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

Elías Cristaldo[1]

[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 procesar 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 categorizacion de los tokens por colores de acuerdo si es una palabra buena, nuetral 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.

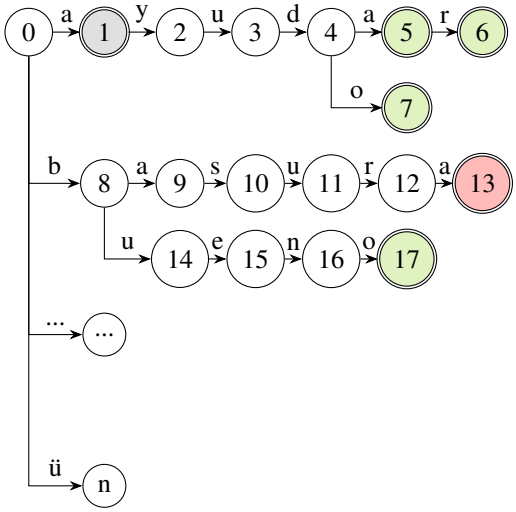

Figura 2: Grafo dirigido con tres nodos.

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

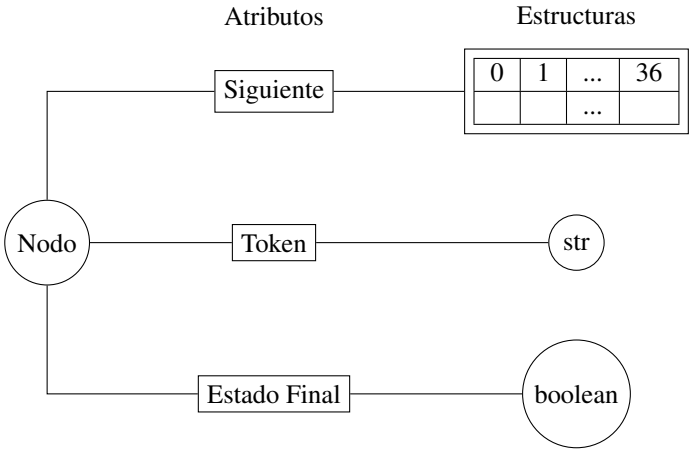

Figura 3: Modelo de un Nodo en el AFD.

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

$$N = 93.000\ [palabras] * 6\ \left[\frac{letras}{palabras}\right]$$
$$= 558.000\ [letras]$$

como cada letra representa un nodo, entonces...

$$C = 558.000\ [nodo] * 464\ \left[\frac{bytes}{nodo}\right]$$
$$= 258.912.000\ [bytes]$$

pasando a MB tenemos que ...

$$258.912.000\ [bytes] \equiv \frac{258.912.000}{2^{20}}$$
$$\approx 258{,}912\,[MB]$$

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:

$$t_{lista} = 0{,}00008580\,[segundos]$$
$$t_{hash} = 0{,}00005990\,[segundos]$$

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

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.

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

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

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

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
           DFA (Deterministic Finite
           Automaton) que permita
           identificar palabras en un
           texto.
4      Atributos:
5          siguiente([]): arreglo de 37
               elementos, que representa
               las posibles transiciones
               del nodo.
6          token(str): token al que
               pertenecen las letras que
               forman la palabra que
               llega al nodo en caso de
               ser un estado final.
7          estado_final(bool): indica si
               el nodo es un estado final
                o no.
8      Ejemplo:
9          siguiente = [Nodo1, Nodo2,
               Nodo3, ..., Nodo37]
10             Nodo1: transición con la
                   letra 'a'
11             Nodo2: transición con la
                   letra 'b'
12             ...
```

```
            Nodo37: transición con la
                letra 'ü'
        token = 'ATC_BUENA'
        estado_final = True
    """
    def __init__(self):
        self.siguiente = np.empty(37,
            dtype=object)
        self.token = ''
        self.estado_final = False

    def mover(self, caracter):
        """
        Metodo que permite mover de un
             nodo a otro, dependiendo
            de la letra que se recibe
            como parámetro.
        Parametros:
            caracter(str): letra que
                se recibe para mover
                al siguiente nodo.
        Returns:
            nodo_siguiente(Nodo): nodo
                 al que se movió.
        """
        print("Moviendo al siguiente
            nodo con {}...".format(
            caracter))

        nodo_siguiente = None
        hash_alfabeto = HashFunction()
            .get_funcion_hash()

        if self.siguiente[
            hash_alfabeto[caracter]]:
            nodo_siguiente = self.
                siguiente[
                hash_alfabeto[caracter
                ]]
        else:
            nodo_siguiente = Nodo()
            self.siguiente[
                hash_alfabeto[caracter
                ]] = nodo_siguiente

        print("Transición completada."
            )
        return nodo_siguiente
```

Listing 1: Clase Nodo

```
class HashFunction:
    """
    Clase que implementa el patrón
        Singleton para crear una funci
        ón hash que asigna un valor a
        cada letra del alfabeto espa
        ñol.
    Ejemplo:
        hash_function = HashFunction()
            .get_funcion_hash()
    """
    _instance = None  # Variable de
        clase para almacenar la
        instancia única
    _funcion_hash = None  # Variable
        de clase para almacenar la
        función hash

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().
                __new__(cls)
        return cls._instance

    def __init__(self):
        if self._funcion_hash is None:
            # Evitar la re-creación
            del hash
            self._funcion_hash = self.
                crear_funcion_hash()

    def get_funcion_hash(self):
        return self._funcion_hash

    @staticmethod
    def crear_funcion_hash():
        """
        Función que crea una función
            hash que asigna un valor a
             cada letra del alfabeto
            español.
        Retorna:
            funcion_hash(dict):
                diccionario que
                contiene la función
                hash.
        Ejemplo:
            {'a': 0, 'b': 1, ..., 'z':
                 13, 'ñ': 14, 'á': 15,
                 'í': 16, 'í': 17, '
                ó': 18, 'ú': 19, 'ä':
                20, 'ë': 21, 'ï': 22,
                'ö': 23, 'ü': 24}
        """
        print("Creando función hash...
            ")

        funcion_hash = {}
        valor = 0

        # De 'a' a la 'n'
        for char in range(ord('a'),
            ord('n')+1):
            funcion_hash[chr(char)] =
                valor
            valor += 1

        # La 'ñ'
        funcion_hash['ñ'] = valor
        valor += 1

        # De la 'o' hasta la 'z'
        for char in range(ord('o'),
            ord('z')+1):
            funcion_hash[chr(char)] =
                valor
            valor += 1

        # Caracteres correspondientes
```

```
            a las vocales con tilde
        acentuadas = "áííóú"
        for char in acentuadas:
            funcion_hash[char] = valor
            valor += 1

        # Caracteres correspondientes
            a las vocales con dieresis
        dieresis = "äëïöü"
        for char in dieresis:
            funcion_hash[char] = valor
            valor += 1

        print("Función hash creada.")
        return funcion_hash
```

Listing 2: Función Hash

```
def procesar(id_peticion, puntaje,
    lexemas_retorno, puntuacion):
    """
    Función que procesa el texto
        ingresado por el usuario y
        muestra los resultados en una
        ventana emergente.
    Parámetros:
        id_peticion(int): 0 para el
            personal, 1 para el
            cliente
        puntaje([]): array de 3
            elementos para almacenar
            el puntaje de cada tipo de
             lexema
        lexemas_retorno([]): lista de
            lexemas
    Returns:
        None
    """
    # Cargar el tokenizador
        dependiendo de la petición
    raiz = cargar_tokenizador(
        id_peticion)

    token_mala = ''
    token_neutra = ''
    token_buena = ''

    # Se establecen las etiquetas
        dependiendo de la petición
    if id_peticion == 0:
        token_mala = 'ATC_MALA'
        token_neutra = 'ATC_NEUTRA'
        token_buena = 'ATC_BUENA'
    else:
        token_mala = 'EXP_MALA'
        token_neutra = 'EXP_NEUTRA'
        token_buena = 'EXP_BUENA'

    # Procesar el texto dependiendo de
         la petición
    if id_peticion == 0:
        print('Procesando el texto del
             personal...')
        entrada = texto_area_personal.
            get("1.0", "end-1c")
    else:
        print('Procesando el texto del
             cliente...')
        entrada = texto_area_cliente.
            get("1.0", "end-1c")

    ban_saludo, ban_despedida = False,
         False
    if id_peticion == 0:

        # Mapear si en la entrada hay
            algun saludo
        print('Mapeando saludos...')
        saludos = ['buenos días', '
            buen día', 'buenas tardes'
            , 'buenas noches']
        ban_saludo = False
        for saludo in saludos:
            if saludo in entrada:
                ban_saludo = True
                break
        if ban_saludo:
            print('Saludo encontrado')
        else:
            print('Saludo no
                encontrado')

        # Mapear si en la entrada hay
            alguna despedida
        print('Mapeando despedidas...'
            )
        despedidas = ['hasta luego']
        ban_despedida = False
        for despedida in despedidas:
            if despedida in entrada:
                ban_despedida = True
                break
        if ban_despedida:
            print('Despedida
                encontrada')
        else:
            print('Despedida no
                encontrada')

    # Si no hay texto, no hacer nada
    if not entrada:
        print('No hay texto para
            procesar')
        return
    else:
        print('Texto a procesar: ' +
            entrada)

        # Generar la función hash
        hash_alfabeto = HashFunction()
            .get_funcion_hash()

        siguiente = raiz
        lexema = ''
        lexemas = []

        # Preprocesamiento de la
            entrada
        entrada = entrada.lower()
        entrada = entrada + ' '
```

```
82
83          # Contadores de lexemas por token
84          buena = 0
85          neutra = 0
86          mala = 0
87
88          # Por cada caracter en la entrada
89          for caracter in entrada:
90
91              # Si el caracter no es un espacio, salto de línea, tabulación o retorno de carro seguir procesando caracteres, caso contrario o es un lexema ya completo o es un espacio en blanco
92              if not(caracter in [' ', '\n', '\t', '\r', '.']):
93                  # Si el caracter es un caracter especial, no se considera caso contrario se agrega al lexema y se mueve al siguiente nodo
94                  if caracter in ['/', ',', '?', '¿', '!', '¡', '(', ')', '"', ':', ';', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' ]:
95                      continue
96                  else:
97                      lexema = lexema + caracter
98                      siguiente = siguiente.mover(caracter)
99              else:
100                 # Si el lexema es vacío, no hacer nada caso contrario se marca el nodo actual como estado final
101                 if lexema == '':
102                     continue
103                 else:
104                     siguiente.estado_final = True
105                     # Interfaz para preguntar al usuario que tipo de lexema es en caso de no tener un token asignado
106                     if siguiente.token == '':
107                         print('\t' + lexema + ' no pertenece a ningún token')
108                         # Mostrar ventana emergente
109                         mostrar_emergente(siguiente, lexema, id_peticion)
110                     else:
111                         print('\t' + lexema + ' pertenece al token ' + siguiente.token)
112
113                     # Agregar el lexema a la lista de lexemas
114                     lexemas.append(lexema)
115
116                     # Contador de lexemas por token
117                     if siguiente.token[-5:] == "BUENA":
118                         buena += 1
119                     elif siguiente.token[-6:] == "NEUTRA":
120                         neutra += 1
121                     else:
122                         mala += 1
123
124                     # Reiniciar el lexema y el nodo actual vuelve a la raiz
125                     lexema = ''
126                     siguiente = raiz
127
128 # Mostrar los resultados en una ventana emergente
129 if id_peticion ==  0:
130
131     # Inicializar las puntuaciones
132     puntuacion_buena = buena
133     puntuacion_mala = mala
134
135     # Ajustar las puntuaciones basadas en los indicadores
```

```
            de saludo y despedida
        if ban_saludo:
            puntuacion_buena += 1
        else:
            puntuacion_mala += 1

        if ban_despedida:
            puntuacion_buena += 1
        else:
            puntuacion_mala += 1

        # Verificar que no haya divisi
            ón por cero
        total_puntuaciones =
            puntuacion_buena +
            puntuacion_mala
        if total_puntuaciones == 0:
            puntuacion_personal = 1
        else:

            # Normalizar los puntajes
            buena_normalizado =
                puntuacion_buena /
                total_puntuaciones
            mala_normalizado =
                puntuacion_mala /
                total_puntuaciones

            # Calcular la ponderación
                final en escala del 1
                al 5
            puntuacion_personal = 1 +
                ( 0 if (
                buena_normalizado -
                mala_normalizado) <= 0
                 else (
                buena_normalizado -
                mala_normalizado) ) *
                4

        puntuacion[0] =
            puntuacion_personal

        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))
    else:

        # Inicializar las puntuaciones
        puntuacion_buena = buena
        puntuacion_mala = mala

        # Verificar que no haya divisi
            ón por cero
        total_puntuaciones =
            puntuacion_buena +
            puntuacion_mala
        if total_puntuaciones == 0:
            puntuacion_cliente = 1
        else:

            # Normalizar los puntajes
            buena_normalizado =
                puntuacion_buena /
                total_puntuaciones
            mala_normalizado =
                puntuacion_mala /
                total_puntuaciones

            # Calcular la ponderación
                final en escala del 1
                al 5
            puntuacion_cliente = 1 + (
                 0 if (
                buena_normalizado -
                mala_normalizado) <= 0
                 else (
                buena_normalizado -
                mala_normalizado) ) *
                4

        puntuacion[1] =
            puntuacion_cliente

        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))

    # Actualizar el puntaje
    puntaje[0] = buena
    puntaje[1] = neutra
    puntaje[2] = mala

    # Eliminar lexemas duplicados
        convirtiendo a conjunto y
        luego a lista
    lexemas = list(set(lexemas))
    # Ordenar lexicográficamente
    lexemas.sort()

    # Copiar lexemas en
        lexemas_retorno
    lexemas_retorno.clear()
    for lexema in lexemas:
        lexemas_retorno.append(lexema)

    resaltar_palabras(raiz,
        id_peticion, lexemas_retorno)
```

```
202     # Guardar el tokenizador
203     guardar_tokenizador(raiz,
            id_peticion)
204
205     # Liberar la memoria
206     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 trienta 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 dia. 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 my 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.

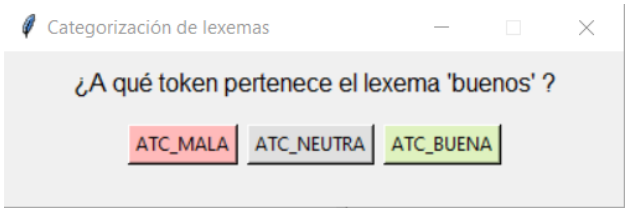

Figura 7: Categorización de los lexemas

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.

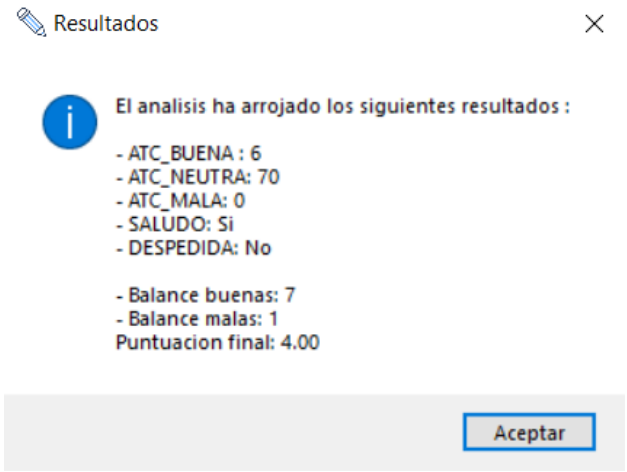

Figura 8: Resultados del análisis

En la pantalla principal también podremos visualizar los resultados de manera gráfica de la categorización, tal y co-

mo se muestra en la Figura 9. En donde el color de resaltado indica el token al que pertenece la palabra.

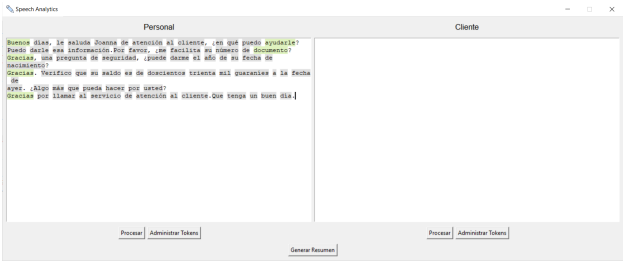

Figura 9: Resultado de la categorización

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.

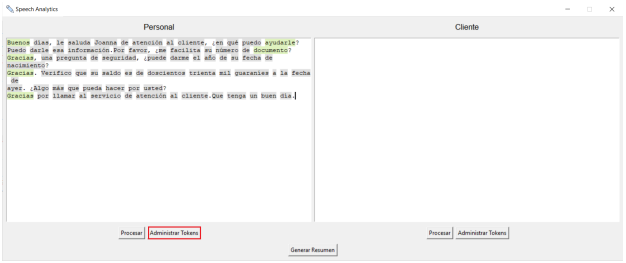

Figura 10: Administración de tokens

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.

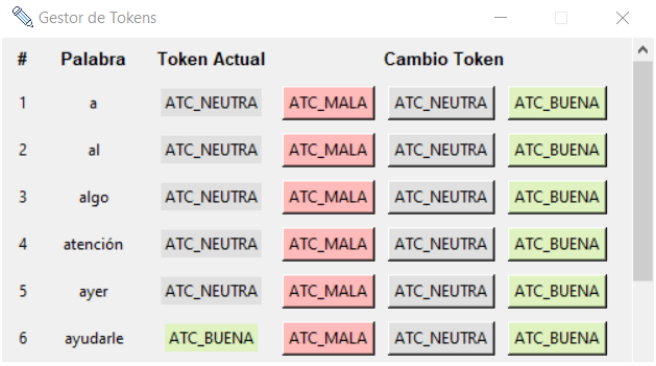

Figura 11: Ventana del administrador de tokens

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.

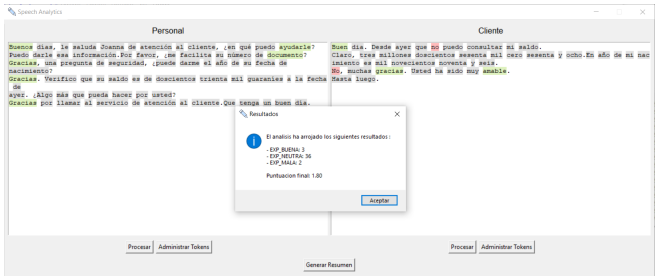

Figura 12: Ventana del administrador de tokens

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.

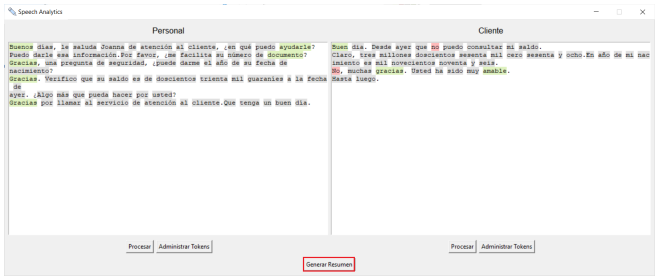

Figura 13: Generación del resumen general

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.

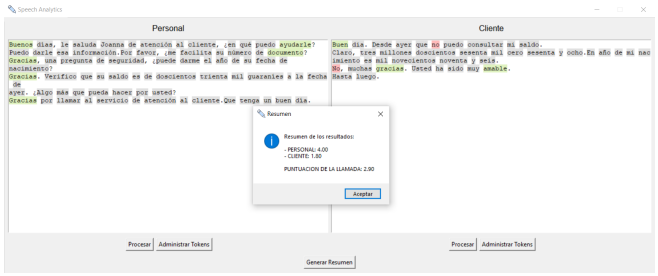

Figura 14: Generación del resumen general

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`.