Procesamiento de lenguaje natural para el análisis y clasificación de sentimiento usando una Red Neuronal Recurrente

Alexis Horacio López Fragoso

Universidad Autónoma de México,
Facultad de Estudios Superiores Cuautitlán,
Cuautitlán, Estado de México, México
guados123@gmail.com

Abstract. Este proyecto presenta el desarrollo de un sistema para el análisis y clasificación de sentimientos utilizando una Red Neuronal Recurrente (RNN). Se emplea PyTorch para el entrenamiento del modelo y torchtext para el procesamiento eficiente de textos, utilizando el *Twitter and Reddit Sentimental Analysis Dataset* como fuente de datos. Este conjunto de datos proporciona una amplia variedad de textos etiquetados en tres categorías: positivo, negativo y neutro. Para el preprocesamiento de los textos, se aplican técnicas de scikit-learn que permiten limpiar y transformar los datos, mejorando así la calidad y el rendimiento del modelo. La arquitectura de la RNN está diseñada para capturar las dependencias secuenciales en los datos textuales, lo que mejora significativamente la precisión en la clasificación de sentimientos. Además, se implementa una interfaz física para la visualización de los resultados mediante una pantalla OLED de 128x64 píxeles conectada a un Arduino Uno. Esta integración permite mostrar las clasificaciones obtenidas.

Keywords: Procesamiento de lenguaje natural, Red Neuronal Recurrente, Análisis de sentimientos, PyTorch, Arduino Uno.

1 Introduccin

El análisis de sentimientos es una disciplina clave dentro del procesamiento de lenguaje natural (PLN) que se enfoca en extraer y analizar las opiniones, emociones y actitudes expresadas en textos escritos [1]. Con el crecimiento exponencial de datos generados en plataformas de redes sociales como Twitter y Reddit, existe una necesidad creciente de herramientas que puedan procesar y entender esta información de manera eficiente y automática [2].

Las Redes Neuronales Recurrentes (RNN) han demostrado ser altamente efectivas en el tratamiento de datos secuenciales, como el lenguaje natural, debido a su capacidad para mantener información contextual a lo largo de secuencias de datos [3]. En este proyecto, se propone el diseño e implementación de una RNN para la clasificación de sentimientos en textos, categorizándolos en positivo, negativo y neutro.

Para la construcción y entrenamiento del modelo, se utiliza PyTorch [4], una biblioteca de código abierto que ofrece gran flexibilidad y eficiencia en el desarrollo de redes neuronales. El procesamiento de textos se gestiona mediante `torchtext` [5], que facilita la manipulación y transformación de grandes conjuntos de datos textuales. El conjunto de datos seleccionado es el *Twitter and Reddit Sentimental Analysis Dataset* [6], que proporciona una amplia gama de ejemplos reales y etiquetados, fundamentales para el aprendizaje efectivo del modelo.

El preprocesamiento de los textos es una etapa crucial para garantizar la calidad y relevancia de los datos de entrada al modelo. Se emplean herramientas de `scikit-learn` [7] para llevar a cabo tareas como la tokenización, eliminación de *stop words* y lematización, optimizando así el rendimiento y la precisión del sistema.

Como parte innovadora del proyecto, se desarrolla una interfaz física utilizando una pantalla OLED de 128x64 píxeles conectada a un Arduino Uno [8]. Esta implementación permite visualizar en tiempo real los resultados de la clasificación de sentimientos, ofreciendo una experiencia interactiva y tangible al usuario, y demostrando cómo las soluciones de aprendizaje automático pueden integrarse con hardware accesible para aplicaciones prácticas.

2 Estado del Arte

El análisis de sentimientos es un área activa de investigación dentro del procesamiento de lenguaje natural (PLN), enfocada en extraer información subjetiva de textos [1]. Con el auge de las redes sociales, se ha generado un gran volumen de datos textuales que requieren técnicas avanzadas para su análisis.

2.1 Análisis de Sentimientos en PLN

Las técnicas tradicionales de análisis de sentimientos se basaban en métodos de aprendizaje automático clásicos y enfoques basados en léxicos [9]. Sin embargo, estos métodos presentan limitaciones al manejar ambigüedades y contextos complejos en el lenguaje natural.

2.2 Redes Neuronales Recurrentes en Análisis de Sentimientos

Las Redes Neuronales Recurrentes (RNN) han demostrado ser eficaces en tareas de PLN debido a su capacidad para procesar secuencias de datos y mantener información contextual [3]. Variantes como las LSTM (Long Short-Term Memory) y GRU (Gated Recurrent Unit) han mejorado aún más el rendimiento en análisis de sentimientos [10].

Estudios recientes han utilizado RNN para clasificar sentimientos en textos de redes sociales, obteniendo resultados superiores a los métodos tradicionales [11].

2.3 Uso de PyTorch y `torchtext` en PLN

PyTorch es una biblioteca de aprendizaje profundo de código abierto que ha ganado popularidad en la comunidad de investigación y desarrollo debido a su enfoque dinámico y su facilidad para construir y entrenar modelos neuronales [4]. A diferencia de otras bibliotecas que utilizan grafos computacionales estáticos, PyTorch permite la construcción dinámica de grafos, lo que facilita el depurado y la modificación de modelos complejos, características especialmente útiles en tareas de procesamiento de lenguaje natural (PLN).

La librería `torchtext` es un paquete complementario de PyTorch diseñado específicamente para el procesamiento de datos textuales [5]. Proporciona conjuntos de datos preprocesados, funciones para cargar y procesar textos, y herramientas para construir vocabularios y representar palabras mediante vectores (*embeddings*). Estas funcionalidades simplifican considerablemente el flujo de trabajo en proyectos de PLN, permitiendo a los investigadores y desarrolladores centrarse en la arquitectura del modelo y en la experimentación.

En el contexto del análisis de sentimientos, `torchtext` facilita la tokenización de textos, la conversión de palabras a índices numéricos y la creación de lotes (*batches*) para el entrenamiento eficiente del modelo [12]. Además, soporta técnicas avanzadas como el uso de *embeddings* preentrenados (por ejemplo, GloVe o word2vec), que enriquecen el modelo con información semántica [13].

2.4 Preprocesamiento de Texto con `scikit-learn`

El preprocesamiento es una etapa crucial en el análisis de sentimientos. Herramientas como `scikit-learn` ofrecen funciones para limpiar y transformar textos, incluyendo tokenización, eliminación de *stop words* y lematización [7].

El uso de `scikit-learn` en combinación con bibliotecas de aprendizaje profundo permite mejorar la calidad de los datos de entrada y, por ende, el rendimiento del modelo.

2.5 Conjuntos de Datos para el Anlisis de Sentimientos

Los conjuntos de datos son fundamentales para entrenar y evaluar modelos de aprendizaje automtico en el anlisis de sentimientos. Adems del conjunto de datos utilizado en este proyecto, existen otros ampliamente reconocidos como el *Stanford Sentiment Treebank* [14], que proporciona anotaciones de sentimiento a nivel de frase y sintaxis, y el *IMDB Movie Reviews Dataset* [15], utilizado comnmente para la clasificacin binaria de sentimientos.

Estos conjuntos de datos permiten a los investigadores comparar el rendimiento de diferentes modelos y tcnicas bajo condiciones similares, contribuyendo al avance del campo.

2.6 Mtricas de Evaluacin en Anlisis de Sentimientos

La evaluacin de los modelos de anlisis de sentimientos se basa en mtricas como la precisin, la exhaustividad (*recall*), la puntuacin F1 y la matriz de confusin [16]. Estas mtricas proporcionan informacin sobre la capacidad del modelo para clasificar correctamente los datos y detectar tanto verdaderos positivos como verdaderos negativos.

Es esencial utilizar mtricas adecuadas para interpretar correctamente el rendimiento del modelo y realizar mejoras continuas.

2.7 Integración de Hardware para Visualización de Datos

La visualización de resultados en dispositivos físicos como pantallas OLED conectadas a microcontroladores como Arduino Uno es menos común en el contexto de PLN. Sin embargo, existen proyectos que integran hardware y aprendizaje automático para crear interfaces interactivas [17].

Esta integración abre posibilidades para aplicaciones prácticas y educativas, permitiendo una interacción más tangible con los resultados de modelos de PLN.

3 Conocimientos Previos

Para comprender plenamente el desarrollo de este proyecto, es esencial tener un conocimiento sólido en varias áreas clave del aprendizaje automático y las matemáticas aplicadas. A continuación, se detallan los conceptos fundamentales que sirven como base para este trabajo.

3.1 Procesamiento de Lenguaje Natural (PLN)

El Procesamiento de Lenguaje Natural (PLN) es una rama de la inteligencia artificial que se ocupa de la interacción entre las computadoras y el lenguaje humano [18]. El PLN permite a las máquinas comprender, interpretar y generar lenguaje humano de manera significativa, lo cual es fundamental para aplicaciones como el análisis de sentimientos, la traducción automática y los sistemas de diálogo.

3.2 Análisis de Sentimientos

El análisis de sentimientos es una aplicación del PLN que busca identificar y extraer opiniones subjetivas de textos, determinando la actitud emocional del autor hacia un tema específico [1]. Formalmente, dado un texto T, el objetivo es asignar una etiqueta de sentimiento $s \in \{\text{positivo}, \text{negativo}, \text{neutro}\}$.

3.3 Fundamentos de Aprendizaje Automático

El aprendizaje automático es un campo que permite a las computadoras aprender patrones a partir de datos y tomar decisiones basadas en ellos [19]. En problemas de clasificación, el objetivo es aprender una función $f : X \to Y$, donde X es el espacio de las características de entrada y Y es el conjunto de etiquetas de clase.

El aprendizaje se realiza minimizando una función de pérdida $L(\theta)$ respecto a los parámetros del modelo θ, generalmente utilizando métodos de optimización como el descenso por gradiente [20]:

$$\theta^* = \arg\min_{\theta} L(\theta). \tag{1}$$

3.4 Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes (RNN) son arquitecturas diseñadas para procesar datos secuenciales [3]. En una RNN, el estado oculto h_t en el tiempo t depende del estado oculto anterior h_{t-1} y de la entrada actual x_t:

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \tag{2}$$

donde W_{xh} y W_{hh} son matrices de pesos, b_h es el sesgo y ϕ es una función de activación no lineal, como la tangente hiperbólica tanh.

La salida y_t se calcula como:

$$y_t = \sigma(W_{hy}h_t + b_y), \tag{3}$$

donde W_{hy} es la matriz de pesos de salida, b_y es el sesgo de salida y σ es la función softmax para obtener probabilidades de clase.

3.5 Retropropagación a Través del Tiempo (BPTT)

La Retropropagación a Través del Tiempo (Backpropagation Through Time, BPTT) es un algoritmo utilizado para entrenar RNN [21]. Consiste en desplegar la red en el tiempo y aplicar el algoritmo de retropropagación estándar para calcular los gradientes de los errores respecto a los pesos.

El gradiente de la función de pérdida L respecto a los pesos se obtiene sumando los gradientes en cada paso temporal:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W}. \tag{4}$$

3.6 Problema de Desvanecimiento y Explosión del Gradiente

Durante el entrenamiento de RNN, es común enfrentar el problema de desvanecimiento o explosión del gradiente [22]. Esto ocurre cuando los gradientes se

vuelven extremadamente pequeños o grandes al propagarse a través de muchas capas, dificultando el aprendizaje efectivo de dependencias a largo plazo.

El desvanecimiento del gradiente se debe a que las derivadas sucesivas de la función de activación ϕ pueden reducir el valor del gradiente exponencialmente:

$$\left\| \frac{\partial h_t}{\partial h_{t-1}} \right\| = \left\| W_{hh} \phi'(a_{t-1}) \right\|, \tag{5}$$

donde $\phi'(a_{t-1})$ es la derivada de la función de activación.

3.7 Long Short-Term Memory (LSTM)

Para resolver el problema de desvanecimiento del gradiente, se introdujeron las unidades de memoria a largo corto plazo (LSTM) [10]. Las LSTM utilizan puertas para controlar el flujo de información y mantener las dependencias a largo plazo.

Las ecuaciones de una LSTM son:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i), \tag{6}$$
$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f), \tag{7}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o), \tag{8}$$
$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c), \tag{9}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \tag{10}$$
$$h_t = o_t \odot \tanh(c_t), \tag{11}$$

donde i_t, f_t y o_t son las puertas de entrada, olvido y salida respectivamente, $\tilde{c}_t$ es el estado de la célula candidata, c_t es el estado de la célula y $\odot$ denota la multiplicación elemento a elemento.

3.8 Funciones de Activación y Función Softmax

Las funciones de activación introducen no linealidad en las redes neuronales. Algunas de las más utilizadas son:

- Tangente hiperbólica (tanh):

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{12}$$

- Sigmoide (σ):

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{13}$$

La función softmax se utiliza en la capa de salida para convertir los valores en probabilidades:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}, \tag{14}$$

donde z_i es la entrada a la neurona i en la capa de salida.

3.9 Función de Pérdida y Optimización

Para entrenar el modelo, se utiliza una función de pérdida que cuantifica la discrepancia entre las predicciones y las etiquetas verdaderas. En problemas de clasificación multiclase, se utiliza comúnmente la entropía cruzada categórica [23]:

$$L = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik}), \tag{15}$$

donde N es el número de muestras, K es el número de clases, y_{ik} es 1 si la muestra i pertenece a la clase k y $\hat{y}_{ik}$ es la probabilidad predicha.

Los parámetros del modelo se actualizan utilizando algoritmos de optimización como el descenso por gradiente estocástico (SGD) o Adam [24].

$$\theta_{t+1} = \theta_t - \eta\nabla_\theta L(\theta_t), \tag{16}$$

donde η es la tasa de aprendizaje y $\nabla_\theta L$ es el gradiente de la pérdida respecto a los parámetros.

4 Metodología

En esta sección se detalla el proceso seguido para desarrollar el sistema de análisis y clasificación de sentimientos utilizando una Red Neuronal Recurrente (RNN). El enfoque se divide en varias etapas clave: preparación del conjunto de datos, preprocesamiento de los textos, construcción y entrenamiento del modelo, evaluación del rendimiento y la implementación de una interfaz física utilizando Arduino y una pantalla OLED.

4.1 Descripción del Conjunto de Datos

Se utilizaron dos conjuntos de datos principales: *Twitter Data* y *Reddit Data*, obtenidos de la fuente proporcionada en [6]. Ambos conjuntos contienen publicaciones etiquetadas como *positivo*, *negativo* o *neutro*, lo que facilita la tarea de clasificación multiclase.

Los datos se cargaron utilizando la biblioteca `pandas` y se combinaron para formar un único conjunto de datos unificado. Se eliminaron las filas con valores nulos en las columnas relevantes (`clean_text` y `category`) para asegurar la calidad de los datos.

4.2 Análisis Exploratorio de Datos

Antes del preprocesamiento, se realizó un análisis exploratorio para entender la distribución de las clases y las características de los textos. Se utilizaron herramientas de visualización como `matplotlib` y `seaborn` para:

– Graficar la distribución de muestras por categoría.
– Analizar la longitud de los textos y su frecuencia.
– Generar una nube de palabras con `wordcloud` para visualizar las palabras más frecuentes en el corpus.

4.3 Preprocesamiento de Datos

El preprocesamiento es una etapa crucial para mejorar la calidad de los datos y el rendimiento del modelo. Se realizaron las siguientes operaciones:

– **Limpieza de texto:** Se implementó una función de preprocesamiento que convierte el texto a minúsculas, elimina URLs, menciones, caracteres especiales y números utilizando expresiones regulares. Esto está en línea con las mejores prácticas en procesamiento de lenguaje natural [25].
– **Tokenización:** Los textos limpios se tokenizaron dividiendo cada oración en palabras individuales. Esto facilita la construcción del vocabulario y la conversión de palabras a índices numéricos.
– **Codificación de etiquetas:** Se utilizó `LabelEncoder` de `scikit-learn` para convertir las etiquetas categóricas en valores numéricos, permitiendo al modelo procesar las clases como salidas discretas.
– **Construcción del vocabulario:** Se creó un vocabulario utilizando `torchtext.vocab` para mapear cada palabra a un índice único. Se consideró una frecuencia mínima para incluir palabras y se manejaron tokens especiales como `<unk>` (desconocido) y `<pad>` (relleno) [5].

4.4 División del Conjunto de Datos

El conjunto de datos procesado se dividió en conjuntos de entrenamiento y validación en una proporción de 80% y 20% respectivamente, utilizando la función `train_test_split` de `scikit-learn` [7]. Esta división permite evaluar el rendimiento del modelo en datos no vistos durante el entrenamiento.

4.5 Construcción del Modelo

Se diseñó una RNN utilizando la biblioteca `PyTorch` [4], con la siguiente arquitectura:

– **Capa de Embedding:** Transforma los índices de palabras en vectores densos de tamaño fijo (*embeddings*) para capturar relaciones semánticas entre palabras.
– **Capa LSTM:** Una capa de *Long Short-Term Memory* con dos capas ocultas y 256 unidades en cada una, capaz de capturar dependencias a largo plazo en las secuencias de texto [10].
– **Capa Fully Connected:** Una capa lineal que proyecta la salida de la LSTM al espacio de las clases.
– **Función Softmax:** Se aplica una función `LogSoftmax` para obtener probabilidades logarítmicas de cada clase.

4.6 Entrenamiento del Modelo

El modelo se entrenó utilizando el conjunto de entrenamiento, con las siguientes configuraciones:

- **Función de pérdida:** `Negative Log Likelihood Loss (NLLLoss)` adecuada para problemas de clasificación multiclase [20].
- **Optimizador:** `Adam` con una tasa de aprendizaje de 0.001, conocido por su eficiencia en problemas de aprendizaje profundo [24].
- **Tamaño de lote:** 64 muestras por lote para equilibrar la velocidad y el uso de memoria.
- **Número de épocas:** 5 iteraciones completas sobre el conjunto de entrenamiento.

Durante el entrenamiento, se utilizaron técnicas para manejar el *overfitting*, como la evaluación periódica en el conjunto de validación y el monitoreo de la pérdida y la precisión.

4.7 Evaluación del Modelo

Para evaluar el rendimiento del modelo, se utilizaron las siguientes métricas:

- **Precisión:** Calculada como la proporción de predicciones correctas sobre el total de muestras evaluadas.
- **Matriz de confusión:** Para visualizar el rendimiento del modelo en cada clase y detectar posibles desbalances o confusiones entre clases.

Se generaron gráficos utilizando `matplotlib` y `seaborn` para representar:

- La pérdida de entrenamiento y validación a lo largo de las épocas.
- La precisión en el conjunto de validación.
- La matriz de confusión con etiquetas de clase.

4.8 Implementación en Arduino

Para mostrar los resultados de la clasificación de sentimientos en una interfaz física, se implementó una comunicación entre el modelo entrenado y un microcontrolador Arduino Uno [8] conectado a una pantalla OLED de 128x64 píxeles.

Comunicación Serial Se estableció una comunicación serial utilizando el módulo `pySerial` en Python y la biblioteca `Serial` en Arduino. El flujo fue el siguiente:

1. El modelo en Python procesa mensajes de texto almacenados en un archivo, realizando la predicción de sentimiento para cada uno.
2. Los resultados se envían al Arduino a través del puerto serial en formato de cadena de caracteres.

Visualización en Pantalla OLED En el Arduino, se utilizó la biblioteca `U8g2lib` para controlar la pantalla OLED [26]. Los mensajes recibidos se muestran en la pantalla, dividiendo el texto en líneas para ajustarse al tamaño de la pantalla y utilizando una fuente legible.

4.9 Integración Completa

La integración permite que, al ejecutar el script en Python, los mensajes sean procesados y las predicciones sean visualizadas en tiempo real en la pantalla OLED. Esto proporciona una representación tangible del análisis de sentimientos realizado por la red neuronal.

4.10 Herramientas Utilizadas

A lo largo del proyecto, se utilizaron las siguientes herramientas y bibliotecas:

- **Python 3.8.10**: Lenguaje de programación principal para el desarrollo del modelo.
- **PyTorch**: Biblioteca para la construcción y entrenamiento de redes neuronales profundas [4].
- **torchtext**: Utilizada para el manejo eficiente de datos textuales y construcción del vocabulario [5].
- **scikit-learn**: Para tareas de preprocesamiento y métricas de evaluación [7].
- **matplotlib** y **seaborn**: Para la generación de gráficos y visualizaciones [27].
- **wordcloud**: Para la creación de nubes de palabras y análisis exploratorio [28].
- **Arduino IDE**: Desarrollo del código para el microcontrolador Arduino [29].
- **U8g2lib**: Biblioteca para el control de pantallas OLED [26].

5 Análisis de Resultados

En esta sección se presentan y analizan los resultados obtenidos tras el entrenamiento y evaluación del modelo de Red Neuronal Recurrente, así como la implementación en el dispositivo Arduino con pantalla OLED.

5.1 Visualización del Conjunto de Datos

Distribución de Clases La Figura 1 muestra la distribución de muestras por categoría en el conjunto de datos combinado. Se observa un balance relativo entre las clases, lo que es favorable para el entrenamiento del modelo sin necesidad de aplicar técnicas de balanceo.

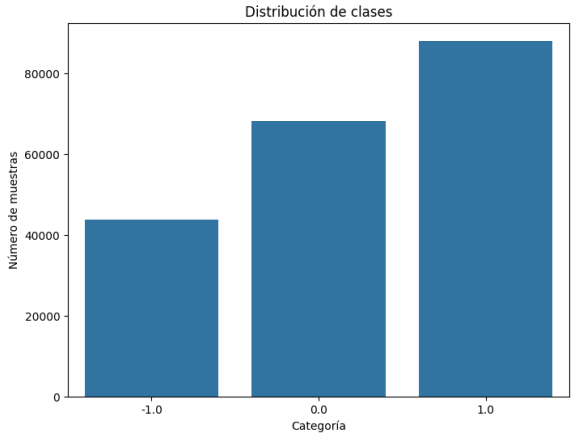

Fig. 1: Distribución de muestras por categoría en el conjunto de datos.

Longitud de los Textos La Figura 2 presenta la distribución de la longitud de los textos en términos del número de palabras. La mayoría de los textos tienen entre 5 y 30 palabras, información que se utilizó para establecer el *padding* y el tamaño máximo de secuencia en el modelo.

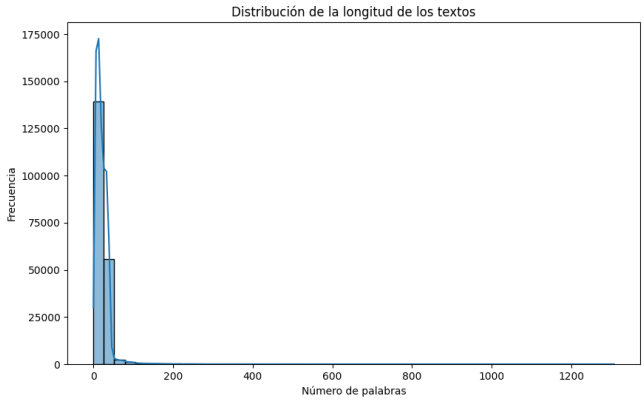

Fig. 2: Distribución de la longitud de los textos en el conjunto de datos.

Nube de Palabras La Figura 3 muestra una nube de palabras generada a partir de todos los textos del conjunto de datos. Las palabras más frecuentes

aparecen con mayor tamaño, proporcionando una visión general de los términos más comunes y potencialmente influyentes en el análisis de sentimientos.

Fig. 3: Nube de palabras del conjunto de datos utilizado.

5.2 Entrenamiento y Validación del Modelo

Pérdida y Precisión Las Figuras 4 y 5 ilustran la evolución de la pérdida y la precisión en los conjuntos de entrenamiento y validación a lo largo de las épocas.

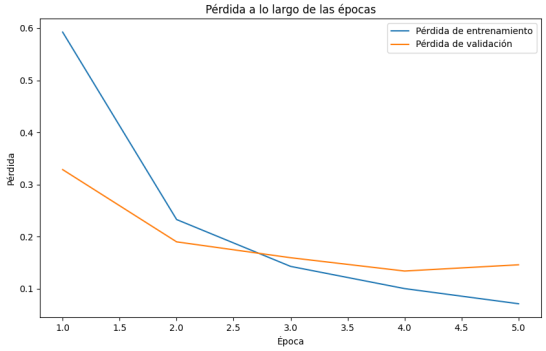

Fig. 4: Evolución de la pérdida durante el entrenamiento y la validación.

Se observa en la Figura 4 que la pérdida disminuye de manera constante en ambas curvas, lo que indica que el modelo está aprendiendo adecuadamente sin signos evidentes de sobreajuste.

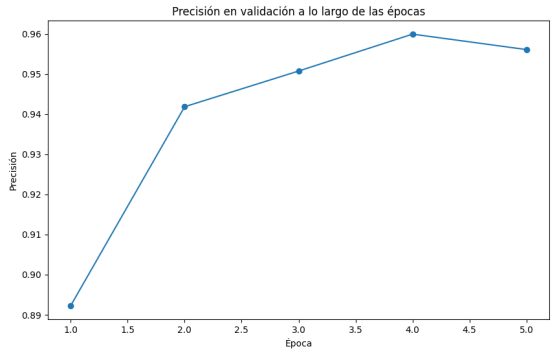

Fig. 5: Evolución de la precisión en el conjunto de validación a lo largo de las épocas.

La Figura 5 muestra que la precisión en el conjunto de validación aumenta progresivamente, alcanzando un valor satisfactorio al final del entrenamiento.

Matriz de Confusión Para evaluar el rendimiento detallado del modelo en cada clase, se calculó la matriz de confusión presentada en la Figura 6.

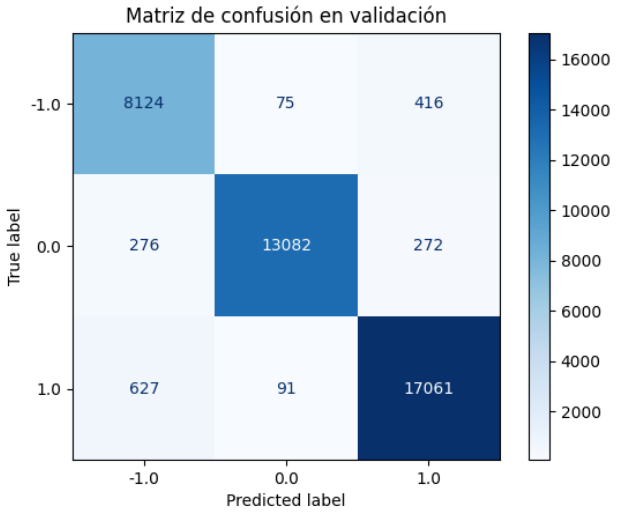

Fig. 6: Matriz de confusión del modelo en el conjunto de validación.

La matriz de confusión revela que el modelo tiene un buen desempeo en la clasificación de las tres categorías, con una mayor precisión en las clases *positivo* y *negativo*. La clase *neutro* presenta ligeramente más confusiones, lo cual es común debido a la ambigüedad inherente en textos de sentimiento neutral.

5.3 Implementación en Arduino

Visualización de Resultados en Pantalla OLED Se seleccionaron tres mensajes de prueba para demostrar la funcionalidad del sistema en tiempo real. Las Figuras 7, 8 y 9 muestran los resultados desplegados en la pantalla OLED del Arduino para cada uno de los sentimientos detectados.

Fig. 7: Resultado de un mensaje clasificado como *positivo* en la pantalla OLED.

En la Figura 7, se observa cómo el sistema muestra correctamente un mensaje con sentimiento positivo, indicando al usuario la clasificación obtenida.

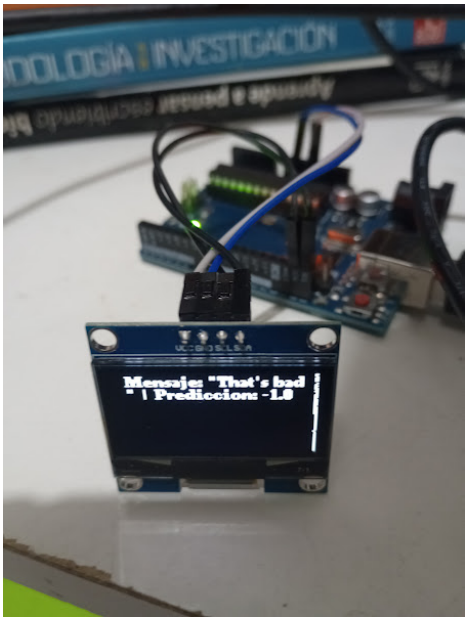

Fig. 8: Resultado de un mensaje clasificado como *negativo* en la pantalla OLED.

La Figura 8 ilustra la presentación de un mensaje con sentimiento negativo, demostrando la capacidad del sistema para identificar y comunicar emociones adversas.

Fig. 9: Resultado de un mensaje clasificado como *neutro* en la pantalla OLED.

Finalmente, en la Figura 9, se muestra un mensaje con sentimiento neutral, completando así la demostración de las tres categorías manejadas por el sistema.

6 Conclusión

En este proyecto, se desarrolló un sistema de análisis y clasificación de sentimientos utilizando una Red Neuronal Recurrente, integrando además una implementación física mediante un Arduino y una pantalla OLED. A través de la preparación y preprocesamiento cuidadoso de los datos, así como de la construcción y entrenamiento del modelo, se demostró la eficacia de las RNN en tareas de procesamiento de lenguaje natural, alcanzando una precisión significativa en la clasificación de sentimientos positivos, negativos y neutros.

La integración con el dispositivo Arduino permitió visualizar los resultados en tiempo real, evidenciando la viabilidad de combinar técnicas avanzadas de aprendizaje automático con hardware accesible para aplicaciones prácticas.

A Anexos

A.1 Código de Entrenamiento de la Red Neuronal

En este anexo se presenta el código fuente utilizado para el entrenamiento de la Red Neuronal Recurrente en Python.

```
import pandas as pd
import re
from sklearn.preprocessing import LabelEncoder
from torchtext.vocab import build_vocab_from_iterator
import torch
from collections import Counter
from torchtext.vocab import Vocab
from sklearn.model_selection import train_test_split
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import accuracy_score
import pandas as pd
import time
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud, STOPWORDS
from sklearn.metrics import confusion_matrix,
    ConfusionMatrixDisplay
import numpy as np


# Carga de los datasets
twitter_df = pd.read_csv('Recurrent_neural_network/DataSet/
    Twitter_Data.csv')
reddit_df = pd.read_csv('Recurrent_neural_network/DataSet/
    Reddit_Data.csv')
# Eliminar filas donde 'clean_text' es nulo
twitter_df = twitter_df.dropna(subset=['clean_text'])
reddit_df = reddit_df.dropna(subset=['clean_text'])
twitter_df = twitter_df.dropna(subset=['category'])
reddit_df = reddit_df.dropna(subset=['category'])


# Combinamos los datasets
data_df = pd.concat([twitter_df, reddit_df], ignore_index=
    True)




# Contar el n mero de muestras por categor a
class_counts = data_df['category'].value_counts()
# Crear una gr fica de barras
plt.figure(figsize=(8,6))
sns.barplot(x=class_counts.index, y=class_counts.values)
plt.title('Distribuci n de clases')
plt.xlabel('Categor a')
plt.ylabel('N mero de muestras')
plt.savefig('RecurenteGraficas/Muestras.png')
```

```
plt.show()


# Calcular la longitud de cada texto
data_df['text_length'] = data_df['clean_text'].apply(lambda x
    : len(x.split()))
# Crear un histograma de la longitud de los textos
plt.figure(figsize=(10,6))
sns.histplot(data_df['text_length'], bins=50, kde=True)
plt.title('Distribuci n de la longitud de los textos')
plt.xlabel('N mero de palabras')
plt.ylabel('Frecuencia')
plt.savefig('RecurenteGraficas/PalabraFrecuencia.png')
plt.show()



# Combinar todos los textos en uno solo
all_text = ' '.join(data_df['clean_text'])
# Definir las palabras de parada (stopwords)
stopwords = set(STOPWORDS)
# Generar la nube de palabras
wordcloud = WordCloud(width=800, height=400, background_color
    ='white', stopwords=stopwords).generate(all_text)
# Mostrar la imagen
plt.figure(figsize=(15,7.5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Nube de palabras de los textos')
plt.savefig('RecurenteGraficas/NubePalabras.png')
plt.show()



def preprocess_text(text):
    # Convertir a min sculas
    text = text.lower()
    # Eliminar URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags
        =re.MULTILINE)
    # Eliminar menciones (@usuario)
    text = re.sub(r'\@\w+','', text)
    # Eliminar caracteres especiales y n meros
    text = re.sub(r'[^A-Za-z\s]+', '', text)
    return text

data_df['clean_text'] = data_df['clean_text'].apply(
    preprocess_text)


```

```
92
93
94 # Codificaci n de etiquetas
95 label_encoder = LabelEncoder()
96 data_df['label'] = label_encoder.fit_transform(data_df['
       category'])
97
98 # Tokenizaci n
99 def tokenize(text):
100     return text.split()
101
102 data_df['tokens'] = data_df['clean_text'].apply(tokenize)
103
104
105
106 # Construcci n del vocabulario
107 counter = Counter()
108 for tokens in data_df['tokens']:
109     counter.update(tokens)
110
111
112 # Funci n generadora para iterar sobre los tokens
113 def yield_tokens(data_iter):
114     for tokens in data_iter:
115         yield tokens
116
117
118 # Construir el vocabulario
119 vocab = build_vocab_from_iterator(
120     yield_tokens(data_df['tokens']),
121     specials=['<unk>', '<pad>'],
122     min_freq=2
123 )
124
125 # Establecer el  ndice por defecto para tokens desconocidos
126 vocab.set_default_index(vocab['<unk>'])
127
128 # Guardar el vocabulario
129 torch.save(vocab, 'vocab.pth')
130
131 # Guardar las clases del LabelEncoder
132 np.save('classes.npy', label_encoder.classes_)
133
134 train_df, val_df = train_test_split(data_df, test_size=0.2,
       random_state=42)
135
136
137
138 class SentimentDataset(Dataset):
139     def __init__(self, df, vocab, max_length):
```

```
140         self.texts = df['tokens']
141         self.labels = df['label']
142         self.vocab = vocab
143         self.max_length = max_length
144 
145     def __len__(self):
146         return len(self.labels)
147 
148     def __getitem__(self, idx):
149         tokens = self.texts.iloc[idx]
150         # Convertir tokens a  ndices
151         indices = [self.vocab[token] for token in tokens]
152         # Padding o truncamiento
153         if len(indices) < self.max_length:
154             indices += [self.vocab['<pad>']] * (self.
                max_length - len(indices))
155         else:
156             indices = indices[:self.max_length]
157         return torch.tensor(indices), torch.tensor(self.
            labels.iloc[idx])
158 
159 max_length = 50  # Puedes ajustar este valor  seg n  tus
    necesidades
160 
161 train_dataset = SentimentDataset(train_df, vocab, max_length)
162 val_dataset = SentimentDataset(val_df, vocab, max_length)
163 
164 train_loader = DataLoader(train_dataset, batch_size=64,
    shuffle=True)
165 val_loader = DataLoader(val_dataset, batch_size=64)
166 
167 
168 class SentimentRNN(nn.Module):
169     def __init__(self, vocab_size, embed_size, hidden_size,
        output_size, num_layers):
170         super(SentimentRNN, self).__init__()
171         self.embedding = nn.Embedding(vocab_size, embed_size,
             padding_idx=vocab['<pad>'])
172         self.lstm = nn.LSTM(embed_size, hidden_size,
            num_layers, batch_first=True)
173         self.fc = nn.Linear(hidden_size, output_size)
174         self.softmax = nn.LogSoftmax(dim=1)
175 
176     def forward(self, x):
177         embeds = self.embedding(x)
178         _, (hidden, _) = self.lstm(embeds)
179         out = self.fc(hidden[-1])
180         return self.softmax(out)
181 
182 vocab_size = len(vocab)
```

```
183 embed_size = 128
184 hidden_size = 256
185 output_size = len(label_encoder.classes_)
186 num_layers = 2
187
188 model = SentimentRNN(vocab_size, embed_size, hidden_size,
        output_size, num_layers)
189
190
191 # Entrenamineto del modelo
192
193
194 criterion = nn.NLLLoss()
195 optimizer = optim.Adam(model.parameters(), lr=0.001)
196
197 # Verificamos si hay GPU disponible
198 device = torch.device('cuda' if torch.cuda.is_available()
        else 'cpu')
199 model.to(device)
200
201 num_epochs = 5  # Ajusta seg n tus necesidades
202
203 # Listas para almacenar las p rdidas y precisiones
204 train_losses = []
205 val_losses = []
206 val_accuracies = []
207
208 for epoch in range(num_epochs):
209     start_time = time.time()
210
211     # Entrenamiento
212     model.train()
213     total_loss = 0
214     for inputs, labels in train_loader:
215         inputs, labels = inputs.to(device), labels.to(device)
216         optimizer.zero_grad()
217         outputs = model(inputs)
218         loss = criterion(outputs, labels)
219         loss.backward()
220         optimizer.step()
221         total_loss += loss.item()
222
223     avg_train_loss = total_loss / len(train_loader)
224     train_losses.append(avg_train_loss)
225
226     # Validaci n
227     model.eval()
228     val_loss = 0
229     all_preds = []
230     all_labels = []
```

```
231     with torch.no_grad():
232         for inputs, labels in val_loader:
233             inputs, labels = inputs.to(device), labels.to(
                    device)
234             outputs = model(inputs)
235             loss = criterion(outputs, labels)
236             val_loss += loss.item()
237             _, preds = torch.max(outputs, 1)
238             all_preds.extend(preds.cpu().numpy())
239             all_labels.extend(labels.cpu().numpy())
240 
241     avg_val_loss = val_loss / len(val_loader)
242     val_losses.append(avg_val_loss)
243 
244     # Calcular precisi n
245     accuracy = accuracy_score(all_labels, all_preds)
246     val_accuracies.append(accuracy)
247 
248     end_time = time.time()
249     epoch_duration = end_time - start_time
250 
251     print(f' poca {epoch+1}/{num_epochs}, P rdida de
            entrenamiento: {avg_train_loss:.4f}, '
252           f'P rdida de validaci n: {avg_val_loss:.4f},
                  Precisi n: {accuracy:.4f}, '
253           f'Tiempo: {epoch_duration:.2f} s')
254 
255 
256 torch.save(model.state_dict(), 'rnn_model.pth')
257 
258 
259 # Crear la gr fica de p rdida
260 plt.figure(figsize=(10,6))
261 plt.plot(range(1, num_epochs+1), train_losses, label='
        P rdida de entrenamiento')
262 plt.plot(range(1, num_epochs+1), val_losses, label='P rdida
        de validaci n')
263 plt.title('P rdida a lo largo de las  pocas')
264 plt.xlabel(' poca')
265 plt.ylabel('P rdida')
266 plt.legend()
267 plt.savefig('RecurenteGraficas/Perdida.png')
268 plt.show()
269 
270 # Crear la gr fica de precisi n
271 plt.figure(figsize=(10,6))
272 plt.plot(range(1, num_epochs+1), val_accuracies, marker='o')
273 plt.title('Precisi n en validaci n a lo largo de las
         pocas')
274 plt.xlabel(' poca')
```

```
plt.ylabel('Precisi n ')
plt.savefig('RecurenteGraficas/Precision.png')
plt.show()


# Calcular la matriz de confusi n
cm = confusion_matrix(all_labels, all_preds)

# Obtener los nombres de las clases
class_names = label_encoder.classes_

# Visualizar la matriz de confusi n
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
    display_labels=class_names)
disp.plot(cmap=plt.cm.Blues)
plt.title('Matriz de confusi n en validaci n ')
plt.savefig('RecurenteGraficas/MatrizConfusion.png')
plt.show()
```

A.2 Código de Prueba de Datos para la Red Neuronal

A continuación se muestra el código utilizado para probar los datos con la red neuronal entrenada.

```
import torch
import torch.nn as nn
import re
import pandas as pd
from torchtext.vocab import build_vocab_from_iterator
from sklearn.preprocessing import LabelEncoder
import numpy as np
import serial
import time

# Definir la clase del modelo (debe coincidir con la
    entrenada)

class SentimentRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size,
        output_size, num_layers):
        super(SentimentRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size,
            padding_idx=vocab['<pad>'])
        self.lstm = nn.LSTM(embed_size, hidden_size,
            num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
```

```
22          embeds = self.embedding(x)
23          _, (hidden, _) = self.lstm(embeds)
24          out = self.fc(hidden[-1])
25          return self.softmax(out)
26
27  # Cargar el vocabulario y el LabelEncoder
28
29  # Cargar el vocabulario guardado
30  vocab = torch.load('vocab.pth')
31
32  # Cargar el LabelEncoder
33  label_encoder = LabelEncoder()
34  label_encoder.classes_ = np.load('classes.npy', allow_pickle=
        True)
35
36  # Cargar el modelo entrenado
37
38  # Par metros del modelo (deben coincidir con los utilizados
        durante el entrenamiento)
39  vocab_size = len(vocab)
40  embed_size = 128
41  hidden_size = 256
42  output_size = len(label_encoder.classes_)
43  num_layers = 2
44
45  # Inicializar el modelo
46  model = SentimentRNN(vocab_size, embed_size, hidden_size,
        output_size, num_layers)
47
48  # Cargar los pesos del modelo entrenado
49  model.load_state_dict(torch.load('rnn_model.pth',
        map_location=torch.device('cpu')))
50  model.eval()
51
52  #Definir las funciones de preprocesamiento
53
54  def preprocess_text(text):
55      # Convertir a min sculas
56      text = text.lower()
57      # Eliminar URLs
58      text = re.sub(r'http\S+|www\S+|https\S+', '', text)
59      # Eliminar menciones (@usuario)
60      text = re.sub(r'\@\w+','', text)
61      # Eliminar caracteres especiales y n meros
62      text = re.sub(r'[^a-z\s]+', '', text)
63      return text
64
65  def tokenize(text):
66      return text.split()
67
```

```
# Leer y preprocesar el archivo de texto


# Leer el archivo de texto (reemplaza 'mensajes.txt' por el nombre de tu archivo)
with open('mensajes.txt', 'r', encoding='utf-8') as f:
    content = f.read()

# Separar los mensajes por puntos
raw_messages = content.strip().split('.')

# Preprocesar y tokenizar los mensajes
processed_messages = []
for msg in raw_messages:
    msg = msg.strip()
    if msg:  # Verificar que el mensaje no est  vac o
        preprocessed = preprocess_text(msg)
        tokens = tokenize(preprocessed)
        processed_messages.append(tokens)

#  Convertir los mensajes en tensores
max_length = 50  # Debe coincidir con el usado durante el entrenamiento

def message_to_tensor(tokens):
    # Convertir tokens a  ndices
    indices = [vocab[token] for token in tokens if token in vocab]
    # Padding o truncamiento
    if len(indices) < max_length:
        indices += [vocab['<pad>']] * (max_length - len(indices))
    else:
        indices = indices[:max_length]
    return torch.tensor(indices).unsqueeze(0)  # A adir dimensi n batch

# Convertir todos los mensajes
message_tensors = [message_to_tensor(tokens) for tokens in processed_messages]

# Realizar las predicciones y enviarlas al Arduino

# Mapear  ndices de etiquetas a categor as originales
label_mapping = {index: label for index, label in enumerate(label_encoder.classes_)}

# Configuraci n de la conexi n serial con Arduino
arduino = serial.Serial('COM8', 9600)  # Cambia 'COM8' por el puerto de tu Arduino
```

```
time.sleep(2)  # Espera para que la conexi n serial se
    estabilice

with torch.no_grad():
    for i, tensor in enumerate(message_tensors):
        outputs = model(tensor)
        _, predicted = torch.max(outputs, 1)
        predicted_label = label_mapping[predicted.item()]

        # Crear el mensaje para enviar al Arduino
        raw_message = raw_messages[i].strip()
        message = f"Mensaje: \"{raw_message}\" | Prediccion:
            {predicted_label}\n"
        print(message)  # Mostrar en la consola

        # Enviar el mensaje al Arduino
        arduino.write(message.encode())
        time.sleep(1)  # Pausa breve entre env os

# Cerrar la conexi n serial
arduino.close()
```

A.3 Código para Arduino IDE

El siguiente código corresponde al programa cargado en el Arduino para la comunicación con la pantalla OLED.

```
#include <Wire.h>
#include <U8g2lib.h>

U8G2_SSD1306_128X64_NONAME_F_HW_I2C u8g2(U8G2_R0, /* reset=*/
    U8X8_PIN_NONE);

void setup() {
    Serial.begin(9600); // Inicia la comunicaci n serial
    u8g2.begin();       // Inicia la pantalla OLED

    u8g2.clearBuffer();
    u8g2.setFont(u8g2_font_ncenB08_tr);
    u8g2.drawStr(0, 20, "Red Neuronal Convolucional");
    u8g2.sendBuffer();
}

void loop() {
    if (Serial.available() > 0) {
        String message = Serial.readStringUntil('\n');  //
            Lee el mensaje completo hasta el salto de l nea

        // Limpiar pantalla
```

```
21          u8g2.clearBuffer();
22          u8g2.setFont(u8g2_font_ncenB08_tr);
23
24          // Configuraci n para dividir el mensaje en l neas
25          int maxCharsPerLine = 20;  // Ajusta seg n la fuente
                usada
26          int lineHeight = 10;       // Ajusta seg n el
                tama o de fuente
27          int y = 10;                // Coordenada inicial en y
28
29          // Mostrar el mensaje divido en l neas
30          for (int i = 0; i < message.length(); i +=
                maxCharsPerLine) {
31              String line = message.substring(i, i +
                    maxCharsPerLine);  // Extrae cada l nea
32              u8g2.drawStr(10, y, line.c_str());
                                       // Muestra la l nea
                    en la pantalla
33              y += lineHeight;
                                                          //
                    Baja a la siguiente l nea
34              if (y > 64 - lineHeight) break;              //
                    Detener si se sale del  rea de la pantalla
35          }
36
37          u8g2.sendBuffer();
38      }
39  }
```

References

1. B. Liu, *Sentiment Analysis and Opinion Mining.* Morgan & Claypool Publishers, 2012.
2. W. Medhat, A. Hassan, and H. Korashy, "Sentiment analysis algorithms and applications: A survey," *Ain Shams Engineering Journal*, vol. 5, no. 4, pp. 1093–1113, 2014.
3. J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.
4. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019, pp. 8026–8037.
5. PyTorch, "Torchtext," https://pytorch.org/text/, 2019, accedido: 12-Oct-2023.
6. Kaggle, "Twitter and reddit sentimental analysis dataset," https://www.kaggle.com/datasets/arkhoshghalb/twitter-sentiment-analysis-hatred-speech, 2019, accedido: 12-Oct-2023.
7. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine

learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
8. Arduino, "Arduino uno rev3," https://store.arduino.cc/arduino-uno-rev3, 2015, accedido: 12-Oct-2023.
9. B. Pang and L. Lee, "Opinion mining and sentiment analysis," *Foundations and Trends in Information Retrieval*, vol. 2, no. 1–2, pp. 1–135, 2008.
10. S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
11. D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1422–1432.
12. G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2016, pp. 260–270.
13. J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 2014, pp. 1532–1543.
14. R. Socher and J. Y. y. C. J. y. M. C. D. y. N. A. y. P. C. Perelygin, Alex y Wu, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2013, pp. 1631–1642.
15. R. E. y. P. P. T. y. H. D. y. N. A. Y. y. P. C. Maas, Andrew L. y Daly, "Learning word vectors for sentiment analysis," *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, vol. 1, pp. 142–150, 2011.
16. D. M. W. Powers, "Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.
17. P. McRoberts, *Arduino for the Cloud: Arduino Yún and Dragino Yún Shield*. Apress, 2018.
18. G. G. Chowdhury, "Natural language processing," *Annual Review of Information Science and Technology*, vol. 37, no. 1, pp. 51–89, 2003.
19. C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.
20. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
21. P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.
22. Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.
23. K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.
24. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
25. G. Miner, J. Elder, T. Fast, R. Hill, R. Nisbet, D. Delen *et al.*, "Practical text mining and statistical analysis for non-structured text data applications," 2012.
26. O. K. Olingo, "U8g2 library for monochrome displays," https://github.com/olikraus/u8g2, 2017, accedido: 12-Oct-2023.
27. J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.
28. A. Mueller, "Wordcloud for python," https://github.com/amueller/word_cloud, 2016, accedido: 12-Oct-2023.

29. Arduino, “Arduino ide,” https://www.arduino.cc/en/Main/Software, 2015, accedido: 12-Oct-2023.
30. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, . Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, vol. 30, 2017, pp. 5998–6008.