
# FACULTAD DE CIENCIAS
# INTELIGENCIA ARTIFICIAL

---

# Proyecto FINAL

---

Equipo: `Skynet Scribes`

Número de poryecto: `03`

**Carlos Daniel Cortés Jiménez**
420004846

**Sarah Sophía Olivares García**
318360638

**Marco Silva Huerta**
316205326

**Laura Itzel Tinoco Miguel**
316020189

**Luis Enrique García Gómez**
315063880

**Fernando Mendoza Eslava**
319097690

Profesora: Cecilia Reyes Peña
Ayudante: Tania Michelle Rubí Rojas

**Semestre 2024-2**

Fecha de entrega:
**24 de Mayo del 2024**

# Índice

# 1. Desarrollo del Proyecto

Nuestra propuesta se centra en el desarrollo de un sistema de inteligencia artificial capaz de generar poesía usando datos entrenados. Este sistema será una herramienta para aquellos que buscan explorar nuevas ideas de presentación y escritura abriendo paso a la creatividad a través de la colaboración con la IA.

- Crear un sistema de generación de poesía automática: Incorporar diferentes estilos literarios en la generación de poesía, asegurando variedad y riqueza en el contenido.
- Aplicar técnicas de procesamiento de lenguaje natural y aprendizaje automático: Modelar patrones poéticos y adaptar la generación de contenido a las preferencias del usuario, permitiendo una experiencia más personalizada.
- Comprender la importancia de un proceso correcto de datos para analizar todo el contenido en un poema y poder usarlo para que nuestro modelo aprenda y pueda generar el contenido nuevo.

## 1.1. Bases Teoricas

**Redes Neuronales**

Una red neuronal es un modelo computacional inspirado en la estructura y funcionamiento del cerebro humano. Está compuesta por nodos interconectados llamados neuronas artificiales, organizados en capas que procesan y transmiten información. Cada neurona recibe entradas, las procesa mediante una función de activación y luego transmite la salida a las neuronas de la capa siguiente. Esto permite que las redes neuronales aprendan patrones complejos y realicen tareas como reconocimiento de patrones, clasificación de datos y predicción.

- Las redes neuronales utilizan una técnica de aprendizaje automático llamada aprendizaje automático supervisado para adaptarse y aprender nuevas tareas.
- Pueden representar información proporcionada durante la etapa de aprendizaje.
- Dado que las redes neuronales pueden calcular datos en paralelo, las máquinas con arquitectura de redes neuronales trabajan más rápido para proporcionar resultados
- Si una red neuronal está parcialmente dañada, puede provocar una caída en el rendimiento; sin embargo, la red neuronal puede conservar algunas de sus propiedades incluso cuando está dañada

**Ajuste de hiperparámetros y selección de modelo**

Imaginemos que vamos a jugar un torneo de cartas, tenemos dos equipos para elegir jugar:

1. Equipo $\alpha$ conformado por jugadores con experiencia en las cartas
2. Equipo $\beta$ conformado por jugadores más novatos pero con habilidades en matemáticas

Con que equipo nos quedamos? Primero, vamos a probar ambos equipos en diferentes partidas y evaluaremos cómo se desempeñan en términos de estrategia y habilidades matemáticas. Tras varias partidas, notas que el equipo de $\alpha$ tiene una ventaja clara en la toma de decisiones durante el juego, pero el equipo $\beta$ tiene una mejor comprensión de las probabilidades y las estadísticas del juego.

Aquí es donde entra el ajuste de hiperparámetros. Vamos a ajustar la estrategia del equipo $\alpha$ para que sea más cauteloso y evite riesgos innecesarios. Pero al equipo $\beta$ le haremos un ajuste para equilibrar su enfoque analítico y hacer decisiones más intuitivas y rápidas.

Después de ajustar ambos equipos, y hacer pruebas, notamos como el equipo $\alpha$ sigue siendo sólido, pero $\beta$ ha mejorado su capacidad para adaptarse a situaciones cambiantes durante el juego.

Aquí es donde entra la validación de exclusión y la validación cruzada. Guardamos algunas partidas para evaluar nuevamente a ambos equipos después de los ajustes. Al hacer esto, podemos asegurar de que los cambios realizados si mejoran el rendimiento de los equipos y no son temporales o un golpe de suerte en una única prueba.

Finalmente la elección del equipo deberá estar basada en quien ha demostrado una mejora constante y sólida en todas las evaluaciones, solo así podremos estar seguros de jugar el torneo.

**Mecanismo de las Redes Neuronales Recurrentes (RNNs)**

Las redes neuronales recurrentes (RNNs) son un tipo especial de red neuronal que se utiliza principalmente para datos secuenciales, como texto o series temporales. Lo que las hace especiales es su capacidad para mantener información a lo largo del tiempo, lo que les permite aprender patrones en secuencias de datos.

Componentes de una RNN

**Memoria y Bucles:** Las RNNs tienen una memoria interna que les permite recordar la información anterior en la secuencia. Esto se logra mediante la conexión en bucle dentro de las neuronas, lo que permite que la información persista.

**Capa Recurrente**: Esta es la capa central de una RNN. A diferencia de las redes neuronales tradicionales, cada neurona en esta capa no solo recibe la entrada actual, sino que también recibe información de la entrada anterior. Esto se logra mediante una conexión recurrente que permite a la red recordar información a lo largo del tiempo.

Como ejemplo imaginemos que vamos a leer una historia. Cada palabra que lees se añade a tu comprensión de la trama. De manera similar, una RNN procesa una palabra a la vez, y cada nueva palabra se añade a su *memoria* de las palabras anteriores.

**Proceso de una RNN**

1. Entrada Secuencial: La RNN toma una secuencia de datos como entrada. Por ejemplo, una oración dividida en palabras.
2. Propagación de la Información: A medida que cada palabra se procesa, la RNN actualiza su estado interno (su ”memoria”) basado en la palabra actual y el estado anterior.
3. Salida Secuencial: La RNN puede generar una salida en cada paso, por ejemplo, predecir la siguiente palabra en una oración.

## 1.2. Estrategia del proyecto

1. Recopilación de Poemas: Juntamos una serie de poemas en dos categorías: amor y tristeza. Son dos porque nos permite enfocarnos en como debe ser el procesamiento y extraer las mejores caracteristicas para entrenar el modelo. Estos datos han sido almacenados en un archivo .json.

2. Pre-procesamiento de Datos: Debemos hacer una limpieza de los datos
3. Desarrollo de Modelos
    - El primer modelo lo queremos entrenar para que pueda aprender y clasificar nuestros dos categorías de poemas utilizando técnicas de procesamiento de lenguaje natural y aprendizaje automático
    - Este segundo modelo será entreando para que pueda procesar los datos ya entrenados y pasandole parte de la estructura del poema pueda construir una nueva versión. Hasta el momento no hemos considerado más variables que pueda influir en la creación de modelo pero es muy posible que se integren.
4. Evaluación y Mejora: Desarrollo de métricas para evaluar la calidad poética y realizar ajustes basados en la retroalimentación de las pruebas realizadas.

# 2. Limpieza de datos

Proceso de limpieza de los poemas, el código fuente se encuentra en el archivo `src/limpiaPoemas.py`.

- **Cargar los datos:** Se utiliza la función `cargar_poemas` para cargar los datos desde un archivo JSON. Esta función abre el archivo en modo lectura y utiliza `json.load` para leer y convertir el contenido del archivo JSON en un diccionario de Python.

```
def cargar_poemas(poemas_json):
    with open(poemas_json, 'r', encoding='utf-8') as archivo:
        poemas = json.load(archivo)
    return poemas
```

- **Convertir a minúsculas:** La función `convertir_minusculas` convierte todo el texto a minúsculas. Esto es esencial para normalizar el texto y facilitar la comparación de palabras.

```
def convertir_minusculas(texto):
    return texto.lower()
```

- **Eliminar caracteres especiales:** La función `eliminar_caracteres_especiales` elimina los caracteres especiales del texto utilizando una expresión regular. Esto ayuda a limpiar el texto de símbolos no deseados que pueden interferir con el procesamiento.

```
def eliminar_caracteres_especiales(texto):
    return re.sub(r'[^\w\s]', '', texto)
```

- **Eliminar acentos:** La función `eliminar_acentos` normaliza el texto eliminando acentos para que las palabras con y sin acentos se traten por igual.

```
def eliminar_acentos(texto):
    forma_nfkd = unicodedata.normalize('NFKD', texto)
    return "".join([c for c in forma_nfkd if not unicodedata.combining(
```

- **Eliminar espacios extra:** La función `eliminar_espacios_extra` reduce múltiples espacios a un solo espacio, mejorando la consistencia del texto.

```
def eliminar_espacios_extra(texto):
    return " ".join(texto.split())
```

- **Tokenizar:** La función `tokenizar` divide el texto en una lista de palabras (tokens). Esto es útil para el análisis y procesamiento del texto.

```
def tokenizar(texto):
    return texto.split()
```

- **Eliminar palabras vacías:** Las palabras vacías (stopwords) son palabras comunes que no aportan mucho significado. La función `eliminar_palabras_vacias` elimina estas palabras de la lista de tokens para centrarse en las palabras significativas.

```
palabras_vacias = set(stopwords.words('spanish'))

def eliminar_palabras_vacias(tokens):
    return [palabra for palabra in tokens if palabra not in palabras_va
```

- **Limpiar contenido:** La función `limpiar_contenido` aplica todas las funciones de limpieza en orden, transformando el texto en un formato limpio y normalizado.

```
def limpiar_contenido(contenido):
    contenido = convertir_minusculas(contenido)
    contenido = eliminar_caracteres_especiales(contenido)
    contenido = eliminar_acentos(contenido)
    contenido = eliminar_espacios_extra(contenido)
    tokens = tokenizar(contenido)
    tokens = eliminar_palabras_vacias(tokens)
    return " ".join(tokens)
```

- **Guardar los datos limpios:** La función `guardar_poemas` guarda los poemas limpios en un nuevo archivo JSON.

```
def guardar_poemas(poemas, poemas_json):
    with open(poemas_json, 'w', encoding='utf-8') as archivo:
        json.dump(poemas, archivo, ensure_ascii=False, indent=4)
```

- **Función principal:** La función `limpiar_poemas` carga, limpia y guarda los poemas. Primero, carga los poemas del archivo de entrada. Luego, limpia cada poema utilizando `limpiar_contenido`. Finalmente, guarda los poemas limpios en el archivo de salida.

```
def limpiar_poemas(archivo_entrada, archivo_salida):
    poemas = cargar_poemas(archivo_entrada)
    poemas_limpios = []

    for item in poemas:
        contenido_limpio = limpiar_contenido(item["contenido"])
        poemas_limpios.append({
            "categoria": item["categoria"],
            "autor": item["autor"],
            "titulo": item["titulo"],
            "contenido": contenido_limpio
        })

    guardar_poemas(poemas_limpios, archivo_salida)
```

# 3. Modelo de entrenamiento

## 3.1. Modelo capas LSTM

Las Redes Neuronales Recurrentes (RNN) son adecuadas para manejar datos secuenciales, como texto. Las capas LSTM (Long Short-Term Memory) son una variante de las RNN que permiten aprender dependencias a largo plazo, superando problemas como el desvanecimiento del gradiente.

Nuestro modelo utiliza capas LSTM bidireccionales, que procesan la secuencia en ambas direcciones, y capas de Dropout, que reducen el sobreajuste. Una capa de embedding convierte los enteros que representan palabras en vectores densos de tamaño fijo, y una capa densa final con activación softmax clasifica las categorías de poemas.

## 3.2. Función: Entrenar RNN

La función `entrenar_rnn` carga datos de poemas limpios, los preprocesa y entrena un modelo RNN con capas LSTM.

Primero, se cargan los datos de poemas limpios desde un archivo JSON. Cada poema se tokeniza, convirtiendo las palabras en números enteros únicos. Luego, se codifican las categorías de los poemas en formato one-hot.

Las secuencias de texto se ajustan para tener la misma longitud mediante padding. El modelo RNN se define con capas de embedding, LSTM bidireccionales y Dropout. Se compila utilizando el optimizador Adam y la función de pérdida `categorical_crossentropy`.

El modelo se entrena con un 20 % de los datos reservados para la validación y se guarda en el disco. Finalmente, se evalúa el modelo utilizando la función `evaluar_modelo`.

## 3.3. Función: Evaluar Modelo

La función `evaluar_modelo` evalúa el rendimiento del modelo RNN entrenado, calculando métricas como pérdida, precisión, precisión por clase y exhaustividad.

Las predicciones del modelo se comparan con las etiquetas verdaderas para generar una matriz de confusión y un reporte de clasificación. La matriz de confusión visualiza el desempeño del modelo mostrando el número de predicciones correctas e incorrectas para cada clase. Esto ayuda a identificar problemas específicos y evaluar la precisión y exhaustividad del modelo en diferentes categorías.

## 3.4. Matriz de Confusión

La matriz de confusión es una herramienta fundamental en la evaluación de modelos de clasificación. Cada fila de la matriz representa las instancias de una clase real, mientras que cada columna representa las instancias de una clase predicha.

Este análisis ayuda a identificar problemas de confusión entre clases específicas y a evaluar la precisión y la exhaustividad del modelo en diferentes categorías. El análisis se complementa con un reporte de clasificación, que proporciona métricas detalladas como precisión, exhaustividad y F1-score para cada clase.

**Resultados**

- **Pérdida (Loss):** El valor de pérdida obtenido después del entrenamiento del modelo es de X, indicando la cantidad de error en las predicciones.

- **Precisión (Accuracy):** La precisión del modelo es del Y %, lo que significa que el modelo clasifica correctamente el Z % de los poemas en el conjunto de prueba.
- **Precisión por clase (Precision):** La precisión por clase varía según la categoría, indicando la proporción de verdaderos positivos entre el total de predicciones positivas realizadas por el modelo para cada clase.
- **Exhaustividad (Recall):** La exhaustividad varía según la categoría, indicando la proporción de verdaderos positivos entre el total de instancias reales de cada clase.
- **Matriz de Confusión:** La matriz de confusión muestra el desempeño del modelo para cada categoría, visualizando las predicciones correctas e incorrectas.
- **Reporte de Clasificación:** El reporte de clasificación detalla las métricas de precisión, recall y F1-score para cada categoría, proporcionando una evaluación exhaustiva del rendimiento del modelo.

# 4. Red RNN

- Carga de Datos: La función `cargar poemas limpios` lee el archivo JSON generado en el paso anterior con todos los poemas ya limpios.

Preprocesamiento de Texto

- Tokenización: El Tokenizer convierte el texto en secuencias de enteros, donde cada entero representa una palabra única en el corpus.
- Padding: `pad sequences` se utiliza para asegurar que todas las secuencias tengan la misma longitud rellenando con ceros las secuencias más cortas.
- Codificación de Categorías: Las categorías de los poemas se codifican en enteros y luego se convierten en vectores binarios (one-hot encoding) para que puedan ser utilizadas en la clasificación.

Construcción del Modelo RNN

- Embedding: La capa de Embedding transforma los enteros de las palabras en vectores densos de un tamaño especificado (100 en este caso).
- LSTM Bidireccional: Las capas Bidirectional LSTM permiten que la red aprenda dependencias en ambas direcciones (pasado y futuro) dentro de la secuencia.
- Dropout: Se utiliza para reducir el sobreajuste al *apagar* aleatoriamente algunas neuronas durante el entrenamiento.
- Dense: La capa Dense al final se utiliza para la clasificación, con una activación softmax para obtener probabilidades de las categorías.
- Compilación y Entrenamiento: El modelo se compila con un optimizador adam y una función de pérdida `categorical crossentropy`, adecuada para clasificación multiclase. Luego, se entrena con los datos de entrada y las categorías codificadas.
- Evaluación del Modelo: Después del entrenamiento, el modelo se evalúa con los mismos datos para obtener métricas como precisión y recall.

Guardado del Modelo

La línea

```
modelo_rnn.save('modelo_poemas_rnn.keras')
```

es responsable de guardar el modelo entrenado

1. Guardar la Arquitectura del Modelo: La estructura completa del modelo, incluyendo la configuración de cada capa (como las capas LSTM bidireccionales y las capas de Dropout), se guarda en el archivo.
2. Guardar los Pesos del Modelo: Los pesos son los valores aprendidos durante el entrenamiento del modelo. Estos son cruciales ya que contienen la información que el modelo ha aprendido sobre los patrones de texto en los poemas.
3. Guardar la Configuración de Entrenamiento: Esto incluye la función de pérdida, el optimizador y cualquier otra métrica que se haya definido durante la compilación del modelo.

4. Guardar el Estado del Optimizador: Esto permite reanudar el entrenamiento desde donde se dejó sin perder el *momentum* que el optimizador ha acumulado.

El archivo resultante, es un archivo binario que contiene toda la información necesaria para recrear el modelo exactamente como estaba en el momento de guardar. Esto significa que puedes cargar el modelo más tarde (incluso en una máquina diferente) y usarlo para hacer predicciones o continuar el entrenamiento. El formato *.keras* es específico de la biblioteca Keras y es una forma conveniente de guardar modelos de TensorFlow, ya que Keras está integrado en TensorFlow.

Este proceso es típico en el aprendizaje automático y el procesamiento del lenguaje natural (NLP) para tareas de generación de texto. El modelo aprende a predecir la siguiente palabra en una secuencia, lo que permite generar texto palabra por palabra.

# 5. Generación de Poemas

## 5.1. BERT (Bidirectional Encoder Representations from Transformers)

BERT, desarrollado por Google, es un modelo de lenguaje basado en la arquitectura Transformer, diseñado para entender el contexto bidireccional en el procesamiento del lenguaje natural. A diferencia de los modelos unidireccionales, que leen el texto de izquierda a derecha o de derecha a izquierda, BERT lee el texto en ambas direcciones simultáneamente. Esto permite una comprensión más profunda del contexto de cada palabra en una oración.

### Tokenización y Preprocesamiento

Para utilizar BERT, primero es necesario tokenizar el texto de entrada. La tokenización divide el texto en tokens, que son las unidades básicas que el modelo procesará. BERT utiliza una técnica de tokenización llamada WordPiece, que segmenta palabras desconocidas en subpalabras conocidas, permitiendo que el modelo maneje un vocabulario más manejable y eficiente.

### Generación de Texto

En la generación de texto con BERT, el modelo se usa para predecir palabras enmascaradas en una secuencia de entrada, generando texto coherente basado en el contexto proporcionado. Este proceso implica codificar el texto de entrada en tokens, procesarlos a través del modelo BERT y decodificar los tokens generados para obtener el texto final.

## 5.2. Redes Neuronales Recurrentes (RNN)

Las RNN son una clase de redes neuronales diseñadas para manejar datos secuenciales, como texto o series temporales. En particular, las LSTM (Long Short-Term Memory) y las GRU (Gated Recurrent Unit) son variantes de RNN que abordan el problema de los gradientes desvanecidos, permitiendo que el modelo recuerde información a largo plazo.

### Tokenización y Secuencias

Para entrenar una RNN en la generación de texto, primero se convierte el texto en secuencias de números, donde cada número representa una palabra o token. Estas secuencias se utilizan para entrenar el modelo a predecir la siguiente palabra en una secuencia dada, basándose en las palabras anteriores.

### Entrenamiento y Generación de Texto

El entrenamiento de una RNN implica alimentar secuencias de texto al modelo y ajustar sus pesos para minimizar el error en la predicción de la siguiente palabra. Una vez entrenado, el modelo puede generar texto comenzando con una secuencia de entrada y prediciendo iterativamente las palabras siguientes.

## 5.3. Fusión de BERT y RNN para la Generación de Texto

### Inicialización del Texto con BERT

El proceso de generación de texto comienza utilizando BERT para generar un texto inicial basado en un verso dado. BERT se encarga de proporcionar un contexto coherente y rico, generando una secuencia inicial que sirve como entrada para la RNN.

#### Extensión del Texto con RNN

Una vez generado el texto inicial con BERT, se utiliza una RNN para extender el texto. La RNN toma la secuencia generada por BERT y predice iterativamente las palabras siguientes, basándose en el contexto proporcionado por las palabras anteriores.

#### Ventajas de la Fusión

La combinación de BERT y RNN aprovecha las fortalezas de ambos modelos. BERT proporciona una comprensión profunda del contexto bidireccional, mientras que la RNN maneja la generación secuencial de texto, permitiendo una extensión coherente y fluida del verso inicial. Esta fusión resulta en un sistema de generación de texto más potente y preciso, capaz de producir poemas y otros tipos de textos creativos de alta calidad.

## 5.4. Aplicaciones y Resultados

Esta metodología se aplica en el entrenamiento y generación de poemas. El proceso comienza con la limpieza y preprocesamiento de los poemas, seguido del entrenamiento de la RNN generativa. Finalmente, se utiliza el modelo entrenado junto con BERT para generar nuevos poemas, demostrando la eficacia y creatividad del enfoque.

- **Inicialización:** Se utiliza BERT para generar un verso inicial basado en un input dado.
- **Generación:** La RNN toma el verso inicial y lo extiende, generando nuevas palabras secuencialmente.
- **Resultados:** El poema generado combina la coherencia contextual de BERT con la fluidez secuencial de la RNN, resultando en un texto creativo y cohesivo.

Este enfoque híbrido representa un avance significativo en la generación automática de texto, combinando lo mejor de ambos mundos: la comprensión contextual profunda de BERT y la capacidad de generación secuencial de las RNN.

# Referencias


[1] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 2022, cap. 16, ISBN: 9781098122461. dirección: https://books.google.com.mx/books?id=X5ySEAAAQBAJ.

[2] L. Tunstall, L. von Werra y T. Wolf, *Natural Language Processing with Transformers*. O'Reilly Media, 2022, ISBN: 9781098103194. dirección: https://books.google.com.mx/books?id=nTxbEAAAQBAJ.

[3] A. Müller y S. Guido, *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, 2016, cap. 7, ISBN: 9781449369903. dirección: https://books.google.com.mx/books?id=1-4lDQAAQBAJ.

[4] S. Bird, E. Klein y E. Loper, *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, 2009, ISBN: 9780596555719. dirección: https://books.google.com.mx/books?id=KGIbfiiP1i4C.