto, la estructura propuesta es viable de implementar, ya que no requiere un espacio considerable. Aunque no se han tenido en cuenta las conjugaciones, hemos obtenido una aproximación que nos servirá como una cota.

En términos de velocidad, se realizaron pruebas utilizando primero una lista y posteriormente la función de hash presentada anteriormente. Obtuvimos los siguientes resultados:

$$\begin{aligned} t_{lista} &= 0,00008580 \text{ [segundos]} \\ t_{hash} &= 0,00005990 \text{ [segundos]} \end{aligned}$$

$$\frac{t_{hash}}{t_{lista}} * 100 \% \approx 69,81 \%$$

por lo tanto, podemos concluir que al aplicar la función hash, la estrategia es aproximadamente un 30 % más rápida. Esto implica que con un caso de 10.000.000 de accesos usando una estructura de lista, experimentaríamos un retraso de 858 segundos, equivalentes a aproximadamente 14 minutos y 18 segundos. En contraste, al aplicar hash, el retraso se reduce a aproximadamente 599 segundos, equivalente a aproximadamente 9 minutos y 59 segundos.

Es posible mejorar esto; el enfoque que utilizamos aquí podría no ser el más óptimo, por lo que hay margen para mejoras.

3.5. CONSIDERACIONES PARA LA PUNTUACIÓN

Para obtener las evaluaciones, se tuvieron en cuenta las siguientes consideraciones:

– $total$: Cantidad de palabras reconocidas en la entrada.

– $total_{buenas}$: Cantidad de palabras buenas encontradas.

Si se encontraron un saludo y una despedida, cada uno suma un punto en este apartado.

– $total_{malas}$: Cantidad de palabras malas encontradas.

Si no se encontraron un saludo y una despedida, cada uno suma un punto en este apartado.

– $total_{malas}$: Cantidad de palabras malas encontradas.

$$\begin{aligned} buenas_{norm} &= \frac{total_{buenas}}{total} \\ malas_{norm} &= \frac{total_{malas}}{total} \\ balance &= buenas_{norm} - malas_{norm}, \end{aligned}$$

teniendo en cuenta lo anterior la puntuación final estará dada por $1 + p(x)$, siendo $p(x)$ definida por,

$$p(x) = \begin{cases} 0 & \text{si } balance \leq 0, \\ balance * 4 & \text{si } balance > 0 \end{cases}$$

con esto, obtendremos una puntuación final que oscilará entre 1 y 5.

4. CÓDIGO FUENTE

A continuación, se presentan las partes más importantes del código fuente en Python:

```
1 class Nodo:
2     """
3     Clase que representa un nodo de un
4     DFA (Deterministic Finite
5     Automaton) que permita
6     identificar palabras en un
7     texto.
8     Atributos:
9     siguiente([]): arreglo de 37
10     elementos, que representa
11     las posibles transiciones
12     del nodo.
13     token(str): token al que
14     pertenecen las letras que
15     forman la palabra que
16     llega al nodo en caso de
17     ser un estado final.
18     estado_final(bool): indica si
19     el nodo es un estado final
20     o no.
21     Ejemplo:
22     siguiente = [Nodo1, Nodo2,
23     Nodo3, ..., Nodo37]
24     Nodo1: transición con la
25     letra 'a'
26     Nodo2: transición con la
27     letra 'b'
28     ...
```

```

13         Nodo37: transición con la
14             letra 'ü'
15         token = 'ATC_BUENA'
16         estado_final = True
17     """
18     def __init__(self):
19         self.siguiente = np.empty(37,
20             dtype=object)
21         self.token = ''
22         self.estado_final = False
23
24     def mover(self, caracter):
25         """
26         Metodo que permite mover de un
27             nodo a otro, dependiendo
28             de la letra que se recibe
29             como parámetro.
30         Parametros:
31             caracter(str): letra que
32                 se recibe para mover
33                 al siguiente nodo.
34         Returns:
35             nodo_siguiente(Nodo): nodo
36                 al que se movió.
37         """
38         print("Moviendo al siguiente
39             nodo con {}...".format(
40                 caracter))
41
42         nodo_siguiente = None
43         hash_alfabeto = HashFunction()
44             .get_funcion_hash()
45
46         if self.siguiente[
47             hash_alfabeto[caracter]]:
48             nodo_siguiente = self.
49                 siguiente[
50                     hash_alfabeto[caracter]
51                 ]
52         else:
53             nodo_siguiente = Nodo()
54             self.siguiente[
55                 hash_alfabeto[caracter]
56             ] = nodo_siguiente
57
58         print("Transición completada.")
59         return nodo_siguiente

```

Listing 1: Clase Nodo

```

1 class HashFunction:
2     """
3     Clase que implementa el patrón
4         Singleton para crear una funci
5         ón hash que asigna un valor a
6         cada letra del alfabeto espa
7         ñol.
8     Ejemplo:
9         hash_function = HashFunction()
10             .get_funcion_hash()
11     """
12     _instance = None # Variable de
13         clase para almacenar la
14         instancia única

```

```

8     _funcion_hash = None # Variable
9         de clase para almacenar la
10         función hash
11
12     def __new__(cls, *args, **kwargs):
13         if cls._instance is None:
14             cls._instance = super().
15                 __new__(cls)
16         return cls._instance
17
18     def __init__(self):
19         if self._funcion_hash is None:
20             # Evitar la re-creación
21             del hash
22             self._funcion_hash = self.
23                 crear_funcion_hash()
24
25     def get_funcion_hash(self):
26         return self._funcion_hash
27
28     @staticmethod
29     def crear_funcion_hash():
30         """
31         Función que crea una función
32             hash que asigna un valor a
33             cada letra del alfabeto
34             español.
35         Retorna:
36             funcion_hash(dict):
37                 diccionario que
38                 contiene la función
39                 hash.
40         Ejemplo:
41             {'a': 0, 'b': 1, ..., 'z':
42                 13, 'ñ': 14, 'á': 15,
43                 'í': 16, 'í': 17, '
44                 ó': 18, 'ú': 19, 'ä':
45                 20, 'ë': 21, 'ï': 22,
46                 'ö': 23, 'ü': 24}
47         """
48         print("Creando función hash...")
49
50         funcion_hash = {}
51         valor = 0
52
53         # De 'a' a la 'n'
54         for char in range(ord('a'),
55             ord('n')+1):
56             funcion_hash[chr(char)] =
57                 valor
58             valor += 1
59
60         # La 'ñ'
61         funcion_hash['ñ'] = valor
62         valor += 1
63
64         # De la 'o' hasta la 'z'
65         for char in range(ord('o'),
66             ord('z')+1):
67             funcion_hash[chr(char)] =
68                 valor
69             valor += 1
70
71         # Caracteres correspondientes

```

```

51         a las vocales con tilde
52         acentuadas = "áííóú"
53         for char in acentuadas:
54             funcion_hash[char] = valor
55             valor += 1
56
57         # Caracteres correspondientes
58         a las vocales con diéresis
59         diéresis = "äëïöü"
60         for char in diéresis:
61             funcion_hash[char] = valor
62             valor += 1
63
64         print("Función hash creada.")
65         return funcion_hash

```

Listing 2: Función Hash

```

1 def procesar(id_peticion, puntaje,
2   lexemas_retorno, puntuacion):
3     """
4     Función que procesa el texto
5     ingresado por el usuario y
6     muestra los resultados en una
7     ventana emergente.
8     Parámetros:
9     id_peticion(int): 0 para el
10      personal, 1 para el
11      cliente
12     puntaje([]): array de 3
13      elementos para almacenar
14      el puntaje de cada tipo de
15      lexema
16     lexemas_retorno([]): lista de
17      lexemas
18     Returns:
19     None
20     """
21     # Cargar el tokenizador
22     dependiendo de la petición
23     raiz = cargar_tokenizador(
24         id_peticion)
25
26     token_mala = ''
27     token_neutra = ''
28     token_buena = ''
29
30     # Se establecen las etiquetas
31     dependiendo de la petición
32     if id_peticion == 0:
33         token_mala = 'ATC_MALA'
34         token_neutra = 'ATC_NEUTRA'
35         token_buena = 'ATC_BUENA'
36     else:
37         token_mala = 'EXP_MALA'
38         token_neutra = 'EXP_NEUTRA'
39         token_buena = 'EXP_BUENA'
40
41     # Procesar el texto dependiendo de
42     la petición
43     if id_peticion == 0:
44         print('Procesando el texto del
45             personal...')
46         entrada = texto_area_personal.
47             get("1.0", "end-1c")

```

```

32 else:
33     print('Procesando el texto del
34         cliente...')
35     entrada = texto_area_cliente.
36         get("1.0", "end-1c")
37
38     ban_saludo, ban_despedida = False,
39         False
40     # Preprocesamiento de la entrada
41     entrada = entrada.lower()
42     entrada = entrada + ' '
43     if id_peticion == 0:
44
45         # Mapear si en la entrada hay
46         algun saludo
47         print('Mapeando saludos...')
48         saludos = ['buenos días', '
49             buen día', 'buenas tardes'
50             , 'buenas noches']
51         ban_saludo = False
52         for saludo in saludos:
53             if saludo in entrada:
54                 ban_saludo = True
55                 break
56         if ban_saludo:
57             print('Saludo encontrado')
58         else:
59             print('Saludo no
60                 encontrado')
61
62         # Mapear si en la entrada hay
63         alguna despedida
64         print('Mapeando despedidas...')
65         despedidas = ['hasta luego']
66         ban_despedida = False
67         for despedida in despedidas:
68             if despedida in entrada:
69                 ban_despedida = True
70                 break
71         if ban_despedida:
72             print('Despedida
73                 encontrada')
74         else:
75             print('Despedida no
76                 encontrada')
77
78         # Si no hay texto, no hacer nada
79         if not entrada:
80             print('No hay texto para
81                 procesar')
82             return
83         else:
84             print('Texto a procesar: ' +
85                 entrada)
86
87         # Generar la función hash
88         hash_alfabeto = HashFunction()
89             .get_funcion_hash()
90
91         siguiente = raiz
92         lexema = ''
93         lexemas = []
94
95         # Preprocesamiento de la

```

<pre> 83 entrada 84 entrada = entrada.lower() 85 entrada = entrada + ' ' 86 87 # Contadores de lexemas por 88 token 89 buena = 0 90 neutra = 0 91 mala = 0 92 93 # Por cada caracter en la 94 entrada 95 for caracter in entrada: 96 97 # Si el caracter no es un 98 espacio, salto de l 99 ínea, tabulación o 100 retorno de carro 101 seguir procesando 102 caracteres, caso 103 contrario o es un 104 lexema ya completo o 105 es un espacio en 106 blanco 107 108 if not(caracter in [' ', '\n', '\t', '\r', '.']): 109 : 110 # Si el caracter es un 111 caracter especial 112 , no se considera 113 caso contrario se 114 agrega al lexema y 115 se mueve al 116 siguiente nodo 117 118 if caracter in ['/', '*', '\', '?', '!', '(', ')', '"', ':', ';', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9']: 119 continue 120 121 else: 122 lexema = lexema + 123 caracter 124 siguiente = 125 siguiente. 126 mover(caracter 127) 128 129 else: 130 # Si el lexema es vac 131 ío, no hacer nada 132 caso contrario se 133 marca el nodo 134 actual como estado 135 final 136 137 if lexema == '': 138 continue 139 140 else: 141 siguiente. 142 estado_final = 143 True 144 145 # Interfaz para 146 preguntar al </pre>	<pre> 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 </pre>	<pre> usuario que tipo de lexema es en caso de no tener un token asignado if siguiente.token == '': print('\t' + lexema + ' no pertenece a ningún token') # Mostrar ventana emergente mostrar_emergente (siguiente , lexema, id_peticion) else: print('\t' + lexema + ' pertenece al token ' + siguiente. token) # Agregar el lexema a la lista de lexemas lexemas.append(lexema) # Contador de lexemas por token if siguiente.token [-5:] == " BUENA": buena += 1 elif siguiente. token[-6:] == "NEUTRA": neutra += 1 else: mala += 1 # Reiniciar el lexema y el nodo actual vuelve a la raiz lexema = '' siguiente = raiz # Mostrar los resultados en una ventana emergente if id_peticion == 0: # Inicializar las puntuaciones puntuacion_buena = buena </pre>
---	--	--

```

136     puntuacion_mala = mala
137
138     # Ajustar las puntuaciones
        basadas en los indicadores
        de saludo y despedida
139     if ban_saludo:
140         puntuacion_buena += 1
141     else:
142         puntuacion_mala += 1
143
144     if ban_despedida:
145         puntuacion_buena += 1
146     else:
147         puntuacion_mala += 1
148
149     # Verificar que no haya divisi
        ón por cero
150     total_puntuaciones =
        puntuacion_buena +
        puntuacion_mala
151     if total_puntuaciones == 0:
152         puntuacion_personal = 1
153     else:
154
155         # Normalizar los puntajes
        buena_normalizado =
        puntuacion_buena /
        total_puntuaciones
157         mala_normalizado =
        puntuacion_mala /
        total_puntuaciones
158
159         # Calcular la ponderación
        final en escala del 1
        al 5
160         puntuacion_personal = 1 +
        ( 0 if (
            buena_normalizado -
            mala_normalizado) <= 0
        else (
            buena_normalizado -
            mala_normalizado) ) *
            4
161
162     puntuacion[0] =
        puntuacion_personal
163
164     messagebox.showinfo("
        Resultados", "El analisis
        ha arrojado los siguientes
        resultados :\n\n- {} :
        {}\n- {}: {} \n- {}: {} \n-
        SALUDO: {} \n- DESPEDIDA:
        {} \n\n- Balance buenas:
        {} \n- Balance malas: {} \
        nPuntuacion final: {:.2f}"
        .format(token_buena, buena
        , token_neutra, neutra,
        token_mala, mala, 'Si' if
        ban_saludo else 'No', 'Si'
        if ban_despedida else 'No
        ', puntuacion_buena,
        puntuacion_mala,
        puntuacion_personal))
165 else:

```

```

166
167     # Inicializar las puntuaciones
        puntuacion_buena = buena
        puntuacion_mala = mala
168
169
170
171     # Verificar que no haya divisi
        ón por cero
172     total_puntuaciones =
        puntuacion_buena +
        puntuacion_mala
173     if total_puntuaciones == 0:
174         puntuacion_cliente = 1
175     else:
176
177         # Normalizar los puntajes
        buena_normalizado =
        puntuacion_buena /
        total_puntuaciones
179         mala_normalizado =
        puntuacion_mala /
        total_puntuaciones
180
181         # Calcular la ponderación
        final en escala del 1
        al 5
182         puntuacion_cliente = 1 + (
            0 if (
                buena_normalizado -
                mala_normalizado) <= 0
            else (
                buena_normalizado -
                mala_normalizado) ) *
                4
183
184         puntuacion[1] =
            puntuacion_cliente
185
186         messagebox.showinfo("
            Resultados", "El analisis
            ha arrojado los siguientes
            resultados :\n\n- {}: {} \
            n- {}: {} \n- {}: {} \n\
            nPuntuacion final: {:.2f}"
            .format(token_buena, buena
            , token_neutra, neutra,
            token_mala, mala,
            puntuacion_cliente))
187
188     # Actualizar el puntaje
        puntaje[0] = buena
        puntaje[1] = neutra
        puntaje[2] = mala
189
190
191
192
193     # Eliminar lexemas duplicados
        convirtiendo a conjunto y
        luego a lista
194     lexemas = list(set(lexemas))
195     # Ordenar lexicográficamente
        lexemas.sort()
196
197
198     # Copiar lexemas en
        lexemas_retorno
199     lexemas_retorno.clear()
200     for lexema in lexemas:
201         lexemas_retorno.append(lexema)

```

```

202     resaltar_palabras(raiz,
203                     id_peticion, lexemas_retorno)
204
205     # Guardar el tokenizador
206     guardar_tokenizador(raiz,
207                       id_peticion)
208
209     # Liberar la memoria
210     raiz = None

```

Listing 3: Core del análisis. Procesar

5. CASO DE PRUEBA

A continuación, se detallan los pasos necesarios para procesar el caso de prueba:

- **Entrada – Atencion al Cliente**
Buenos días, le saluda Joanna de atención al cliente, ¿en qué puedo ayudarle? Puedo darle esa información. Por favor, ¿me facilita su número de documento? Gracias, una pregunta de seguridad, ¿puede darme el año de su fecha de nacimiento? Gracias. Verifico que su saldo es de doscientos treinta mil guaraníes a la fecha de ayer. ¿Algo más que pueda hacer por usted? Gracias por llamar al servicio de atención al cliente. Que tenga un buen día.
- **Entrada – Cliente**
Buen día. Desde ayer que no puedo consultar mi saldo. Claro, tres millones doscientos sesenta mil cero sesenta y ocho. En año de mi nacimiento es mil novecientos noventa y seis. No, muchas gracias. Usted ha sido muy amable. Hasta luego.

5.1. PROCESAMIENTO DE LA INTERACCIÓN DEL ATC

A continuación se detallan los pasos a seguir para procesar la interacción del personal de atención al cliente.

5.1.1. INGRESO DE LOS DATOS

Primero, se ingresa el texto en el apartado correspondiente al Personal de Atención al Cliente (ATC), tal como se muestra en la Figura 5.

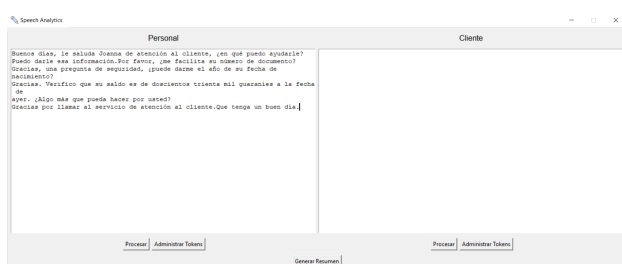


Figura 5: Ingreso de datos del personal

5.1.2. PROCESAMOS EL TEXTO

A continuación, se presiona el botón **Procesar** correspondiente al personal, tal como se muestra en la Figura 6.

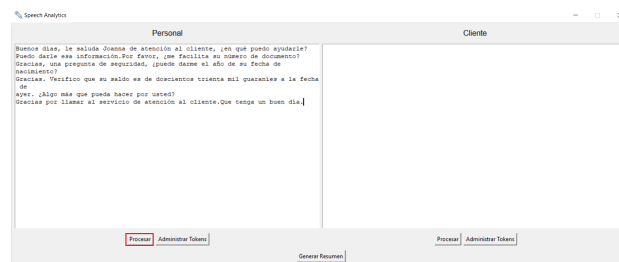


Figura 6: Procesamiento del texto ingresado

5.1.3. CATEGORIZACIÓN DE LOS LEXEMAS

Al principio, es necesario entrenar el programa, ya que aún no tiene cargada ninguna información. Por lo tanto, debemos indicarle palabra por palabra a qué token pertenecen, tal y como se observa en la Figura 7.

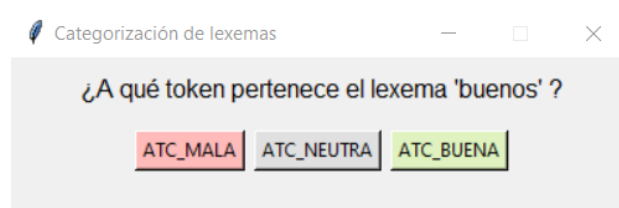


Figura 7: Categorización de los lexemas

5.1.4. RESULTADOS PRELIMINARES

Una vez completado el paso anterior, podremos observar los resultados generados para este análisis, los cuales se ajustarán a los criterios establecidos previamente. En la Figura 8, podremos observar esto. Estos resultados nos proporcionan información sobre el desempeño del personal de ATC y nos permiten identificar si es necesario realizar mejoras en su capacitación.

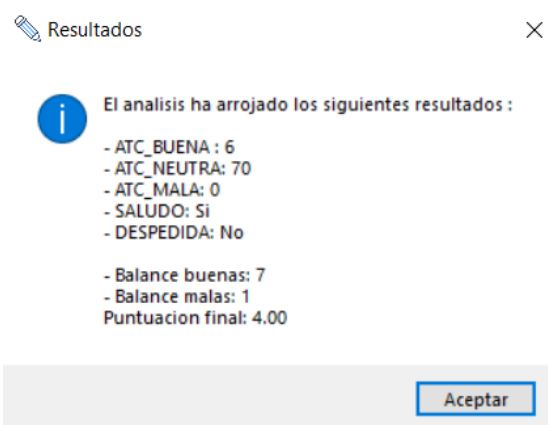


Figura 8: Resultados del análisis

En la pantalla principal también podremos visualizar los resultados de manera gráfica de la categorización, tal y como se muestra en la Figura 9. En donde el color de resalta-do indica el token al que pertenece la palabra.

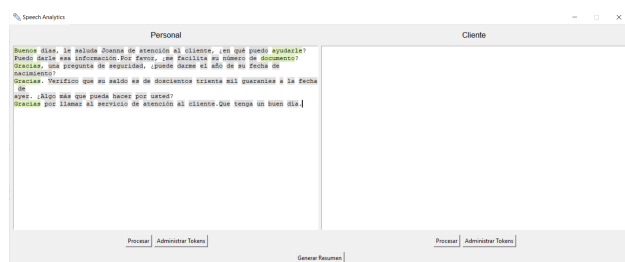


Figura 9: Resultado de la categorización

5.1.5. ADMINISTRACIÓN DE TOKENS

Una vez procesado el texto, también podemos gestionar los tokens de las palabras reconocidas en el texto. Es decir, tenemos la opción de cambiar los tokens ya establecidos para que el tokenizador pueda actualizar su base de datos y aprender conforme vayamos estableciendo los parámetros. Para acceder a esta función, simplemente hacemos clic en el botón **Administrar Tokens**, como se muestra en la Figura 10.

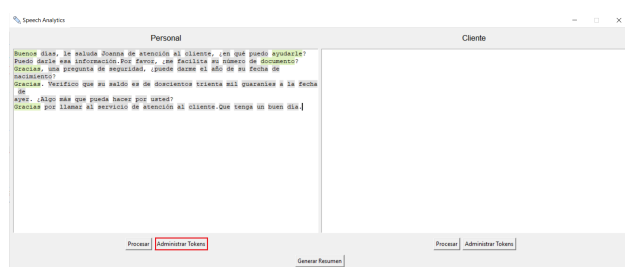


Figura 10: Administración de tokens

A continuación, se desplegará la ventana de administración de tokens, donde tendremos la capacidad de modificar el token asociado al lexema reconocido, tal y como se puede observar en la Figura 11.

Gestor de Tokens

#	Palabra	Token Actual	Cambio Token			
1	a	ATC_NEUTRA	ATC_MALA	ATC_NEUTRA	ATC_BUENA	
2	al	ATC_NEUTRA	ATC_MALA	ATC_NEUTRA	ATC_BUENA	
3	algo	ATC_NEUTRA	ATC_MALA	ATC_NEUTRA	ATC_BUENA	
4	atención	ATC_NEUTRA	ATC_MALA	ATC_NEUTRA	ATC_BUENA	
5	ayer	ATC_NEUTRA	ATC_MALA	ATC_NEUTRA	ATC_BUENA	
6	ayudarle	ATC_BUENA	ATC_MALA	ATC_NEUTRA	ATC_BUENA	

Figura 11: Ventana del administrador de tokens

5.2. PROCESAMIENTO DE LA INTERACCIÓN DEL CLIENTE

Todos los pasos descritos anteriormente para la interacción del personal de atención al cliente (ATC) se repiten para el cliente. El resultado final se puede observar en la Figura 12. Esta información es útil para determinar la calidad del servicio ofrecido.

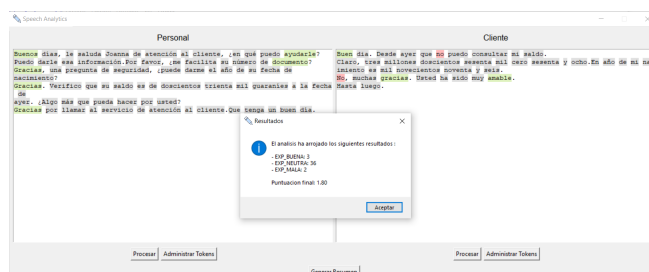


Figura 12: Ventana del administrador de tokens

5.3. RESULTADO Y PUNTUACIÓN GENERAL DE LA INTERACCIÓN

A continuación, para generar el resumen general de la interacción, debemos presionar el botón **Generar Resumen**, como se muestra en la Figura 13.

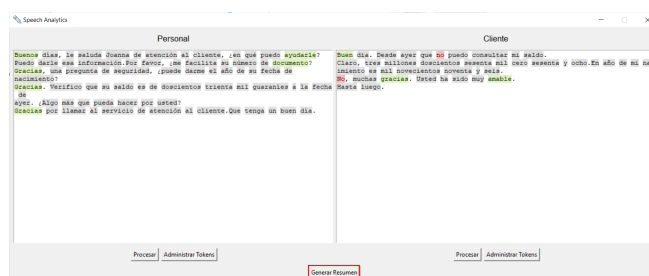


Figura 13: Generación del resumen general

Esto nos generará un resumen de la interacción de la llamada entre el ATC y el cliente. Esta puntuación es un promedio de ambas interacciones y nos proporciona una noción de la puntuación general de la llamada realizada. El resultado se puede observar en la Figura 14.

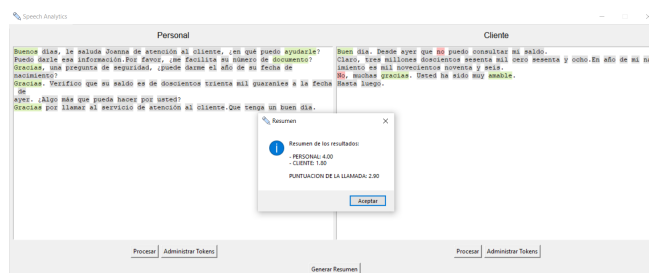


Figura 14: Generación del resumen general

REFERENCIAS

- [1] Alfred V. Aho et al. *Compiladores. Principios, técnicas y herramientas*. Pearson PLC, 2008.
- [2] Real Academia Española. “La Academia entrega la 23.ª edición del DRAE, que se publicará en octubre”. En: *Diccionario de la lengua española* (2014). Consultado el 27 de junio de 2024. URL: <https://www.rae.es>.