PROYECTO FINAL RECUPERACIÓN DE LA INFORMACIÓN

Huerta Aguilar, Jesús., Huitzil Juárez, Guadalupe Quetzalli.,
Pérez Sánchez, Jasmine., Ruiz Ramírez, Gabino.
Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias De La Computación
Av. San Claudio, Bulevard 14 surCd. Universitaria 72592 Puebla, Mexico
Fecha: 19 de mayo 2024

Resumen. El presente proyecto pretende desarrollar un sistema que permita analizar los discursos presidenciales, conferencias y otros textos políticos desde el sexenio de Carlos Salinas de Gortari hasta el sexenio de Andrés Manuel López Obrador. El sistema que proporciona este proyecto son estadísticas generales, gráficas de dispersión léxica, modelos de series temporales, representaciones en nubes de palabras y gráficos de barras para visualizar y comparar temas abordados por diferentes presidentes y partidos políticos, esto cosificando las corrientes en neoliberales y humanistas. Permitiendo así la caracterización ideológica de cada texto recabado. Este proyecto tiene como uno de sus objetivos permitir a investigadores identificar patrones y tendencias en la política a través del tiempo.

Palabras clave: Gráficas, México, dispersión léxica, partidos políticos, estadísticas, análisis, ideología, neoliberal, humanismo.

I. INTRODUCCIÓN

En el ámbito de la ciencia política, el análisis de textos se ha convertido en una metodología esencial para examinar los discursos y manifiestos de los partidos políticos. Con el progreso de las técnicas automatizadas de procesamiento de texto, los investigadores tienen la capacidad de explorar grandes cantidades de datos textuales para identificar características ocultas y patrones discursivos. Este proyecto tiene como objetivo desarrollar un sistema automatizado para el análisis de textos políticos, centrándose inicialmente en los informes presidenciales de México desde la administración de Carlos Salinas de Gortari hasta el presente.

El objetivo principal de este proyecto es proporcionar una herramienta que permita analizar de manera detallada y concisa los discursos de distintos presidentes y partidos

políticos. Entre las funcionalidades previstas se incluyen la generación de estadísticas textuales generales, la creación de gráficas de dispersión léxica, modelos de series temporales para observar tendencias a lo largo del tiempo, nubes de palabras para una visualización rápida y diagramas de barras para la comparación de temas específicos. Además, el sistema permitirá la caracterización ideológica de los textos, facilitando la identificación de diferencias entre discursos neoliberales y humanistas.

Estas herramientas permitirán a los investigadores realizar un análisis detallado de los textos políticos y ofrecerán información valiosa sobre las diferencias y similitudes en las propuestas y enfoques de diversos partidos y líderes políticos. El objetivo es contribuir a una comprensión más profunda del discurso político en México y apoyar la toma de decisiones informadas en los ámbitos político y académico.

II. OBJETIVO Y PLANTEAMIENTO DEL PROBLEMA

Desarrollar un sistema automatizado de análisis de textos políticos que permita identificar y comparar características discursivas, temas predominantes y patrones ideológicos en los discursos presidenciales y otros textos políticos de México, con el fin de proporcionar herramientas analíticas que faciliten una comprensión profunda del discurso político y apoyen la toma de decisiones informadas en los ámbitos político y académico.

III. DESARROLLO EXPERIMENTAL

ALGORITMO DE CLASIFICACIÓN UTILIZADO

En este proyecto, se ha implementado un algoritmo de clasificación utilizando una red neuronal basada en Long Short-Term Memory (LSTM). Las LSTM son una variante de las redes neuronales recurrentes (RNN) que son capaces de aprender dependencias a largo plazo en secuencias de datos, lo que las hace especialmente útiles para tareas de procesamiento de lenguaje natural (NLP).

El modelo de clasificación se construyó utilizando la biblioteca Keras con un backend de TensorFlow. La arquitectura del modelo se compone de las siguientes capas:

- **Capa de Embeddings:** Esta capa convierte las palabras en vectores de características densos de dimensión fija. Este proceso permite que el modelo capture relaciones semánticas entre las palabras. En este caso, se usa la función Embedding para transformar las secuencias de texto en vectores de embeddings.

- **Capa de Dropout Espacial:** Se añade una capa de SpatialDropout1D para ayudar a prevenir el sobreajuste. Esta técnica regulariza los datos de embeddings al establecer aleatoriamente algunos de los valores en cero durante el entrenamiento.

- **Capa LSTM:** Esta es la capa central del modelo. Las LSTM son capaces de recordar patrones en los datos de secuencias a lo largo de periodos de tiempo extendidos. La capa LSTM en este modelo tiene 50 unidades.

- **Capa Densa con Activación Sigmoide:** La capa final es una capa densa (Dense) con una única neurona y una función de activación sigmoide. Esta configuración es adecuada para problemas de clasificación binaria, ya que la función sigmoide produce una probabilidad de pertenencia a una de las dos clases.

El modelo se compila con la función de pérdida binary_crossentropy, que es adecuada para problemas de clasificación binaria. El optimizador utilizado es Adam con una tasa de aprendizaje de 0.0001. La métrica utilizada para evaluar el rendimiento del modelo es la accuracy.

```
modelo.compile(
    loss='binary_crossentropy',
    optimizer=Adam(learning_rate=0.0001),
    metrics=['accuracy']
)
```

El modelo se entrena utilizando los datos de entrenamiento (X_train, y_train) durante 12 épocas con un tamaño de lote (batch_size) de 32. También se utilizan datos de validación (X_val, y_val) para monitorear el rendimiento del modelo durante el entrenamiento.

```
modelo.fit(
    X_train, y_train,
    epochs=12,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=2
)
```

MODELO DE RED NEURONAL

Para este proyecto utilizamos un modelo de aprendizaje profundo para clasificar discursos en dos categorías: Humanismo y neoliberalismo. El flujo de trabajo incluye la carga de datos, preprocesamiento, entrenamiento del modelo, y la capacidad de guardar y cargar el modelo junto con su tokenizador y codificador de etiquetas. Además, se proporciona una función para clasificar nuevos documentos.

- **entrenar_y_guardar_modelo():** Esta función entrena un modelo de aprendizaje profundo y luego guarda el modelo entrenado, el tokenizador y el codificador de etiquetas en archivos especificados por el usuario.
  Algoritmo de Clasificación: Utiliza un modelo de red neuronal con una capa LSTM (Long Short-Term Memory) para la clasificación binaria.

```
    def entrenar_y_guardar_modelo():
        modelo, tokenizer, label_encoder = entrenar_modelo()
        nombre_modelo = simpledialog.askstring("Guardar modelo", "Ingrese el nombre del
archivo para guardar el modelo:")
        save_model(modelo, f"{nombre_modelo}_modelo.h5")
        with open(f"{nombre_modelo}_tokenizer.pickle", 'wb') as handle:
            pickle.dump(tokenizer, handle, protocol=pickle.HIGHEST_PROTOCOL)
        with open(f"{nombre_modelo}_label_encoder.pickle", 'wb') as handle:
            pickle.dump(label_encoder, handle, protocol=pickle.HIGHEST_PROTOCOL)
        return modelo, tokenizer, label_encoder
```

- **Función cargar_modelo():** Permite cargar un modelo previamente guardado junto con su tokenizador y codificador de etiquetas desde archivos seleccionados por el usuario.
  Para esto le pide al usuario un archivo del modelo para cargarlo junto el tokenizador y el codificador de etiquetas desde los archivos correspondientes.

```
    def cargar_modelo():
        filepath = filedialog.askopenfilename(filetypes=[("H5 files", "*.h5")])
        if filepath:
            modelo = load_model(filepath)
            tokenizer_path = filepath.replace("_modelo.h5", "_tokenizer.pickle")
            label_encoder_path = filepath.replace("_modelo.h5", "_label_encoder.pickle")
            with open(tokenizer_path, 'rb') as handle:
                tokenizer = pickle.load(handle)
            with open(label_encoder_path, 'rb') as handle:
                label_encoder = pickle.load(handle)
            return modelo, tokenizer, label_encoder
        else:
            return None, None, None
```

- **entrenar_modelo():** Entrena un modelo LSTM para la clasificación de discursos en las categorías de humanismo y neoliberalismo.
  El primer paso implica cargar los discursos de dos categorías diferentes: humanismo y neoliberalismo. Esto se logra utilizando la función cargar_discursos() Los discursos se cargan en dos listas separadas: discursos_humanismo y discursos_neoliberalismo. Luego, se extraen los

textos de los discursos de ambas categorías y se almacenan en las listas textos_humanismo y textos_neoliberalismo, respectivamente.
Una vez que se tienen los textos de ambas categorías, se procede a preprocesarlos. Esto implica realizar operaciones como eliminar puntuaciones, convertir el texto a minúsculas y eliminar palabras irrelevantes o stopwords. Después del preprocesamiento, se obtienen los textos preprocesados, que se almacenan en la lista textos_preprocesados.

```
def entrenar_modelo():
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    textos_humanismo = [texto[1] for texto in discursos_humanismo]
    textos_neoliberalismo = [texto[1] for texto in discursos_neoliberalismo]
    textos = textos_humanismo + textos_neoliberalismo
    etiquetas = ['humanismo'] * len(textos_humanismo) + ['neoliberalismo'] *
len(textos_neoliberalismo)
    textos_preprocesados = [' '.join(preprocesar_texto(texto)) for texto in textos]
```

- **Tokenización y Secuenciación de Textos:** En este paso, se utiliza la clase Tokenizer de Keras para convertir los textos preprocesados en secuencias de números enteros. Primero, se crea un objeto Tokenizer con un límite máximo de palabras (10000) y un token especial para palabras fuera del vocabulario Luego, se ajusta el tokenizador a los textos preprocesados utilizando el método fit_on_texts(), que construye el índice de palabras internas. Después de ajustar el tokenizador, se convierten los textos preprocesados en secuencias de enteros utilizando el método texts_to_sequences(). Estas secuencias se almacenan en la lista secuencias.

```
max_words = 10000
    max_sequence_length = 100
    tokenizer = Tokenizer(num_words=max_words, oov_token="<OOV>")
    tokenizer.fit_on_texts(textos_preprocesados)
    secuencias = tokenizer.texts_to_sequences(textos_preprocesados)
    secuencias_padded = pad_sequences(secuencias, maxlen=max_sequence_length,
padding='post', truncating='post')
```

- **Padding de Secuencias:** Dado que las secuencias de enteros pueden tener longitudes diferentes, es necesario igualarlas para que todas tengan la misma longitud. Esto se logra mediante el relleno de las secuencias con ceros (padding). La función pad_sequences() de Keras se utiliza para realizar este paso. Las secuencias se rellenan con ceros al final (padding='post') y se truncan si exceden la longitud máxima (truncating='post'). Las secuencias rellenadas se almacenan en la variable secuencias_padded.

```
max_words = 10000
    max_sequence_length = 100
    tokenizer = Tokenizer(num_words=max_words, oov_token="<OOV>")
    tokenizer.fit_on_texts(textos_preprocesados)
    secuencias = tokenizer.texts_to_sequences(textos_preprocesados)
    secuencias_padded = pad_sequences(secuencias, maxlen=max_sequence_length,
padding='post', truncating='post')
```

- **Codificación de Etiquetas:** En este paso, las etiquetas de las categorías se codifican en formato numérico. Se utiliza la clase LabelEncoder de scikit-learn para realizar esta tarea. Las etiquetas originales ('humanismo' y 'neoliberalismo') se convierten en valores numéricos, donde 'humanismo' puede ser codificado como 0 y 'neoliberalismo' como 1. Las etiquetas codificadas se almacenan en la variable etiquetas_encoded.

```
label_encoder = LabelEncoder()
    etiquetas_encoded = label_encoder.fit_transform(etiquetas)
    X_train, X_val, y_train, y_val = train_test_split(secuencias_padded,
etiquetas_encoded, test_size=0.2, random_state=42)
```

- **División de Datos**: Una vez que se han preparado los datos de texto y las etiquetas, se dividen en conjuntos de entrenamiento y validación. Esto se logra utilizando la función train_test_split() de scikit-learn. Se reserva el 20% de los datos para validación, mientras que el 80% restante se utiliza para el entrenamiento. Los conjuntos de entrenamiento y validación se almacenan en las variables X_train, X_val, y_train y y_val.

- **Construcción del Modelo LSTM:** Finalmente, se construye el modelo de aprendizaje profundo utilizando Keras. El modelo se compila utilizando la función de pérdida binaria 'binary_crossentropy' y el optimizador Adam con una tasa de aprendizaje específica. Luego, se entrena el modelo utilizando los datos de entrenamiento (X_train y y_train) durante 12 épocas con un tamaño de lote de 32. Los datos de validación se utilizan para monitorear el rendimiento del modelo durante el entrenamiento.

```
modelo = Sequential()
    modelo.add(Embedding(max_words, 128, input_length=max_sequence_length))
    modelo.add(SpatialDropout1D(0.2))
    modelo.add(LSTM(50))
    modelo.add(Dropout(0.5))
    modelo.add(Dense(1, activation='sigmoid'))

    modelo.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.0001),
metrics=['accuracy'])
    modelo.fit(X_train, y_train, epochs=12, batch_size=32, validation_data=(X_val,
y_val), verbose=2)

    return modelo, tokenizer, label_encoder
```

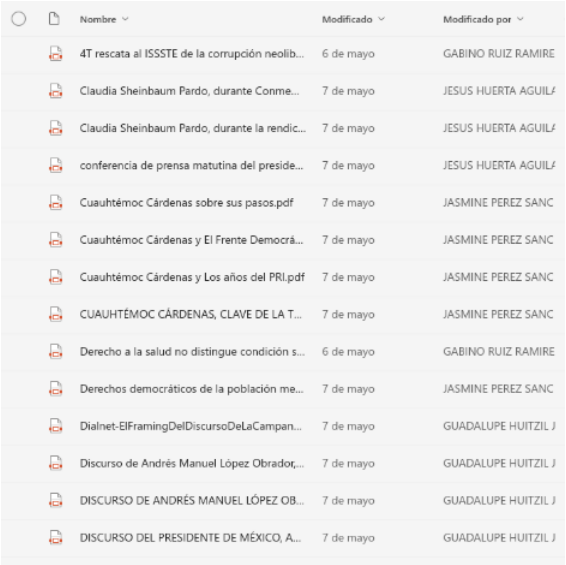
- **Función clasificar_nuevo_documento():** se encarga de clasificar un nuevo documento utilizando el modelo de aprendizaje profundo previamente entrenado. Lo primero que hace la función es recibir como entrada la ruta del archivo del nuevo documento que se desea clasificar. Utiliza esta ruta para abrir y leer el contenido del archivo de texto. Una vez que se ha leído el contenido del documento, se realiza el mismo proceso de preprocesamiento que se aplicó durante el entrenamiento del modelo. El texto preprocesado se convierte en una secuencia de números enteros utilizando el tokenizador. La secuencia de enteros se rellena con ceros si es necesario para que tenga la misma longitud que las secuencias utilizadas durante el entrenamiento. Esto asegura que la entrada tenga el formato esperado por el modelo.

```
def clasificar_nuevo_documento(filepath, modelo, tokenizer, label_encoder,
max_sequence_length):
    with open(filepath, 'r', encoding='utf-8') as file:
        nuevo_texto = file.read()

    nuevo_texto_preprocesado = ' '.join(preprocesar_texto(nuevo_texto))
    nueva_secuencia = tokenizer.texts_to_sequences([nuevo_texto_preprocesado])
    nueva_secuencia_padded = pad_sequences(nueva_secuencia, maxlen=max_sequence_length,
padding='post', truncating='post')
    prediccion = modelo.predict(nueva_secuencia_padded)
    categoria = label_encoder.inverse_transform([int(round(prediccion[0][0]))])[0]
    messagebox.showinfo("Clasificación del Documento", f"El documento
'{os.path.basename(filepath)}' es clasificado como: {categoria}")
```

FUNCIONES

1. Para realizar este proyecto, primero recabamos una serie de archivos con los que podemos trabajar. Estos archivos abarcan enfoques humanistas y neoliberales desde el sexenio de Salinas de Gortari hasta el de Andrés Manuel López Obrador. Entre los materiales recopilados se incluyen conferencias, informes de Gobierno, entrevistas y otros documentos. Además, los archivos fueron clasificados según su corriente de pensamiento para facilitar su análisis. En total obtuvimos 39 documentos de humanismo y 39 documentos para neoliberalismo.



Nombre	Modificado	Modificado por
4T rescata al ISSSTE de la corrupción neolib...	6 de mayo	GABINO RUIZ RAMIRE
Claudia Sheinbaum Pardo, durante Conme...	7 de mayo	JESUS HUERTA AGUILA
Claudia Sheinbaum Pardo, durante la rendic...	7 de mayo	JESUS HUERTA AGUILA
conferencia de prensa matutina del preside...	7 de mayo	JESUS HUERTA AGUILA
Cuauhtémoc Cárdenas sobre sus pasos.pdf	7 de mayo	JASMINE PEREZ SANC
Cuauhtémoc Cárdenas y El Frente Democrá...	7 de mayo	JASMINE PEREZ SANC
Cuauhtémoc Cárdenas y Los años del PRI.pdf	7 de mayo	JASMINE PEREZ SANC
CUAUHTÉMOC CÁRDENAS, CLAVE DE LA T...	7 de mayo	JASMINE PEREZ SANC
Derecho a la salud no distingue condición s...	6 de mayo	GABINO RUIZ RAMIRE
Derechos democráticos de la población me...	7 de mayo	JASMINE PEREZ SANC
Dialnet-ElFramingDelDiscursoDeLaCampan...	7 de mayo	GUADALUPE HUITZIL J
Discurso de Andrés Manuel López Obrador,...	7 de mayo	GUADALUPE HUITZIL J
DISCURSO DE ANDRÉS MANUEL LÓPEZ OB...	7 de mayo	GUADALUPE HUITZIL J
DISCURSO DEL PRESIDENTE DE MÉXICO, A...	7 de mayo	GUADALUPE HUITZIL J

Imagen 1: Documentos recabados

1. Después, procedimos a etiquetar cada archivo con el nombre del presidente correspondiente, la fecha del archivo, y convertimos todos los documentos (PDF, Word, etc.) en archivos de texto. Esta conversión se realizó eliminando imágenes y caracteres especiales para asegurar un formato uniforme y facilitar el análisis.

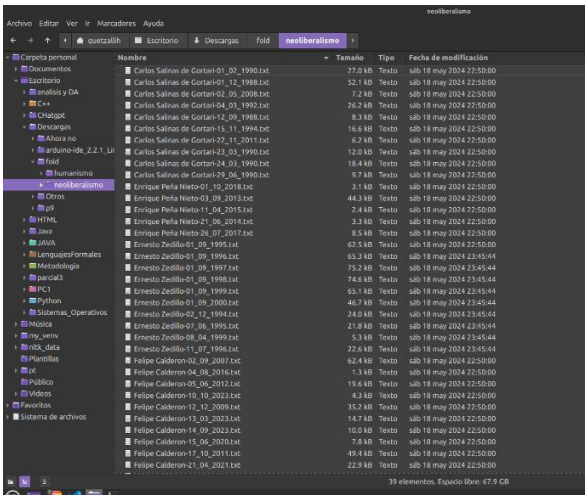


Imagen 2: Renombramiento de documentos

2. Luego, avanzamos a cargar y procesar los textos de cada corriente, asegurándonos de que estuvieran en formato .txt. Almacenamos el contenido de cada archivo en una lista llamada "recursos" y posteriormente procesamos cada texto para su análisis. Para ello, realizamos las siguientes operaciones: convertimos el texto a minúsculas, eliminamos acentos, tokenizamos el texto, filtramos palabras no alfabéticas, palabras vacías y palabras cortas, y reducimos las palabras a su forma base.

```
def cargar_discursos(directorio):
    discursos = []
    for filename in os.listdir(directorio):
        if filename.endswith(".txt"):
            with open(os.path.join(directorio, filename), 'r', encoding='utf-8') as file:
                discursos.append((filename, file.read()))
    return discursos

stop_words = set(stopwords.words('spanish'))
lemmatizer = WordNetLemmatizer()

def preprocesar_texto(texto):
    texto = texto.lower()
    texto = unidecode.unidecode(texto)
    tokens = word_tokenize(texto)
    tokens = [word for word in tokens if word.isalpha()]
    tokens = [word for word in tokens if word not in stop_words]
    tokens = [word for word in tokens if len(word) > 3]
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    return tokens
```

3. El siguiente paso implicó la extracción del vocabulario único de una lista de textos preprocesados. Para ello, inicializamos un conjunto vacío para almacenar este vocabulario y recorrimos cada texto preprocesado. En cada iteración, tokenizamos el texto y agregamos las palabras al conjunto de vocabulario.

```python
def extraer_vocabulario(textos_preprocesados):
    vocabulario = set()
    for texto in textos_preprocesados:
        if isinstance(texto, list):
            texto = ' '.join(texto)
        palabras = word_tokenize(texto)
        vocabulario.update(palabras)
    return vocabulario

def guardar_vocabulario(vocabulario, archivo_salida):
    with open(archivo_salida, 'w', encoding='utf-8') as file:
        for palabra in sorted(vocabulario):
            file.write(f"{palabra}\n")
```

El vocabulario extraído se guarda en dos archivos de texto distintos: uno para el vocabulario de la corriente neoliberal y otro para la corriente humanista.

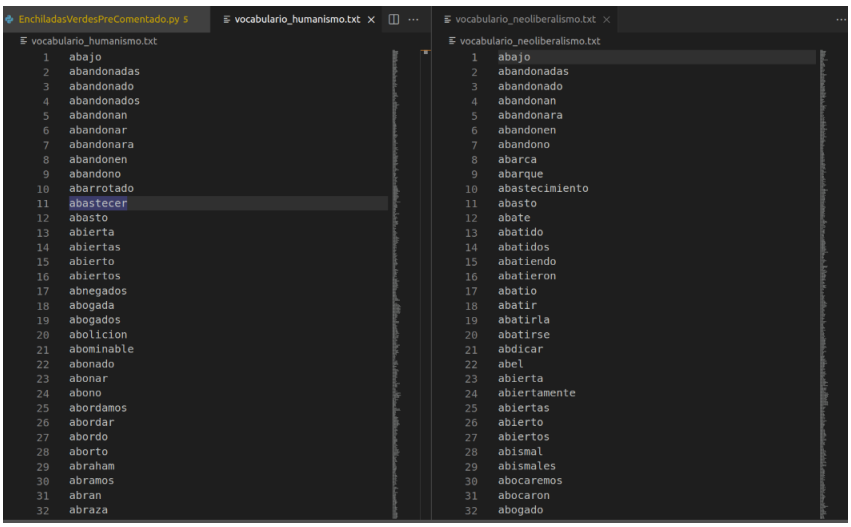


Imagen 3: Vocabulario para humanismo y neoliberalismo

4. Comenzamos abordando las estadísticas, donde calculamos el total de palabras presentes en los textos preprocesados. A partir de este cálculo, determinamos el promedio de palabras por texto, junto con las longitudes máxima y mínima de los mismos. Además, realizamos un análisis para identificar las palabras más frecuentes.

```
def calcular_estadisticas(textos_preprocesados):
    total_palabras = sum(len(texto) for texto in textos_preprocesados)
    promedio_palabras = total_palabras / len(textos_preprocesados)
    longitudes = [len(texto) for texto in textos_preprocesados]
    longitud_maxima = max(longitudes)
    longitud_minima = min(longitudes)

    palabras_frecuentes = Counter([palabra for texto in textos_preprocesados for palabra
in texto]).most_common(10)
    return {
        'total_palabras': total_palabras,
        'promedio_palabras': promedio_palabras,
        'longitud_maxima': longitud_maxima,
        'longitud_minima': longitud_minima,
        'palabras_frecuentes': palabras_frecuentes
    }
```

5. Durante la segunda etapa de nuestra práctica, exploramos la dispersión léxica, que implica analizar la distribución de palabras clave dentro de los documentos preprocesados. Es importante destacar que en esta fase, el usuario nos proporciona las palabras específicas sobre las cuales desea conocer la dispersión léxica en cada corriente. El proceso funciona de la siguiente manera: para cada palabra dada, identificamos los índices de los documentos que contienen dicha palabra y luego generamos un gráfico para visualizar esta dispersión.

```
def graficar_dispersion_lexica(textos_preprocesados, palabras, label):
    plt.figure(figsize=(10, 6))
    for palabra in palabras:
        indices = [i for i, texto in enumerate(textos_preprocesados) if palabra in texto]
        plt.plot(indices, [palabra] * len(indices), '|', label=palabra)
    plt.title(f'Dispersión Léxica - {"Humanismo" if label else "Neoliberalismo"}')
    plt.xlabel('Índice del documento')
    plt.ylabel('Palabra')
    plt.legend()
    plt.show()
```

6. Luego, implementamos una función para graficar las series de tiempo, la cual se encarga de visualizar la evolución de la frecuencia de las palabras proporcionadas por el usuario a lo largo del tiempo en los textos de discursos. Para lograr esto, fue necesario extraer las fechas de los nombres de los textos previamente etiquetados, lo que facilitó y ordenó el proceso. Utilizando estas fechas, se registraron las ocurrencias de cada palabra en los textos correspondientes y se crearon series temporales que luego fueron graficadas.

```
def graficar_series_tiempo(discursos, palabras, titulo):
    series_tiempo = {palabra: [] for palabra in palabras}
    fechas = []

    for filename, texto in discursos:
        fecha = extraer_fecha(filename)
        if fecha:
            fechas.append(fecha)
            tokens = preprocesar_texto(texto)
            conteo = Counter(tokens)
            for palabra in palabras:
                series_tiempo[palabra].append(conteo.get(palabra, 0))

    if fechas and any(series_tiempo.values()):
        fechas, series_tiempo_ordenado = zip(*sorted(zip(fechas,
zip(*[series_tiempo[palabra] for palabra in palabras]))))
        for i, palabra in enumerate(palabras):
            plt.plot(fechas, [serie[i] for serie in series_tiempo_ordenado],
label=palabra)

        plt.xlabel('Fecha')
        plt.ylabel('Frecuencia')
        plt.title(f'Series de Tiempo - {"Humanismo" if titulo else "Neoliberalismo"}')
        plt.legend()
        plt.show()
    else:
        print("No hay datos suficientes para graficar las series de tiempo.")
```

7. Continuamos con la función WordCloud, la cual nos permite crear nubes de palabras a partir de los textos procesados. Para lograrlo, concatenamos todos los textos y generamos una nube de palabras para cada corriente (neoliberal o humanista). Finalmente, configuramos el tamaño, color y fondo de la imagen resultante utilizando esta función.

```
def generar_wordcloud(textos_preprocesados, label):
    texto = ' '.join([' '.join(texto) for texto in textos_preprocesados])
    wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(texto)
    plt.figure(figsize=(10, 6))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.title(f'WordCloud - {"Humanismo" if label else "Neoliberalismo"}')
    plt.axis('off')
    plt.show()
```

8. En cuanto a la función de frecuencia de palabras, esta se encarga de visualizar la frecuencia de las palabras proporcionadas por el usuario en los textos. En primer lugar, contabilizamos las ocurrencias de estas palabras en los textos, para luego generar una gráfica de barras que muestre esta información de manera clara y concisa.

```
def graficar_frecuencia_palabras(textos_preprocesados, palabras, label):
    frecuencias = Counter([palabra for texto in textos_preprocesados for palabra in texto
if palabra in palabras])
    plt.figure(figsize=(10, 6))
    sns.barplot(x=list(frecuencias.keys()), y=list(frecuencias.values()))
    plt.title(f'Frecuencia de Palabras - {"Humanismo" if label else "Neoliberalismo"}')
    plt.xlabel('Palabra')
    plt.ylabel('Frecuencia')
    plt.show()
```

9. A continuación, se presentan una serie de funciones que nos ayudan a mostrar información en pantalla mediante ventanas emergentes, así como funciones generales que nos ayudan a lo largo del código. Estas funciones son:

   Mostrar características generales: Esta función muestra las estadísticas generales de los discursos humanistas o neoliberales en una ventana emergente utilizando Tkinter. Carga los documentos desde ubicaciones específicas, los procesa y calcula estadísticas como el número total de palabras, el promedio de palabras por documento, entre otras, y las muestra en la ventana.

```
def mostrar_caracteristicas_generales(tipo):
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    textos_preprocesados_humanismo = [preprocesar_texto(texto[1]) for texto in
discursos_humanismo]
    textos_preprocesados_neoliberalismo = [preprocesar_texto(texto[1]) for texto in
discursos_neoliberalismo]
    estadisticas_humanismo = calcular_estadisticas(textos_preprocesados_humanismo)
    estadisticas_neoliberalismo =
calcular_estadisticas(textos_preprocesados_neoliberalismo)

    def mostrar_en_ventana(titulo, estadisticas):
        ventana = tk.Toplevel(root)
        ventana.title(titulo)

        texto = f"Características Generales - {titulo}\n\n"
        texto += f"Total de palabras: {estadisticas['total_palabras']}\n"
        texto += f"Promedio de palabras por documento:
{estadisticas['promedio_palabras']}\n"
        texto += f"Longitud del documento más largo: {estadisticas['longitud_maxima']}\n"
        texto += f"Longitud del documento más corto: {estadisticas['longitud_minima']}\n"
        texto += f"Palabras más frecuentes: {estadisticas['palabras_frecuentes']}\n"
        texto += f"Longitudes: {[len(texto) for texto in textos_preprocesados_humanismo]
if tipo == 1 else [len(texto) for texto in textos_preprocesados_neoliberalismo]}\n"
        texto += f"Palabras únicas: {len(set([palabra for texto in
textos_preprocesados_humanismo for palabra in texto])) if tipo == 1 else len(set([palabra
for texto in textos_preprocesados_neoliberalismo for palabra in texto]))}\n"

        label = tk.Label(ventana, text=texto, justify="left", padx=10, pady=10)
        label.pack()

    if tipo == 1:
        mostrar_en_ventana("Humanismo", estadisticas_humanismo)
    else:
        mostrar_en_ventana("Neoliberalismo", estadisticas_neoliberalismo)
```

mostrar_grafico_dispersion_lexica: Genera y muestra un gráfico de dispersión léxica para los discursos seleccionados. Carga y preprocesa los discursos, solicita al usuario las palabras a graficar y muestra la dispersión léxica en función del tipo de discurso.

```
def mostrar_grafico_dispersion_lexica(tipo):
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    textos_preprocesados_humanismo = [preprocesar_texto(texto[1]) for texto in
discursos_humanismo]
    textos_preprocesados_neoliberalismo = [preprocesar_texto(texto[1]) for texto in
discursos_neoliberalismo]
    palabras_a_graficar = pedir_palabras()
    if tipo == 1:
        graficar_dispersion_lexica(textos_preprocesados_humanismo, palabras_a_graficar,
1)
    else:
        graficar_dispersion_lexica(textos_preprocesados_neoliberalismo,
palabras_a_graficar, 0)
```

mostrar_grafico_series_tiempo: Muestra un gráfico de series de tiempo para los discursos humanistas o neoliberales. Carga los discursos, solicita palabras al usuario y genera el gráfico correspondiente.

```
def mostrar_grafico_series_tiempo(tipo):
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    palabras_a_graficar = pedir_palabras()
    if tipo == 1:
        graficar_series_tiempo(discursos_humanismo, palabras_a_graficar, 1)
    else:
        graficar_series_tiempo(discursos_neoliberalismo, palabras_a_graficar, 0
```

mostrar_wordcloud: Genera y muestra una nube de palabras (WordCloud) para los discursos humanistas o neoliberales. Carga y preprocesa los discursos, y luego genera la nube de palabras según el tipo seleccionado.

```
def mostrar_wordcloud(tipo):
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    textos_preprocesados_humanismo = [preprocesar_texto(texto[1]) for texto in
discursos_humanismo]
    textos_preprocesados_neoliberalismo = [preprocesar_texto(texto[1]) for texto in
discursos_neoliberalismo]
    if tipo == 1:
        generar_wordcloud(textos_preprocesados_humanismo, 1)
    else:
        generar_wordcloud(textos_preprocesados_neoliberalismo, 0)
```

mostrar_grafico_frecuencia_palabras: Muestra un gráfico de frecuencia de palabras para los discursos. Carga y preprocesa los discursos, solicita palabras al usuario y genera el gráfico de frecuencia.

```
def mostrar_grafico_frecuencia_palabras(tipo):
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    textos_preprocesados_humanismo = [preprocesar_texto(texto[1]) for texto in
discursos_humanismo]
    textos_preprocesados_neoliberalismo = [preprocesar_texto(texto[1]) for texto in
discursos_neoliberalismo]
    palabras_a_graficar = pedir_palabras()
    if tipo == 1:
        graficar_frecuencia_palabras(textos_preprocesados_humanismo, palabras_a_graficar,
1)
    else:
        graficar_frecuencia_palabras(textos_preprocesados_neoliberalismo,
palabras_a_graficar, 0)
```

vocabulario: Extrae y guarda el vocabulario de los discursos humanistas y neoliberales en archivos de texto. Carga y preprocesa los discursos, extrae el vocabulario y lo guarda en archivos separados para cada tipo de discurso.

```
def vocabulario():
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    textos_preprocesados_humanismo = [preprocesar_texto(texto[1]) for texto in
discursos_humanismo]
    textos_preprocesados_neoliberalismo = [preprocesar_texto(texto[1]) for texto in
discursos_neoliberalismo]
    vocabulario_humanismo = extraer_vocabulario(textos_preprocesados_humanismo)
    vocabulario_neoliberalismo = extraer_vocabulario(textos_preprocesados_neoliberalismo)
    guardar_vocabulario(vocabulario_humanismo, 'vocabulario_humanismo.txt')
    guardar_vocabulario(vocabulario_neoliberalismo, 'vocabulario_neoliberalismo.txt')
    vocabulario_corpus = vocabulario_neoliberalismo.union(vocabulario_humanismo)
    guardar_vocabulario(vocabulario_corpus, 'corpus.txt')
```

cargar_y_clasificar_documento: Permite al usuario seleccionar un archivo de texto y clasificarlo utilizando un modelo previamente entrenado. Abre un cuadro de diálogo para seleccionar el archivo y lo clasifica.

```
def cargar_y_clasificar_documento():
    filepath = filedialog.askopenfilename(filetypes=[("Text files", "*.txt")])
    if filepath:
        clasificar_nuevo_documento(filepath, modelo, tokenizer, label_encoder,
max_sequence_length
```

10. Una función crucial en nuestro proyecto: la entrada por teclado. Esta función facilita la interacción con el usuario y la visualización de resultados. Para esto, le pide al usuario ingresar palabras separadas por comas, las cuales son luego divididas y convertidas en una lista. Es importante mencionar que esta función auxilia a varias funciones principales previamente descritas, entre las cuales se incluyen: mostrar_grafico_dispersion_lexica, mostrar_grafico_series_tiempo, mostrar_wordcloud y mostrar_grafico_frecuencia_palabras.

```
def pedir_palabras():
    root = tk.Tk()
    root.withdraw()
    palabras_str = simpledialog.askstring("Palabras", "Ingrese las palabras separadas por
comas y que estas sean mayores a 3 caracteres:")
    palabras = [palabra.strip() for palabra in palabras_str.split(',')]
    return palabras
```

11. Por último, tenemos la sección de la interfaz gráfica. En esta función, creamos una ventana principal (root) con el título "Sistema de Análisis Político". La interfaz se divide en dos marcos, uno para la categoría de "Humanismo" y otro para la categoría de "Neoliberalismo". Cada marco presenta un título y botones que permiten al usuario interactuar con diversas funciones del sistema, como ver características generales, gráficos de dispersión léxica, series de tiempo, nubes de palabras y gráficos de frecuencia de palabras para cada categoría. A demás de estas funcionalidades específicas, hay un botón diseñado para cargar y clasificar un nuevo documento. La interfaz se ejecuta continuamente en un bucle hasta que el usuario la cierra.

```
root = tk.Tk()
root.title("Sistema de Análisis Político")

frame_humanismo = ttk.Frame(root, padding="10")
frame_humanismo.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

titulo_humanismo = ttk.Label(frame_humanismo, text="Humanismo", font=("Helvetica", 16))
titulo_humanismo.grid(row=0, column=0, columnspan=2, pady=10)

ttk.Button(frame_humanismo, text="Ver Características Generales",
command=lambda: mostrar_caracteristicas_generales(1)  ).grid(row=1, column=0, pady=5)
ttk.Button(frame_humanismo, text="Ver Gráfico de Dispersión Léxica",
command=lambda: mostrar_grafico_dispersion_lexica(1)  ).grid(row=2, column=0, pady=5)
ttk.Button(frame_humanismo, text="Ver Gráficos de Series de Tiempo",
command=lambda: mostrar_grafico_series_tiempo(1)      ).grid(row=3, column=0, pady=5)
ttk.Button(frame_humanismo, text="Ver Wordclouds",
command=lambda: mostrar_wordcloud(1)                  ).grid(row=4, column=0, pady=5)
ttk.Button(frame_humanismo, text="Ver Gráficos de Frecuencia de Palabras",
command=lambda: mostrar_grafico_frecuencia_palabras(1)).grid(row=5, column=0, pady=5)

frame_neoliberalismo = ttk.Frame(root, padding="10")
frame_neoliberalismo.grid(row=0, column=1, sticky=(tk.W, tk.E, tk.N, tk.S))

titulo_neoliberalismo = ttk.Label(frame_neoliberalismo, text="Neoliberalismo",
font=("Helvetica", 16))
titulo_neoliberalismo.grid(row=0, column=0, columnspan=2, pady=10)

ttk.Button(frame_neoliberalismo, text="Ver Características Generales",
command=lambda: mostrar_caracteristicas_generales(0)  ).grid(row=1, column=0, pady=5)
ttk.Button(frame_neoliberalismo, text="Ver Gráfico de Dispersión Léxica",
command=lambda: mostrar_grafico_dispersion_lexica(0)  ).grid(row=2, column=0, pady=5)
ttk.Button(frame_neoliberalismo, text="Ver Gráficos de Series de Tiempo",
command=lambda: mostrar_grafico_series_tiempo(0)      ).grid(row=3, column=0, pady=5)
ttk.Button(frame_neoliberalismo, text="Ver Wordclouds",
command=lambda: mostrar_wordcloud(0)                  ).grid(row=4, column=0, pady=5)
ttk.Button(frame_neoliberalismo, text="Ver Gráficos de Frecuencia de Palabras",
command=lambda: mostrar_grafico_frecuencia_palabras(0)).grid(row=5, column=0, pady=5)

ttk.Button(root, text="Cargar y Clasificar Nuevo Documento",
command=cargar_y_clasificar_documento).grid(row=1, column=0, columnspan=2, pady=20)
root.mainloop()
```

TAMAÑO Y ORGANIZACIÓN DEL CORPUS

El corpus está constituido por la combinación de los vocabularios pertenecientes a dos divisiones de tipos de documentos: el vocabulario humanista y el vocabulario neoliberal. Este corpus incluye todas las palabras distintas que forman parte de ambos vocabularios, permitiendo un análisis comparativo y exhaustivo de los términos utilizados en cada uno.

El vocabulario humanista contiene 10,715 palabras distintas, mientras que el vocabulario neoliberal incluye 12,440 palabras distintas. Por lo tanto, al unir estos dos vocabularios, el corpus resulta en un total de 17,471 palabras distintas. Esta unificación permite un análisis más amplio y detallado, al considerar la diversidad léxica presente en ambos conjuntos de documentos.

Imagen 4: Vocabularios y corpus

Fragmento de código donde es creado el corpus:

```
def vocabulario():
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    textos_preprocesados_humanismo = [preprocesar_texto(texto[1]) for texto in
discursos_humanismo]
    textos_preprocesados_neoliberalismo = [preprocesar_texto(texto[1]) for texto in
discursos_neoliberalismo]
    vocabulario_humanismo = extraer_vocabulario(textos_preprocesados_humanismo)
    vocabulario_neoliberalismo = extraer_vocabulario(textos_preprocesados_neoliberalismo)
    guardar_vocabulario(vocabulario_humanismo, 'vocabulario_humanismo.txt')
    guardar_vocabulario(vocabulario_neoliberalismo, 'vocabulario_neoliberalismo.txt')
    vocabulario_corpus = vocabulario_neoliberalismo.union(vocabulario_humanismo)
    guardar_vocabulario(vocabulario_corpus, 'corpus.txt')
```

INSTALAR CODIGO Y EJECUTAR

Para ejecutar el programa correctamente, es necesario contar inicialmente con el código fuente del clasificador denominado “PFRI.py”. Además, se deben tener las siguientes carpetas:

- **fold:** Esta carpeta contiene dos subcarpetas denominadas "humanismo" y "neoliberalismo". La carpeta "humanismo" incluye los archivos transcritos de informes de políticos humanistas, mientras que la carpeta "neoliberalismo" contiene los archivos transcritos de informes de políticos neoliberales.

- **Pruebas:** En esta carpeta se encuentran los archivos .txt de prueba, los cuales se utilizarán para evaluar el programa una vez que el modelo esté entrenado o cargado, según sea el caso.

Adicionalmente, es fundamental tener descargados los archivos del mejor modelo entrenado hasta el momento, identificado como “*08710_12epochs*”. Los archivos necesarios son:

i. *08710_12epochs_label_encoder.pickle*
ii. *08710_12epochs_modelo.h5*
iii. *08710_12epochs_tokenizer.pickle*

Una vez que el usuario disponga de todos los archivos mencionados, es crucial ajustar las rutas de los directorios en el código a las ubicaciones correspondientes en su computador. Específicamente, se deben modificar las rutas de los archivos de las carpetas "humanismo" y "neoliberalismo". Los demás archivos pueden estar ubicados en cualquier directorio sin afectar el funcionamiento del programa.

Ejemplo: directorios que se deben de modificar por las rutas nuevas del usuario:

```
def mostrar_wordcloud(tipo):
    discursos_humanismo = cargar_discursos(r"A:\Principal\Escritorio\xd\fold\humanismo")
    discursos_neoliberalismo =
cargar_discursos(r"A:\Principal\Escritorio\xd\fold\neoliberalismo")
    textos_preprocesados_humanismo = [preprocesar_texto(texto[1]) for texto in
discursos_humanismo]
    textos_preprocesados_neoliberalismo = [preprocesar_texto(texto[1]) for texto in
discursos_neoliberalismo]
    if tipo == 1:
        generar_wordcloud(textos_preprocesados_humanismo, 1)
    else:
        generar_wordcloud(textos_preprocesados_neoliberalismo, 0)
```

IV. DISCUSIÓN Y RESULTADOS

Los resultados del proyecto final se mostrarán a continuación:

Al iniciar el programa, se le pregunta al usuario si desea cargar un modelo ya entrenado o entrenar uno nuevo. En este caso, ya se ha entrenado el modelo adecuado, el cual muestra un mejor índice de clasificación de documentos, por lo que se optará por cargar un modelo ya entrenado.

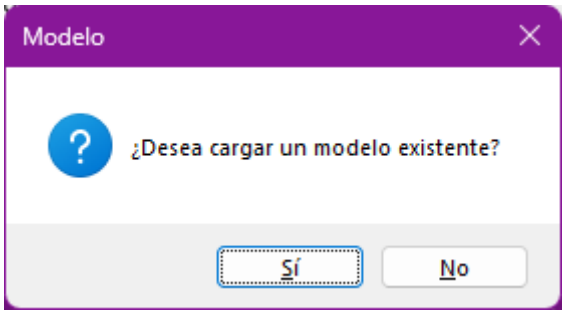


Imagen 5: Elección de carga de modelo

A continuación, se abre una ventana en la que el usuario debe navegar para encontrar y seleccionar el modelo entrenado, el cual debe tener la extensión .h5 o pickle. El sistema permite la búsqueda y selección de estos archivos desde el directorio donde están almacenados. Este modelo previamente entrenado contiene los parámetros y pesos necesarios para realizar la clasificación de documentos de manera eficiente y precisa, evitando así el tiempo y recursos que implicaría entrenar un nuevo modelo desde cero. Una vez seleccionado el modelo, el programa lo carga en memoria y lo prepara para su uso inmediato. El usuario puede entonces proceder a utilizar el modelo para clasificar nuevos documentos, aprovechando el alto rendimiento y la precisión del modelo entrenado. Este enfoque no solo mejora la eficiencia del proceso, sino que también garantiza la consistencia en los resultados de la clasificación de documentos.

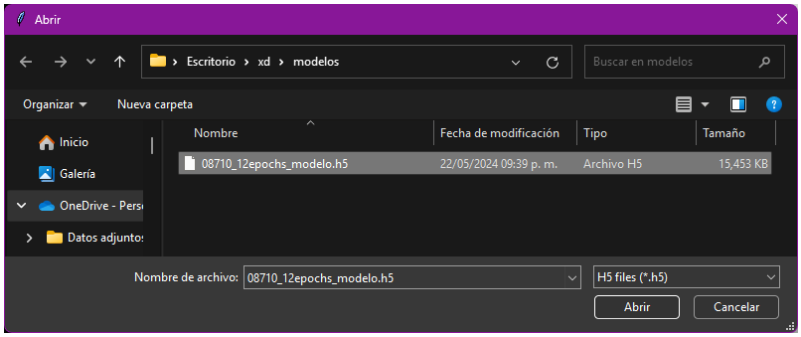


Imagen 6: Carga de modelo

Se despliega una ventana emergente que presenta el sistema de análisis político con un menú tanto para la corriente humanista como para la neoliberal. Cada sección del menú proporciona opciones para acceder a las características generales, gráficos de dispersión léxica, series de tiempo, nubes de palabras y frecuencia de palabras específicas para cada corriente. Además, incluye un botón adicional para clasificar un nuevo documento.

Imagen 7: Menú principal

Cuando se presiona el botón "Características Generales", se despliega una ventana emergente que muestra información detallada, incluyendo el recuento total de palabras, el promedio de palabras por documento, la longitud del documento más largo y más corto, las palabras más frecuentes, la longitud de cada documento y el número de palabras únicas para cada corriente política.

Imagen 8. Salida Características principales

Al hacer clic en el botón "Dispersión Léxica", se abre una ventana emergente que solicita al usuario ingresar las palabras clave sobre las cuales desea visualizar la dispersión. Una vez ingresadas las palabras, se muestra otra ventana emergente que presenta la dispersión de estas palabras en cada documento, permitiendo al usuario observar cómo están distribuidas a lo largo de los textos.

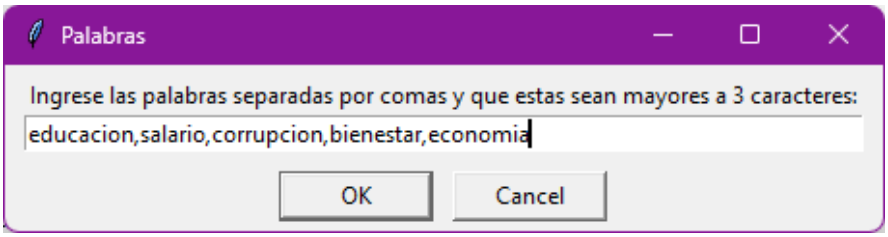


Imagen 9: Entrada de datos

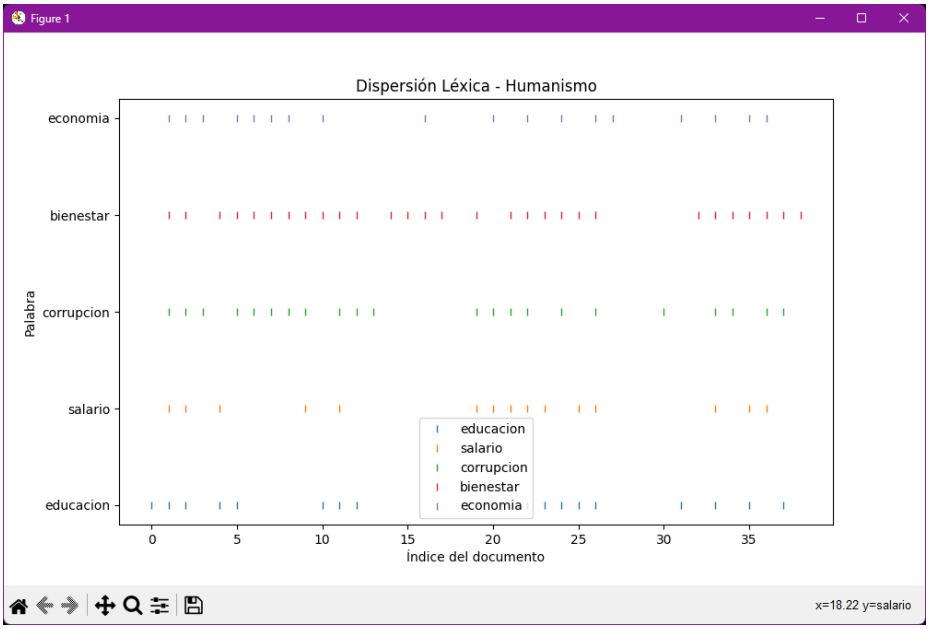


Imagen 10: Salida dispersión léxica

Al pulsar el botón "Series de Tiempo", se solicita al usuario ingresar las palabras deseadas para luego mostrar una ventana emergente que visualiza la serie temporal de estas palabras en los documentos. Esto permite al usuario observar cómo varía la frecuencia de las palabras seleccionadas a lo largo del tiempo en los textos analizados.

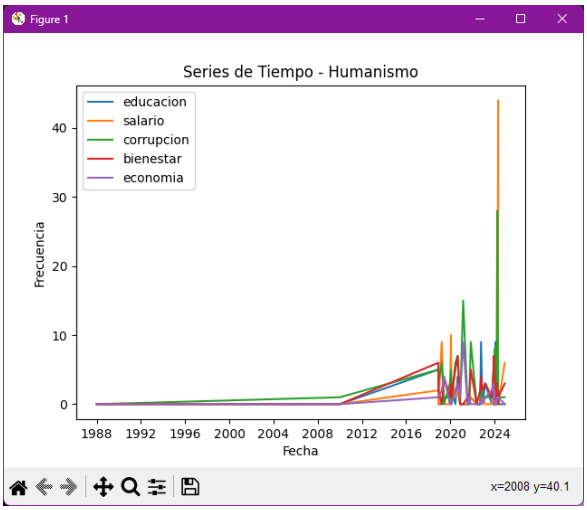


Imagen 11: Salida series de tiempo

Cuando se presiona el botón correspondiente a las "WordClouds", una ventana emergente aparece mostrando una nube de palabras para cada una de las corrientes políticas analizadas. Esto proporciona una visualización rápida y concisa de las palabras más relevantes en cada conjunto de textos.

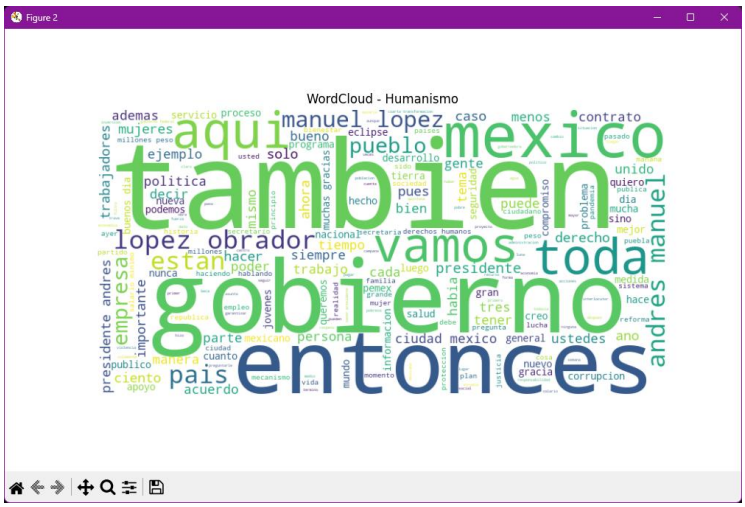


Imagen 12: Salida wordcloud

Al seleccionar el botón de "Frecuencia de Palabras", se desencadena una función similar a la mencionada anteriormente, en la cual el usuario puede ingresar las palabras cuya frecuencia desea conocer para cada corriente. Posteriormente, se genera una ventana emergente que presenta gráficos de barras mostrando la frecuencia de estas palabras, donde en el eje x se representa la frecuencia de aparición y en el eje y se listan las palabras proporcionadas por el usuario.

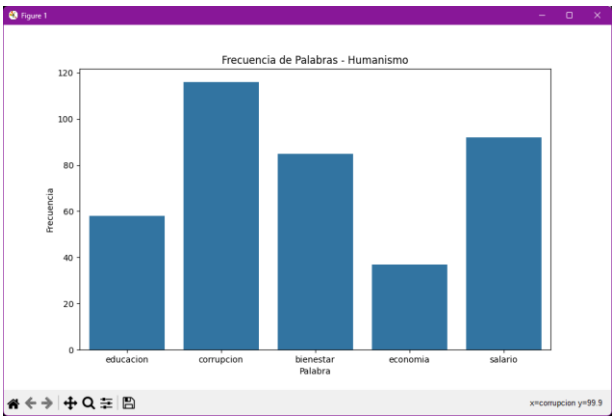


Imagen 13: Salida Frecuencia de palabras

Por último, seleccionar el botón para clasificar un nuevo documento, se carga un archivo de texto (.txt) seleccionado por el usuario. El modelo de inteligencia procesa este texto y, al finalizar, se muestra una ventana emergente con la clasificación sugerida para el documento analizado.

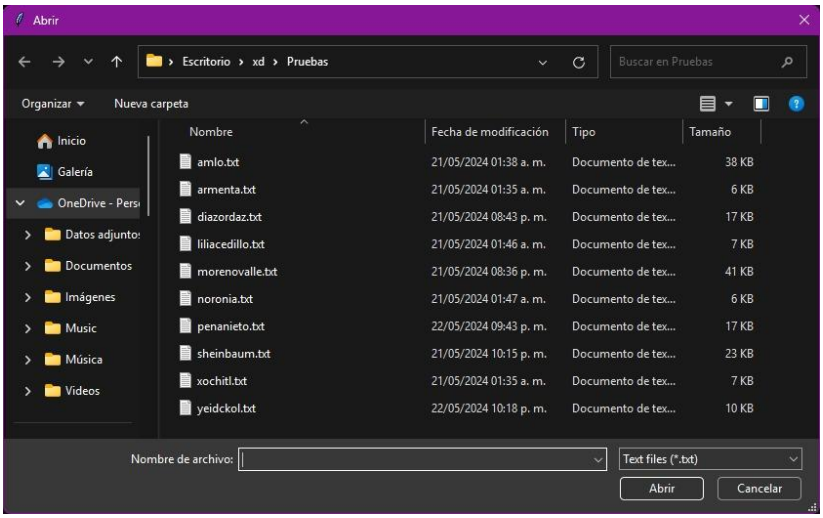


Imagen 14: Documentos nuevos para clasificar.

Elegimos y cargamos un archivo dentro del menú:

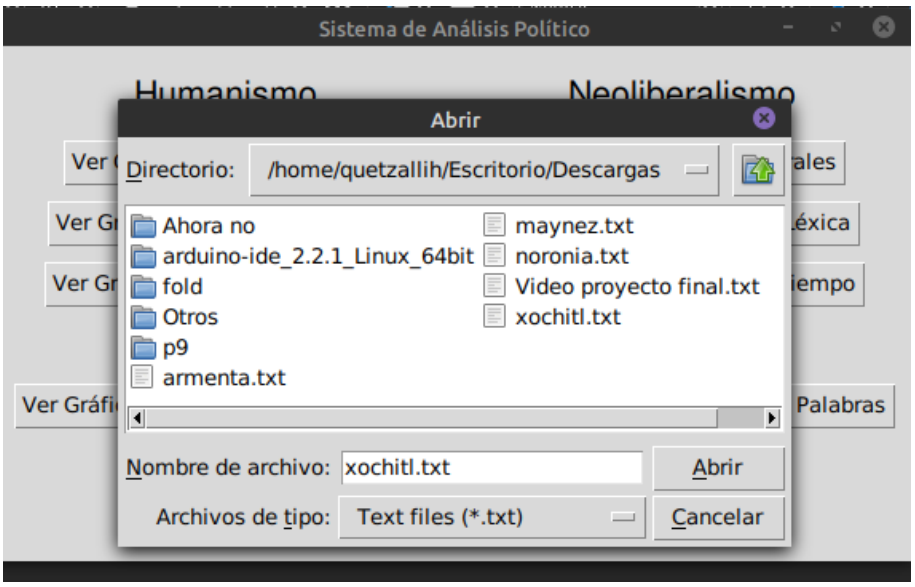


Imagen 15: Cargar archivo .txt.

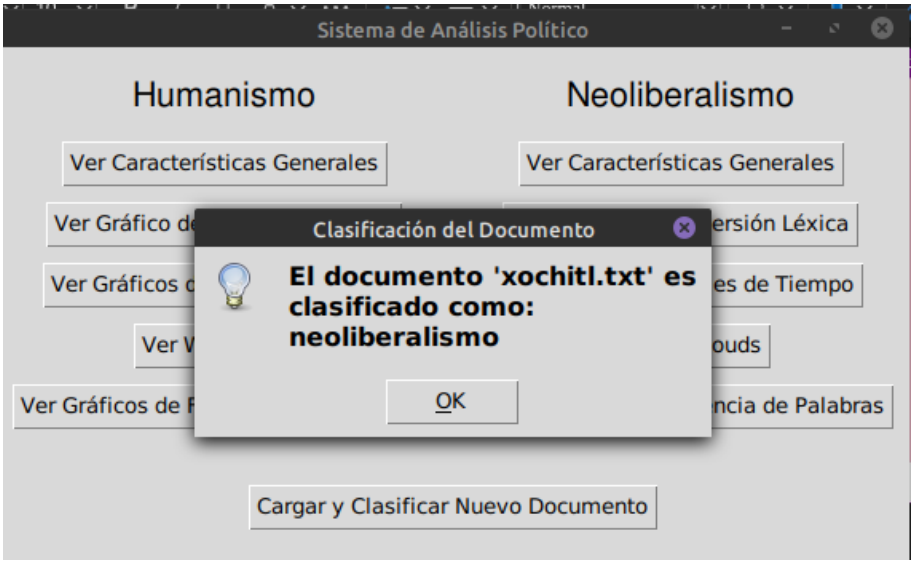


Imagen 16: Salida de la clasificación del documento nuevo.

Para corroborar que las clasificaciones predichas por el modelo son las esperados ingresamos nuevos documentos de diversos personajes políticos tanto del neoliberalismo como humanismo:

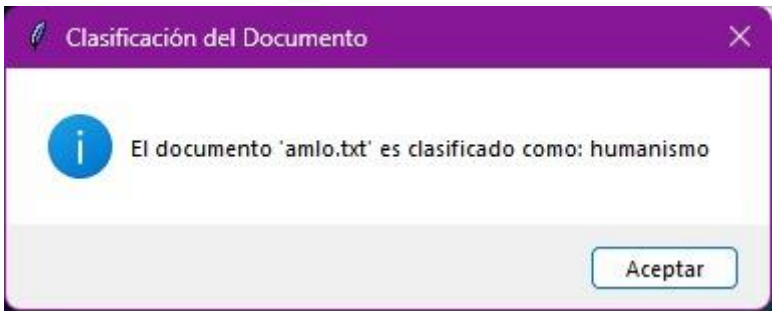


Imagen 17: Salida de la clasificación del documento nuevo, Amlo: humanismo.

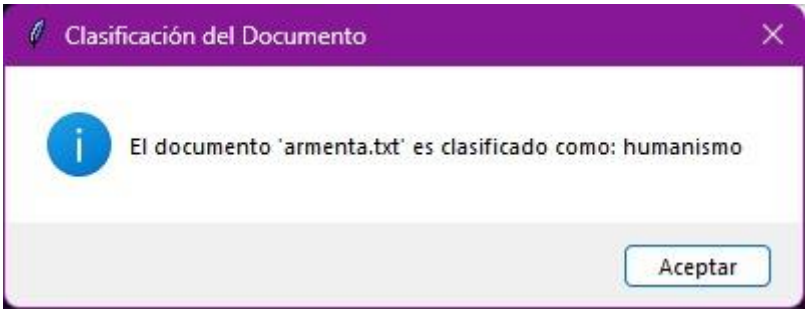


Imagen 18: Salida de la clasificación del documento nuevo, Armenta: humanismo.

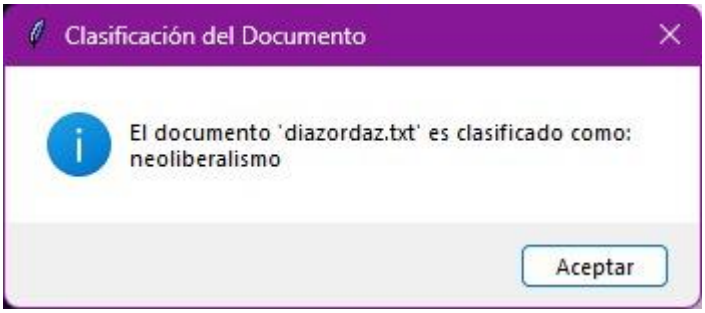


Imagen 19: Salida de la clasificación del documento nuevo, Díaz Ordaz: Neoliberalismo.

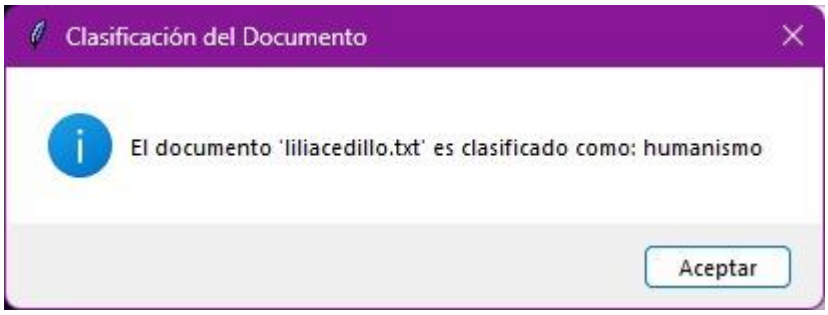


Imagen 20: Salida de la clasificación del documento nuevo, Lilia Cedillo: humanismo.

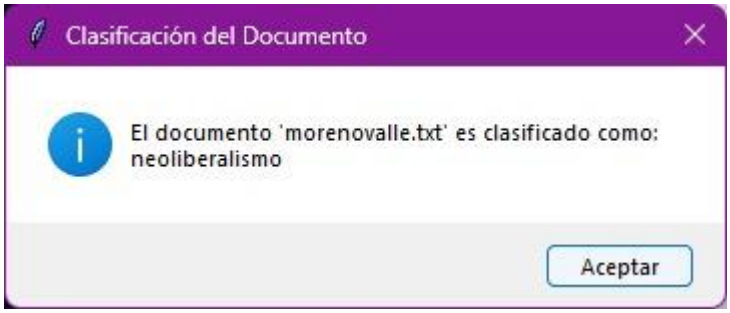


Imagen 21: Salida de la clasificación del documento nuevo, Moreno Valle: Neoliberalismo.

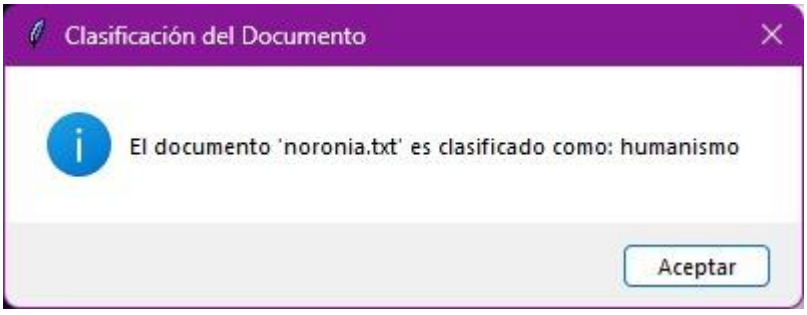


Imagen 22: Salida de la clasificación del documento nuevo, Noroña: humanismo.

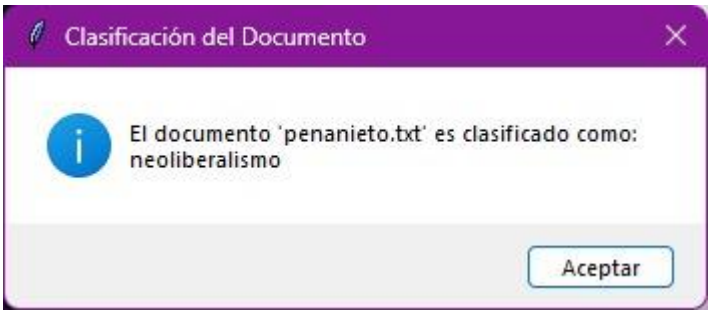


Imagen 23: Salida de la clasificación del documento nuevo, Peñanieto: Neoliberalismo.

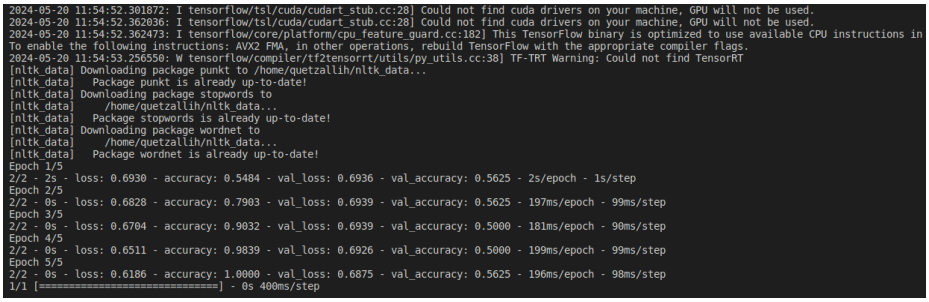


Imagen 24: Entrenamiento de la red.

V. CONCLUSIONES

En conclusión, este proyecto ha sido una experiencia sumamente enriquecedora. A lo largo de su desarrollo, hemos explorado diversas técnicas de recuperación de información, desde la preprocesamiento de textos hasta la implementación de modelos de aprendizaje profundo. Esta práctica nos ha permitido ampliar nuestros conocimientos en el campo de la recuperación de información, comprendiendo en profundidad sus técnicas y aplicaciones.

Hemos implementado una amplia gama de funciones, desde la visualización de características generales hasta la generación de gráficos de dispersión léxica, series de tiempo, nubes de palabras y frecuencia de palabras. Estas herramientas proporcionan una visión profunda y detallada de los discursos políticos, permitiendo identificar tendencias, patrones y temas clave dentro de cada corriente ideológica.

Además, hemos integrado una interfaz gráfica de usuario intuitiva que facilita la interacción con el sistema, permitiendo a los usuarios cargar nuevos documentos y obtener clasificaciones sugeridas de manera rápida y eficiente.

En resumen, nuestro proyecto ha sido un éxito en la creación de una herramienta poderosa para el análisis político, que puede ser utilizada para comprender mejor las tendencias y dinámicas en el discurso político contemporáneo.

VI. REFERENCIAS

- NLTK. (2022). SnowballStemmer. Recuperado de https://www.nltk.org/api/nltk.stem.SnowballStemmer.html?highlight=stopwords
- NLTK. (2022). WordNetLemmatizer. Recuperado de https://www.nltk.org/api/nltk.stem.WordNetLemmatizer.html?highlight=wordnet