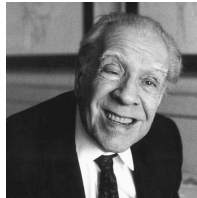


Anexo 8

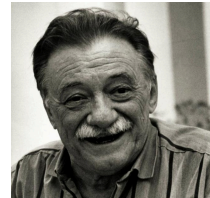
Proyecto 8: Clasificador de texto

Mg. Luis Felipe Bustamante Narváez

En este ejercicio realizaremos un clasificador de texto basado en la forma escritural, sintáctica y semántica de dos escritores latinoamericanos, por un lado al argentino Jorge Luis Borges, y por otro frente al uruguayo Mario Benedetti.



Jorge Luis Borges



Mario Benedetti

1. Temas y Filosofía

Borges: Su poesía es filosófica, abstracta y llena de referencias literarias, mitológicas y metafísicas. Le interesaban temas como el tiempo, el infinito, el destino, la identidad y la memoria. Su tono es intelectual y a veces enigmático.

Benedetti: Escribe de manera más directa y accesible. Sus temas son el amor, la vida cotidiana, la lucha social, el exilio y la esperanza. Su tono es cálido, humano y cercano al lector.

2. Lenguaje y Estilo

Borges: Usa un lenguaje elegante, erudito y con muchas metáforas complejas. Su poesía es reflexiva, con estructuras clásicas y a veces con formas fijas como sonetos.

Benedetti: Usa un lenguaje sencillo, directo y coloquial. Sus poemas parecen conversaciones o pensamientos escritos sin mucha ornamentación.

3. Estructura y Ritmo

Borges: Tiende a usar estructuras más tradicionales con rima y métrica cuidadas, aunque también experimenta con versos libres.

Benedetti: Prefiere el verso libre y la naturalidad del habla cotidiana, sin preocuparse demasiado por la métrica.

Librerías

```
In [... pip install tensorflow
In [... pip install colorama
In [... import numpy as np
import matplotlib.pyplot as plt
import string
from sklearn.model_selection import train_test_split
import os
from tqdm import tqdm
import PyPDF2
from IPython.display import display, HTML
from tensorflow.keras.preprocessing.sequence import pad_sequences
from colorama import Fore, Back, Style
```

Cargamos los documentos

```
In [... def extraer_texto_desde_pdf(ruta_archivo):
with open(ruta_archivo, 'rb') as archivo:
lector = PyPDF2.PdfReader(archivo)
```

```

    texto = ''
    for pagina in range(len(lector.pages)):
        texto += lector.pages[pagina].extract_text()
    return texto

```

```
In [... ruta_carpeta = 'textos'
```

Guardamos los textos en una lista

```

In [... todos_los_textos = []
for archivo in tqdm(os.listdir(ruta_carpeta),
                    bar_format=f'{Back.WHITE}{Fore.GREEN}{{1_bar}}{{bar}}{{Style.RESET_ALL}}'):
    if archivo.endswith('.pdf'):
        ruta_completa = os.path.join(ruta_carpeta, archivo)
        try:
            documento = extraer_texto_desde_pdf(ruta_completa)
            todos_los_textos.append(documento)
        except Exception as e:
            print(f'Error al procesar {archivo}: {e}')

```

```
100%|██████████
```

```
In [... todos_los_textos[0]
```

Procesamiento de los datos

Vamos a separar los textos por etiquetas, enumerando los textos de Borgues con la etiqueta 0 y los de Benedetti con la etiqueta 1

```

In [... # Eliminamos espacios al inicio y al final para evitar problemas con el pdf
for texto in todos_los_textos:
    archivo = texto.strip()
    archivo

```

```

In [... # Creamos las listas vacías
textos = []
etiquetas = []

```

```

In [... # Mostramos línea por línea los textos de cada escritor
count = 0 #contador de línea para pruebas
for etiqueta, texto in enumerate(todos_los_textos):
    print(f'\n--- Texto {etiqueta} ---\n')
    for linea in texto.split(' \n'):
        count += 1 #contador de líneas para prueba
        print(linea)
        # Convertimos a minúsculas
        linea = linea.rstrip().lower()
        print(linea)
        # Eliminamos signos de puntuación
        if linea:
            linea = linea.translate(str.maketrans(' ', '', string.punctuation))
            print(linea)
            # Agregamos el texto limpio y le asignamos su respectiva etiqueta
            textos.append(linea)
            etiquetas.append(etiqueta)

```

```

In [... # Mostramos las listas
textos

```

Entrenamiento

X representa la lista de los textos, y Y, representa la lista de las etiquetas, quien sería nuestra variable a predecir.

```
In [... X_train, X_test, Y_train, Y_test = train_test_split(textos, etiquetas, train_size=0.9, random_state=42)
```

```

In [... # Mostramos en forma de tupla el tamaño de cada muestra
len(Y_train), len(Y_test)

```

```
Out[... (2222, 247)
```

```
In [... # Probamos Las muestras de entrenamiento
X_train[0], Y_train[0]
```

```
Out[...] ('o acaso no la mira', 1)
```

```
In [... # Probamos Las muestras de prueba
X_test[0], Y_test[0]
```

```
Out[...] ('a inventar la verdad', 1)
```

Representación de palabras desconocidas

<unk>

Es una convención utilizada a menudo en **NPL** para representar palabras desconocidas o fuera del vocabulario. Por ejemplo, *si una palabra no se encontró en la muestra de entrenamiento, pero aparece en la muestra de prueba, será desconocida, y se requiere agregarle un índice que diferencie a esta palabra.*

```
In [... indice = 1
indice_palabras = {'<unk>': 0}
```

Construcción del diccionario de codificación de palabras a índice

```
In [... for texto in X_train:
    # Separamos cada palabra
    tokens = texto.split()
    #print(tokens) # Probamos como se ven Los tokens
    for token in tokens:
        # Buscamos si La palabra no está en el índice para luego agregarla sin repetir
        if token not in indice_palabras:
            indice_palabras[token] = indice
            indice += 1
```

```
In [... # Mostramos el índice de palabras - palabras únicas
indice_palabras
```

```
In [... # tamaño de palabras únicas
indice_palabras
```

Conversión del índice de palabras de String a enteros

Cómo el entrenamiento no se debe hacer con palabras, creamos una muestra convertida a su valor específico en enteros

```
In [... # Listas para enteros
X_train_int = []
X_test_int = []
# Banderas para ejecutarse una sola vez
X_int_train_hecho = False
X_int_test_hecho = False
```

```
In [... # Conversión de Los datos de entrenamiento
if not X_int_train_hecho:
    for texto in X_train:
        # dividimos de nuevo en palabras
        tokens = texto.split()
        # Por cada palabra encontrada la cambia por su valor numérico de la clave del diccionario
        linea_entero = [ indice_palabras[token] for token in tokens ]
        #print(linea_entero)
        # Agregamos el nuevo valor a la lista de entrenamiento
        X_train_int.append(linea_entero)
    X_int_train_hecho = True
    print("Conversión de entrenamiento ejecutada con éxito.")
else:
    print("La conversión de entrenamiento ya se había ejecutado previamente.")
```

Conversión de entrenamiento ejecutada con éxito.

```
In [... # Mostramos la conversión de entrenamiento
X_train_int
```

```
In [... # Conversión de Los datos de prueba -- Como puede haber desconocidos, debemos hacer esto:
if not X_int_test_hecho:
    for texto in X_test:
        tokens = texto.split()
        linea_entero = [indice_palabras.get(token, 0) for token in tokens] #trae el token o 0
        #print(linea_entero)
        X_test_int.append(linea_entero)
    X_int_test_hecho = True
    print("Conversión de prueba ejecutada con éxito.")
else:
    print("La conversión de prueba ya se había ejecutado previamente.")
```

Conversión de prueba ejecutada con éxito.

```
In [... # Mostramos la conversión de prueba
len(X_test_int)
```

```
Out[...] 247
```

Matriz de Transición

Como se indicó en la teoría de los procesos de **Markov**, se requiere construir una matriz de transición y los estados iniciales para cada escritor:

1. Creamos un vector **V** con el tamaño total del **indice_palabras**
2. Creamos la matriz **A0** para las palabras de **Borges**
3. Creamos el vector de probabilidad inicial **pi0** para las palabras de **Borges**
4. Creamos la matriz **A1** para las palabras de **Benedetti**
5. Creamos el vector de probabilidad inicial **pi1** para las palabras de **Benedetti**

```
In [... V = len(indice_palabras)
# Creamos Las matrices y vectores con 1 para poder hacer el suavizado
A0 = np.ones((V, V))
pi0 = np.ones(V)
A1 = np.ones((V, V))
pi1 = np.ones(V)
#Mostramos, por ejemplo
pi0
```

```
Out[...] array([1., 1., 1., ..., 1., 1., 1.])
```

Función de conteo de palabras

```
In [... def compute_counts(texto_as_int, A, pi):
    #Recorremos Los tokens
    for tokens in texto_as_int:
        #Creamos el posible último elemento como referencia
        last_index = None
        #Recorremos cada elemento de cada línea
        for index in tokens:
            #Nos ubicamos en la primera secuencia
            if last_index is None:
                # Agregamos el valor inicial
                pi[index] +=1
            else:
                # Agregamos Los valores a La matriz
                A[last_index, index] += 1
            # Asignamos el valor actual al last_index
            last_index = index
```

```
In [... # Llamamos la función
#Para Borges
compute_counts([t for t, y in zip(X_train_int, Y_train) if y == 0], A0, pi0)
#Para Benedetti
compute_counts([t for t, y in zip(X_train_int, Y_train) if y == 1], A1, pi1)
```

```
In [... # Probamos
A1

Out[...] array([[1., 1., 1., ..., 1., 1., 1.],
               [1., 1., 2., ..., 1., 1., 1.],
               [1., 1., 1., ..., 1., 1., 1.],
               ...,
               [1., 1., 1., ..., 1., 1., 1.],
               [1., 1., 1., ..., 1., 1., 1.],
               [1., 1., 1., ..., 1., 1., 1.]])
```

Explicación

pi0 = array([1., 10., 1., ..., 1., 1., 1.])

pi1 = array([1., 14., 1., ..., 1., 1., 1.])

A0 = array([[1., 1., 1., ..., 1., 1., 1.], [1., 1., 1., ..., 1., 1., 1.], [1., 1., 1., ..., 1., 1., 1.], ..., [1., 1., 1., ..., 1., 1., 1.], [1., 1., 1., ..., 1., 1., 1.], [1., 1., 1., ..., 1., 1., 1.]])

A1 = array([[1., 1., 1., ..., 1., 1., 1.], [1., 1., 2., ..., 1., 1., 1.], [1., 1., 1., ..., 1., 1., 1.], ..., [1., 1., 1., ..., 1., 1., 1.], [1., 1., 1., ..., 1., 1., 1.], [1., 1., 1., ..., 1., 1., 1.]])

En el vector inicial **pi0**, la primera posición corresponde a los unk, la segunda posición corresponde a la palabra **o**, y nos está indicando que en los textos de Borges aparece iniciando la línea 10 veces. Si observamos los textos de Benedetti, **pi1**, nos indica que aparece 14 veces comenzando la línea. De esta manera podemos proceder a encontrar la probabilidad.

Con respecto a las matrices de transición, podemos observar en la matriz de Benedetti, **A1**, que en la segunda fila, hubo una transición de la palabra actual a la siguiente, valores que nos indicarán el comportamiento natural de los textos.

Distribución de Probabilidad

Normalizamos

Para observar las probabilidades, se requiere normalizar los vectores y matrices generados en el conteo, para que su valor oscile entre 0 y 1, como debe ser.

Esta es una manera empírica de demostrar las fórmulas mencionadas en la teoría

```
In [... #Conservamos Los datos originales A0, A1, pi0 y pi1, creando Las variables nomralizadas
#Para esto vamos a guardar Los datos originales
A0_norm = A0.copy()
pi0_norm = pi0.copy()
A1_norm = A1.copy()
pi1_norm = pi1.copy()
# Bandera de normalizado
normalize = False
```

```
In [... # Borges
if not normalize:
    # Borges
    A0_norm /= A0_norm.sum(axis=1, keepdims = True)
    pi0_norm /= pi0_norm.sum()
    # Benedetti
    A1_norm /= A1_norm.sum(axis=1, keepdims = True)
    pi1_norm /= pi1_norm.sum()
    print("Normalización ejecutada con éxito.")
    normalize = True
else:
    print("Las variables ya fueron normalizadas previamente.")
```

Normalización ejecutada con éxito.

```
In [... # Probamos
pi0_norm
```

```
Out[... array([0.00023111, 0.00231107, 0.00023111, ..., 0.00023111, 0.00023111,
0.00023111])
```

```
In [... A0_norm
```

```
Out[... array([[0.00029472, 0.00029472, 0.00029472, ..., 0.00029472, 0.00029472,
0.00029472],
[0.00029155, 0.00029155, 0.00029155, ..., 0.00029155, 0.00029155,
0.00029155],
[0.00029455, 0.00029455, 0.00029455, ..., 0.00029455, 0.00029455,
0.00029455],
...,
[0.00029472, 0.00029472, 0.00029472, ..., 0.00029472, 0.00029472,
0.00029472],
[0.00029472, 0.00029472, 0.00029472, ..., 0.00029472, 0.00029472,
0.00029472],
[0.00029464, 0.00029464, 0.00029464, ..., 0.00029464, 0.00029464,
0.00029464]])
```

Explicación

```
pi0_norm = array([0.00023111, 0.00231107, 0.00023111, ..., 0.00023111, 0.00023111, 0.00023111])
```

```
A0_norm = array([[0.00029472, 0.00029472, 0.00029472, ..., 0.00029472, 0.00029472, 0.00029472], [0.00029155, 0.00029155,
0.00029155, ..., 0.00029155, 0.00029155, 0.00029155], [0.00029455, 0.00029455, 0.00029455, ..., 0.00029455, 0.00029455,
0.00029455], ..., [0.00029472, 0.00029472, 0.00029472, ..., 0.00029472, 0.00029472, 0.00029472], [0.00029472, 0.00029472,
0.00029472, ..., 0.00029472, 0.00029472, 0.00029472], [0.00029464, 0.00029464, 0.00029464, ..., 0.00029464, 0.00029464,
0.00029464]])
```

En el vector inicial **pi0**, observemos que no aparece ningún valor en cero, lo que indica que el método de suavizar funcionó perfectamente, al igual que en la matriz **A0**, lo que permite una **distribución de probabilidad**.

Espacio logarítmico

Como vimos en la teoría, estas probabilidades pueden tener un desbordamiento por debajo, ya que se aproximan a cero, entonces, para evitar errores computacionales, usaremos el espacio logarítmico.

```
In [... #Borges
log_A0_norm = np.log(A0_norm)
log_pi0_norm = np.log(pi0_norm)
#Benedetti
log_A1_norm = np.log(A1_norm)
log_pil_norm = np.log(pil_norm)
```

```
In [... # Probamos
log_pi0_norm
```

```
Out[... array([-8.37262974, -6.07004465, -8.37262974, ..., -8.37262974,
-8.37262974, -8.37262974])
```

```
In [... # Probamos
log_A0_norm
```

```
Out[... array([[ -8.12946976, -8.12946976, -8.12946976, ..., -8.12946976,
-8.12946976, -8.12946976],
[ -8.14031554, -8.14031554, -8.14031554, ..., -8.14031554,
-8.14031554, -8.14031554],
[ -8.13005904, -8.13005904, -8.13005904, ..., -8.13005904,
-8.13005904, -8.13005904],
...,
[ -8.12946976, -8.12946976, -8.12946976, ..., -8.12946976,
-8.12946976, -8.12946976],
[ -8.12946976, -8.12946976, -8.12946976, ..., -8.12946976,
-8.12946976, -8.12946976],
[ -8.12976445, -8.12976445, -8.12976445, ..., -8.12976445,
-8.12976445, -8.12976445]])
```

Pre-análisis

Vamos a revisar diferentes elementos que nos permitan entender mejor lo que desarrollamos

```
In [...] # Conteo de etiquetas de clase 0 (Borges) en Y_train
count_Y_0 = sum(y == 0 for y in Y_train)
# Conteo de etiquetas de clase 1 (Benedetti) en Y_train
count_Y_1 = sum(y == 1 for y in Y_train)
# Cantidad total de ejemplos de entrenamiento
total = len(Y_train)
# Probabilidad a priori de la clase 0
p0 = count_Y_0 / total
# Probabilidad a priori de la clase 1
p1 = count_Y_1 / total
# Logaritmo de la clase a priori 0
log_p0 = np.log(p0)
# Logaritmo de la clase a priori 1
log_p1 = np.log(p1)

display(HTML(f'''
Se encontró {count_Y_0} etiquetas de clase 0, <b style='color:fuchsia;'>Borges</b>,<br>
Se encontró {count_Y_1} etiquetas de clase 1, <b style='color:skyblue;'>Benedetti</b>,<br>
para un total de <b style='color:red;'>{total}</b> ejemplos de entrenamiento.
<hr>
Las probabilidades a priori serían las siguientes:<br>
<table style="border: 1px solid black; border-collapse: collapse;">
  <tr>
    <td style="border: 1px solid black; padding: 5px;">Borges</td>
    <td style="border: 1px solid black; padding: 5px;">{p0}</td>
  </tr>
  <tr>
    <td style="border: 1px solid black; padding: 5px;">Benedetti</td>
    <td style="border: 1px solid black; padding: 5px;">{p1}</td>
  </tr>
</table>
<hr>
Como usamos el espacio logarítmico, estas serían las probabilidades reales de encontrar un texto de la clase 0 o 1:
<table style="border: 1px solid black; border-collapse: collapse;">
  <tr>
    <td style="border: 1px solid black; padding: 5px;">Borges</td>
    <td style="border: 1px solid black; padding: 5px;">{log_p0}</td>
  </tr>
  <tr>
    <td style="border: 1px solid black; padding: 5px;">Benedetti</td>
    <td style="border: 1px solid black; padding: 5px;">{log_p1}</td>
  </tr>
</table>
<hr>
'''))
```

Se encontró 934 etiquetas de clase 0, **Borges**,
Se encontró 1288 etiquetas de clase 1, **Benedetti**,
para un total de **2222** ejemplos de entrenamiento.

Las probabilidades a priori serían las siguientes:

Borges	0.42034203420342037
Benedetti	0.5796579657965797

Como usamos el espacio logarítmico, estas serían las probabilidades reales de encontrar un texto de la clase 0 o 1:

Borges	-0.8666865319707326
Benedetti	-0.5453170635352763

Construcción del Clasificador

```
In [... # Creamos una clase
class Classifier:
    # Constructor
    def __init__(self, log_As, log_pis, log_apriors):
        self.log_As = log_As
        self.log_pis = log_pis
        self.log_apriors = log_apriors
        # número de clases
        self.k = len(log_apriors)




    # Método de verosimilitud
    def _compute_log_likelihood(self, input_, class_):
        log_A = self.log_As[class_]
        log_pi = self.log_pis[class_]
        #Repetimos lo hecho en el ejemplos de creación de la matriz
        last_index = None
        log_prob = 0
        #Recorremos la entrada del usuario
        for index in input_:
            if last_index is None:
                #Primer token en la secuencia
                log_prob += log_pi[index]
            else:
                #Calculamos la probabilidad de transición del a palabra anterior a la actual
                log_prob += log_A[last_index, index]
                #Actualizamos el index para la próxima iteración
                last_index = index
        return log_prob

    # Función de predicción
    def predict(self, inputs):
        predictions = np.zeros(len(inputs))
        for i, input_ in enumerate(inputs):
            # Calcula los logaritmos de las probabilidades posteriores para cada clase
            posteriors = [self._compute_log_likelihood(input_, c) + self.log_apriors[c] \
                          for c in range(self.k)]
            #Elige la clase de mayor probabilidad posterior como la predicción
            pred = np.argmax(posteriors)
            predictions[i] = pred
        return predictions
```

Explicación

1 Constructor (init) ¿Qué parámetros recibe?

log_As: Matrices de probabilidades de transición entre palabras en logaritmo. log_pis: Probabilidades iniciales de cada palabra en logaritmo. log_apriors: Probabilidades previas (prior) de cada clase en logaritmo. self.k: Número total de clases.

 ¿Por qué usa logaritmos?  Evita problemas de underflow cuando se multiplican muchas probabilidades pequeñas.  Convierte productos en sumas, lo que hace más fácil la optimización.

2 Método _compute_log_likelihood (Cálculo de Verosimilitud)

 ¿Qué hace?

Calcula la log-verosimilitud de una secuencia (input_) dada una clase (class_). Usa la probabilidad inicial de la primera palabra (log_pi[index]). Luego, suma las probabilidades de transición entre palabras (log_A[last_index, index]). Retorna log_prob, que indica qué tan probable es la secuencia dada la clase.

3 Método predict (Clasificación)

 ¿Qué hace?

Inicializa predictions con ceros (un array para almacenar las predicciones).

Para cada entrada en inputs: Calcula las log-verosimilitudes para cada clase. Suma la probabilidad previa (prior) log_apriors[c] de cada clase. Elige la clase con mayor probabilidad posterior usando np.argmax(). Devuelve predictions, que contiene las clases predichas.

Objeto de la clase Clasifier

```
In [...] # Creamos un objeto de la clase Clasifier para llamar los métodos del clasificador
clf = Classifier([log_A0_norm, log_A1_norm], [log_pi0_norm, log_pi1_norm], [log_p0, log_p1])
```

Explicación

La clase Classifier, recibe 3 parámetros en su constructor, es decir 3 atributos. En su orden estos atributos son:

- 1 [log_A0_norm, log_A1_norm] que serán los argumentos de log_As
- 2 [log_pi0_norm, log_pi1_norm] que serán los argumentos de log_pis
- 3 [log_p0, log_p1] que serán los argumentos de log_apriors

Es decir,

- 1 Las matrices de transición normalizadas
- 2 Los vectores con los valores iniciales o estados iniciales
- 3 Las probabilidades de cada clase utilizando el espacio logarítmico

Predicción

```
In [...] # Llamamos al método predict (Datos de entrenamiento: Aprox 1.0)
P_train = clf.predict(X_train_int)
# Mostramos la predicción con la muestra de entrenamiento
print(f'Accuraci Train: {np.mean(P_train == Y_train)}')
```

Accuraci Train: 0.9932493249324933

```
In [...] len(X_test_int)
```

Out[...] 247

```
In [...] # Llamamos al método predict (Datos de prueba)
P_test = clf.predict(X_test_int)
# Mostramos la predicción con la muestra de prueba
print(f'Accuraci Test: {np.mean(P_test == Y_test)}')
```

Accuraci Test: 0.7530364372469636

Probamos con textos nuevos

Textos en los estilos de Borges y Benedetti

Le solicitamos a ChatGPT4 que nos generara textos con escritura similar a cada uno de nuestros poetas, y esto fue lo que nos entregó.

Estilo Borges

*En el vasto archivo del tiempo,
donde las sombras del ayer se funden
con las visiones inciertas del mañana,
el hombre camina por un laberinto
de palabras y azares.
Sabe, aunque lo olvida a menudo,
que su destino está tejido con hilos invisibles,
urdidos por manos que jamás verá.*

*Alguna tarde,
quizá en la penumbra de una biblioteca
o en la geometría secreta de un sueño,
descubrirá que su vida no ha sido sino el eco de otras vidas,*

*el reflejo de una historia que ya ha sido escrita
en un idioma remoto y perfecto.*

Estilo Benedetti

*La ciudad despierta con su ritmo de siempre,
entre murmullos de bocacalles
y pasos apurados que no saben bien a dónde van.*

*En un café cualquiera,
un hombre revuelve su taza con la mirada perdida,
tal vez recordando un amor que ya no está,
tal vez soñando con la ternura que aún no llega.*

*Afuera, la vida sigue su curso,
con su cuota justa de olvidos y de esperanzas,
con sus rutinas que a veces duelen y a veces salvan.*

*Porque, al final de cuentas,
lo importante no es cuánto nos golpea el tiempo,
sino con quién elegimos compartirlo.*

Funciones para poner a prueba el nuevo texto

```
In [... def preprocesar_texto(texto):
    texto = texto.lower().translate(str.maketrans('', '', string.punctuation))
    tokens = texto.split()
    return tokens

def convertir_a_indices(tokens, indice_palabras):
    return [indice_palabras.get(token, 0) for token in tokens] # 0 para <unk>

def calcular_probabilidad(tokens_indices, log_pi, log_A):
    if not tokens_indices:
        return -np.inf # Evitar errores con texto vacío
    log_prob = log_pi[tokens_indices[0]] # Probabilidad inicial
    for i in range(len(tokens_indices) - 1):
        log_prob += log_A[tokens_indices[i], tokens_indices[i + 1]] # Probabilidad de transición
    return log_prob

def clasificar_texto(texto, log_pi0_norm, log_A0_norm, log_pi1_norm, log_A1_norm, indice_palabras):
    tokens = preprocesar_texto(texto)
    tokens_indices = convertir_a_indices(tokens, indice_palabras)

    log_prob_borges = calcular_probabilidad(tokens_indices, log_pi0_norm, log_A0_norm)
    log_prob_benedetti = calcular_probabilidad(tokens_indices, log_pi1_norm, log_A1_norm)

    #Conversión de LOG-PROBABILIDADES a PROBABILIDADES
    max_log_prob = max(log_prob_borges, log_prob_benedetti) # Para evitar underflow numérico
    prob_borges = np.exp(log_prob_borges - max_log_prob)
    prob_benedetti = np.exp(log_prob_benedetti - max_log_prob)
    total_prob = prob_borges + prob_benedetti
    prob_borges /= total_prob # Normalizamos
    prob_benedetti /= total_prob

    print(f"Probabilidad Borges: {prob_borges:.4f}")
    print(f"Probabilidad Benedetti: {prob_benedetti:.4f}")

    return "Borges" if prob_borges > prob_benedetti else "Benedetti"
```

Explicación

Este código implementa un clasificador de texto que determina si un texto nuevo se asemeja más a un escrito de Borges o de Benedetti. Lo hace utilizando un modelo basado en cadenas de Markov, donde se calculan probabilidades de transición entre palabras.

1 preprocesar_texto(texto)

Objetivo: Convierte el texto en minúsculas, elimina signos de puntuación y lo divide en palabras (tokens).

Proceso:

1. `texto.lower()` → Convierte todo el texto a minúsculas.
2. `.translate(str.maketrans("", "", string.punctuation))` → Elimina la puntuación.
3. `.split()` → Divide el texto en palabras, generando una lista de tokens.

2 convertir_a_indices(tokens, indice_palabras)

Objetivo: Convierte cada palabra (token) en un índice numérico basado en un diccionario `indice_palabras`.

Proceso:

1. Usa `indice_palabras.get(token, 0)` para obtener el índice de cada palabra en el diccionario.
2. Si la palabra no está en el diccionario, se asigna 0 (se usa para , palabras desconocidas).

3 calcular_probabilidad(tokens_indices, log_pi, log_A)

Objetivo: Calcula la log-probabilidad de una secuencia de palabras en base a un modelo de cadenas de Markov.

Proceso:

1. Si `tokens_indices` está vacío, retorna `-np.inf` (evita errores).
2. Toma la log-probabilidad inicial de la primera palabra: `log_pi[tokens_indices[0]]`.
3. Para cada par de palabras consecutivas en la secuencia, suma la log-probabilidad de transición `log_A`.

4 clasificar_texto(...)

Objetivo: Clasifica un texto como "Borges" o "Benedetti" en función de su probabilidad de generación en cada modelo.

Proceso:

1. Preprocesa el texto → Obtiene los tokens.
2. Convierte los tokens a índices → Transforma palabras en números.
3. Calcula log-probabilidades para ambos modelos (Borges y Benedetti).
4. Convierte log-probabilidades a probabilidades reales para interpretación:
 - Se resta el máximo log-probabilidad para evitar problemas numéricos (underflow).
 - Se aplica `np.exp(log_prob - max_log_prob)` para convertir log a probabilidad.
 - Se normaliza dividiendo entre la suma total.
5. Imprime las probabilidades y retorna la clasificación.

Ingreso del nuevo texto - Similar a Borges

```
In [... texto_prueba = """
En la vastedad de la memoria,
donde los ecos de los días transcurridos
se entrelazan con los espejismos del porvenir,
el hombre camina por un laberinto
de palabras y azares.

Sabe, aunque lo olvida a menudo,
que su destino está tejido con hilos invisibles,
urdidos por manos que jamás verá.

Alguna tarde,
quizá en la penumbra de una biblioteca
o en la geometría secreta de un sueño,
descubrirá que su vida no ha sido sino el eco de otras vidas,
el reflejo de una historia que ya ha sido escrita
en un idioma remoto y perfecto.
"""

resultado = clasificar_texto(texto_prueba, log_pi0_norm, log_A0_norm, log_pi1_norm, log_A1_norm, indice_palabras)
print(f"Clasificación: {resultado}")
```

Probabilidad Borges: 1.0000
Probabilidad Benedetti: 0.0000
Clasificación: Borges

Ingreso del nuevo texto - Similar a Benedetti

```
In [... texto_prueba = """
La ciudad despierta con su ritmo de siempre,
entre murmullos de bocacalles
y pasos apurados que no saben bien a dónde van.

En un café cualquiera,
un hombre revuelve su taza con la mirada perdida,
tal vez recordando un amor que ya no está,
tal vez soñando con la ternura que aún no llega.

Afuera, la vida sigue su curso,
con su cuota justa de olvidos y de esperanzas,
con sus rutinas que a veces duelen y a veces salvan.

Porque, al final de cuentas,
lo importante no es cuánto nos golpea el tiempo,
sino con quién elegimos compartirlo.
"""
resultado = clasificar_texto(texto_prueba, log_pi0_norm, log_A0_norm, log_pi1_norm, log_A1_norm, indice_palabras)
print(f"Clasificación: {resultado}")

Probabilidad Borges: 0.0028
Probabilidad Benedetti: 0.9972
Clasificación: Benedetti
```

Guardar como .py

```
In [... #!/jupyter nbconvert --to script Proyecto8.ipynb
```

Conclusiones

Se puede observar como los datos de prueba, permiten clasificar los textos con un 75% de precisión, de tal manera que al poner textos de estos poetas, fácilmente podrá indicar quien lo escribió.

Mg. Luis Felipe Bustamante Narváez