

Natural Language Processing with Disaster Tweets

En este notebooks, se utilizó el dataset "Natural Language Processing with Disaster Tweets" de Kaggle con el objetivo de clasificar tweets en dos categorías: desastres reales o no.

Se construyeron y evaluaron dos modelos de clasificación (LSTM) para determinar si un tweet se refiere a un desastre real. Se probaron diferentes algoritmos para comparar su rendimiento y se abordó el contexto a través del análisis de palabras clave y patrones en los datos.

Authors:

- [Andrea Ramirez](#)
- [Adrian Flores](#)

(1) Import Libraries

```
In [ ]: #!/pip install unidecode
```

```
In [ ]: #!/pip install nlpaug
```

```
In [ ]: # Data manipulation and visualization
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from scipy.stats import kurtosis, skew, probplot
from sklearn.metrics import mean_absolute_error, mean_squared_error
import statsmodels.api as sm
import itertools
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from unidecode import unidecode
import tensorflow as tf
from tensorflow import keras
from google.colab import files

# Standard libraries
import warnings
warnings.filterwarnings('ignore')
```

```
# ===== Reproducibility Seed =====
# Set a fixed seed for the random number generator for reproducibility
random_state = 42

# Set matplotlib inline
%matplotlib inline

# Set default figure size
plt.rcParams['figure.figsize'] = (6, 4)

# Define custom color palette
palette = sns.color_palette("viridis", 12)

# Set the style of seaborn
sns.set(style="whitegrid")
```

(2) Data Upload

```
In [ ]: df = pd.read_csv('data/train.csv')
df.head()
```

```
Out[ ]:
```

	id	keyword	location	text	target
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
2	5	NaN	NaN	All residents asked to 'shelter in place' are ...	1
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1

(3) Exploratory Analysis

(1) Descripción General de los Datos

```
In [ ]: # Print the number of records in the DataFrame
print("The given dataset has", df.shape[0], "registers and", df.shape[1], "c
```

The given dataset has 7613 registers and 5 columns.

Observaciones -->

- El conjunto de datos original cuenta con 7613 registros y 5 columnas, lo que indica que tiene una dimensión relativamente pequeña. Cada uno de los 7613 registros representa una observación única, mientras que las 5 columnas corresponden a diferentes características o variables medidas para cada observación, incluyendo el texto de un tweet, una palabra clave asociada y la ubicación desde donde se envió, aunque

estas últimas dos pueden estar en blanco en algunas ocasiones.

Fuente: [Página oficial de Kaggle](#)

```
In [ ]: # Basic information about the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7613 entries, 0 to 7612
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  -
 0   id          7613 non-null   int64
 1   keyword     7552 non-null   object
 2   location    5080 non-null   object
 3   text        7613 non-null   object
 4   target      7613 non-null   int64
dtypes: int64(2), object(3)
memory usage: 297.5+ KB
```

Como se puede observar, se cuentan con 5 columnas o features en este conjunto de datos, siendo estas las que se describen a continuación.

- **id:** un identificador único para cada tweet.
- **text:** el texto del tweet.
- **location:** la ubicación desde donde se envió el tweet (puede estar en blanco).
- **keyword:** una palabra clave particular del tweet (puede estar en blanco).
- **target:** en el archivo `train.csv` solamente, indica si un tweet es sobre un desastre real (1) o no (0).

(2) Clasificación de las Variables

Nombre	Descripción	Tipo de variable
id	Un identificador único para cada tweet	Cuantitativa
text	El texto del tweet	Cualitativa (descriptiva)
location	La ubicación desde donde se envió el tweet	Cualitativa (descriptiva)
keyword	Una palabra clave particular del tweet	Cualitativa (descriptiva)
target	Indica si un tweet es sobre un desastre real o no	Cualitativa (binaria)

Observaciones 💡 -->

- El conjunto de datos posee 3 variables cualitativas descriptivas y 1 de tipo binaria.
- La última variable del conjunto de datos (id) es de tipo cuantitativo.

(3) Exploración y Limpieza Inicial de los Datos

(1) Preprocesamiento de los Datos

```
In [ ]: df.drop(columns=['id'], inplace=True)
```

```
In [ ]: # Download the NLTK stopwords if not already available
nlk.download('stopwords')
# Initialize the PorterStemmer
stemmer = PorterStemmer()
```

```
[nlk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Package stopwords is already up-to-date!
```

```
In [ ]: # Get the list of stopwords from NLTK
stop_words = set(stopwords.words('english'))
```

```
In [ ]: # Function to remove stopwords and apply stemming
def preprocess_text(text):
    # Tokenize the text
    words = text.split()
    # Remove stopwords and apply stemming
    processed_words = [stemmer.stem(word) for word in words if word not in stop_words]
    # Reassemble the text
    return ' '.join(processed_words)
```

```
In [ ]: # Convert all entries to strings
df['text'] = df['text'].astype(str)
# Remove URLs
df['text'] = df['text'].str.replace(r'http\S+|www\S+|https\S+', '', case=False)
# Convert to lowercase
df['text'] = df['text'].str.lower()
# Remove leading/trailing whitespaces
df['text'] = df['text'].str.strip()
# Remove special characters and punctuation (keeping letters, numbers, and spaces)
df['text'] = df['text'].str.replace(r'[\W_]', '', regex=True)
# Remove extra spaces
df['text'] = df['text'].str.replace(r'\s+', ' ', regex=True)
# Apply preprocessing (stopwords removal and stemming)
df['text'] = df['text'].apply(preprocess_text)
```

```
In [ ]: df.isnull().sum()
```

```
Out[ ]:
```

	0
keyword	61
location	2533
text	0
target	0

dtype: int64

```
In [ ]: # Converting the column to a list
column_to_list = df['location'].unique().tolist()
# Convert all elements to strings to avoid TypeError
column_to_list_str = [str(location) for location in column_to_list]

# === WARNING ===
# The detailed listing of unique locations has been commented out to maintain
# If a comprehensive view is required, please uncomment the following lines:
# print("Unique Locations:")
# print(", ".join(column_to_list_str))

print(f"\nTotal number of unique locations: {len(column_to_list_str)}")
```

Total number of unique locations: 3342

```
In [ ]: df = df.drop(['location'],axis=1)
```

```
In [ ]: # Drop rows where 'keyword' column has NaN values
df = df.dropna(subset=['keyword'])
```

```
In [ ]: # Check duplicate rows in dataset
df = df.drop_duplicates()
# Print the number of records in the DataFrame
print("The given dataset has", df.shape[0], "registers and", df.shape[1], "c
```

The given dataset has 6940 registers and 3 columns.

Observaciones 💡 -->

- Primero eliminaremos la columna `id`, ya que no aporta información significativa al conjunto de datos. Dado que cada registro corresponde a una observación única, esta columna no contribuye a la variabilidad o al análisis. Al eliminarla, reducimos la dimensionalidad, lo que puede mejorar la eficiencia de nuestro algoritmo y simplificar la interpretación posterior de este.

- Continuamos el preprocesamiento con dos pasos cruciales para optimizar el análisis de texto: la aplicación de un stemmer y la eliminación de stopwords (o palabras vacías).
- Continúa el preprocesamiento con la realización de varias transformaciones para limpiar y estandarizar el texto en la columna `text`. Primero, convertimos todas las entradas a cadenas de texto para asegurar la consistencia en el tipo de dato. A continuación, eliminamos URLs, que suelen introducir ruido sin aportar valor al análisis. También convertimos el texto a minúsculas y eliminamos los espacios en blanco al inicio y al final de las cadenas para mantener uniformidad. Luego, removemos caracteres especiales y signos de puntuación, dejando solo letras, números y espacios, lo que facilita el análisis posterior. Además, eliminamos espacios extra y normalizamos el texto quitando acentos.
- La columna `location` presenta una cantidad considerable de datos faltantes, aproximadamente un 33% de las entradas son nulas. Además, estas ausencias no son uniformes, ya que algunas se representan con símbolos como `'???'` o contienen caracteres especiales que complican aún más su interpretación debido a que son entradas escritas por usuarios. Esta falta de consistencia en los datos, sumada al hecho de que la variable de ubicación podría introducir sesgos significativos en el modelo, nos lleva a concluir que es mejor eliminar esta columna.
- Seguidamente se realiza imputaciones en la columna `keyword` al eliminar las entradas nulas, esto se justifica con el hecho de que estas entradas representan únicamente alrededor de un 0.9% de nuestros datos.
- Como último paso del preprocesamiento, filtramos los valores duplicados en el conjunto de datos. Al eliminar duplicados, aseguramos que cada registro sea único, lo que mejora la integridad de los datos y optimiza la precisión del modelo al trabajar con un conjunto no redundante.

Nota:

1. El **stemmer** reduce las palabras a su raíz o forma básica, lo que permite agrupar diferentes variaciones de una misma palabra bajo una única representación. Esto no solo disminuye la dimensionalidad del conjunto de datos haciendo que sea más fácil procesar este, sino que también mejora la capacidad de los algoritmos para identificar patrones relevantes en el texto. [\[Referencia\]](#)

2. Por otro lado, la eliminación de **stopwords** filtra palabras comunes que, aunque frecuentes, aportan poco valor semántico al análisis, como "y", "el", "en", entre otras. Al excluir estas palabras, se enfoca el modelo en términos más significativos, lo que puede resultar en una mejora notable en la precisión y eficiencia del análisis textual. Sin embargo, el beneficio más grande es la reducción de dimensionalidad, permitiendo que el entrenamiento sea más rápido. [\[Referencia\]](#)
3. Para algunas de las tareas de procesamiento de lenguaje natural descritas con anterioridad, se optó por implementar nltk, para más información por favor ingresar a la documentación oficial en el siguiente [enlace](#).

(2) Exploración de los Datos

```
In [ ]: # Calculate the length of text entries in the 'text' column.
length = df["text"].apply(len)
# Display descriptive statistics of text lengths.
print("Training Set: Text Length Statistics")
print(length.describe())
```

```
Training Set: Text Length Statistics
count      6940.000000
mean         58.801729
std          22.872899
min           4.000000
25%          42.000000
50%          60.000000
75%          76.000000
max         127.000000
Name: text, dtype: float64
```

Observaciones 💡 -->

- Este pequeño análisis de la longitud de los textos en el conjunto de datos nos permite descubrir varias características interesantes. Con un total de 7,001 registros, la longitud promedio de los textos es de aproximadamente 59 caracteres, lo que indica que la mayoría de las entradas son relativamente cortas. La desviación estándar es de aproximadamente 23 caracteres, lo que sugiere una variabilidad moderada en la longitud de los textos.
- El rango de las longitudes varía significativamente, desde un mínimo de 3 caracteres hasta un máximo de 127 caracteres. El 50% de los textos tienen una longitud de 60 caracteres o menos, con el 25% de los textos por debajo de 41 caracteres y el 75% por debajo de 76 caracteres. Por ende, aunque la mayoría de los textos tienen una longitud similar, hay una presencia de textos mucho más cortos o más largos.

```
In [ ]: # See what are the 10 most frequent values for each of the dataframe columns
for column in df.columns:
    frequency_values = df[column].value_counts().head(10)
    print("Top 10 most frequent values for column '{}':".format(column))
    for index, (value, frequency) in enumerate(frequency_values.items(), start=1):
        print("{:<5} {:<30} {:<10}".format(index, value, frequency))
    print("\n=====")
```

Top 10 most frequent values for column 'keyword':

1	fatalities	45
2	deluge	42
3	armageddon	42
4	damage	41
5	harm	41
6	evacuate	40
7	fear	40
8	body%20bags	40
9	twister	40
10	siren	40

=====

Top 10 most frequent values for column 'text':

1	angri woman openli accus nema steal relief materi meant idp angri inte rn displac wom 2
2	cindi noonancindynoonanheartbreak baltimor riot yahistor undergroundra ilraod 2
3	feel like sink low selfimag take quiz 2
4	horribl sink feel youû²v home phone realis 3g whole time 2
5	break obama offici gave muslim terrorist weapon use texa attack 2
6	trafford centr film fan angri odeon cinema evacu follow fals fire alar m 2
7	new evacu order 25 home danger hwi 8 fire near roosevelt wash koin6new 2
8	drunk meal 101 cook your total oblitter 2
9	wacko like michelebachman predict world soon oblitter burn firey infern o cant accept globalwarm hello 2
10	choke hazard prompt recal kraft chees singl 2

=====

Top 10 most frequent values for column 'target':

1	0	4105
2	1	2835

=====

```
In [ ]: # Create a figure with a single subplot
fig, ax = plt.subplots(figsize=(6, 4), dpi=100) # Use subplots() to get the
plt.tight_layout(pad=3.0) # Add padding around the plot

# Calculate target counts and percentages
target_counts = df['target'].value_counts()
total_count = target_counts.sum()
labels = [f'Non-Disaster Tweets ({target_counts[0]/total_count:.1%})',
          f'Disaster Tweets ({target_counts[1]/total_count:.1%})']

# Define colors from the viridis palette
```



```
colors = [palette[0], palette[6]] # Assign specific colors to each slice

# Plot pie chart for target distribution with customized colors
wedges, texts, autotexts = ax.pie(target_counts, autopct='%1.1f%%', startang

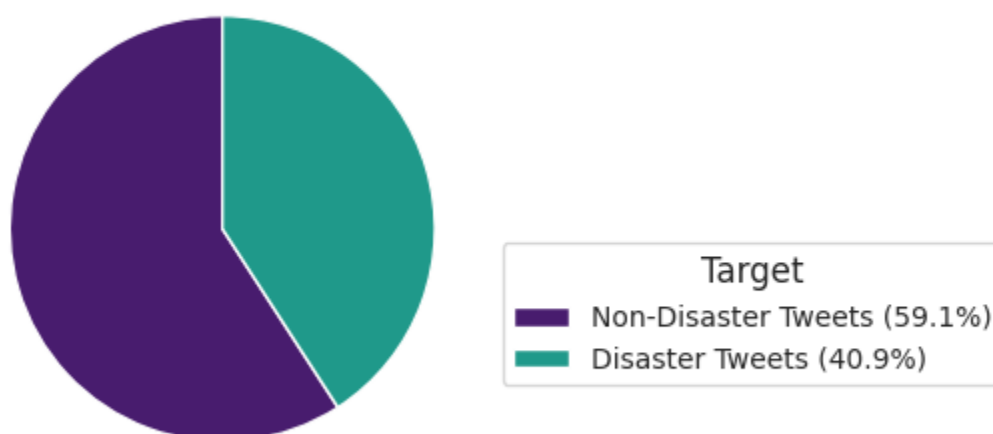
# Set the title
ax.set_title('Target Distribution in Training Set')

# Remove the default labels (texts)
for text in texts:
    text.set_visible(False)
for autotext in autotexts:
    autotext.set_visible(False)

# Add a legend with a specific location and size
ax.legend(wedges, labels, title='Target', loc='best', bbox_to_anchor=(1, 0.5

# Display the plot
plt.show()
```

Target Distribution in Training Set



Observaciones 💡 -->

- Como es posible observar, nuestras tablas de frecuencia muestran que el conjunto de datos está desbalanceado en términos de la variable 'target': el **valor 0** (que indica la ausencia de un desastre) aparece con mucha más frecuencia (4105 casos) que el **valor 1** (que indica la presencia de un desastre) con 2835 casos. Este desbalance podría afectar el rendimiento de nuestro modelo pero primero evaluaremos su desempeño antes de tomar decisiones.
- Para la columna keyword, esta simple tabla de frecuencia no nos dice lo suficiente así que haremos un análisis más profundo.

```
In [ ]: # Pairing 0's with their most frequent keywords
target_0_keywords = df[df['target'] == 0]['keyword'].value_counts().head(10)
print("Top 10 most frequent keywords for target '0':")
for index, (keyword, frequency) in enumerate(target_0_keywords.items(), start=1):
    print(f"{index:<5} {keyword:<30} {frequency:<10}")
print("\n=====")

# Pairing 1's with their most frequent keywords
target_1_keywords = df[df['target'] == 1]['keyword'].value_counts().head(10)
print("Top 10 most frequent keywords for target '1':")
for index, (keyword, frequency) in enumerate(target_1_keywords.items(), start=1):
    print(f"{index:<5} {keyword:<30} {frequency:<10}")
```

Top 10 most frequent keywords for target '0':

1	body%20bags	39
2	harm	37
3	armageddon	37
4	deluge	36
5	ruin	36
6	wrecked	36
7	twister	35
8	fear	35
9	siren	35
10	panic	34

=====

Top 10 most frequent keywords for target '1':

1	evacuated	31
2	debris	31
3	earthquake	30
4	derailment	29
5	wildfire	29
6	nuclear%20disaster	28
7	suicide%20bombing	28
8	evacuation	28
9	buildings%20on%20fire	27
10	mass%20murder	27

```
In [ ]: # Create a figure with increased size for better visual clarity
fig = plt.figure(figsize=(6, 4), dpi=100) # Adjusted figsize for better vis

# Group by 'keyword' and calculate mean target values
grouped_df = df.groupby('keyword')['target'].transform('mean')

# Sort DataFrame by calculated mean target values
sorted_df = df.assign(mean_target=grouped_df).sort_values(by='mean_target',

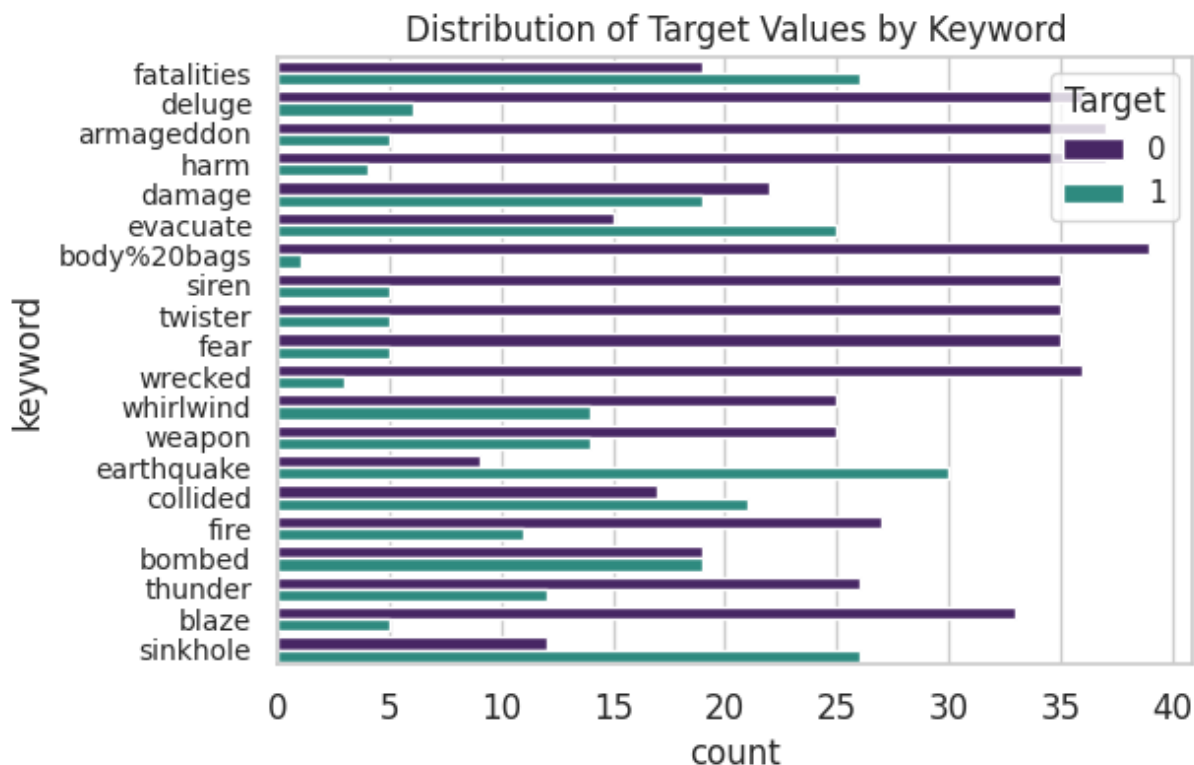
# Optional: Display only the top N keywords for better visibility
top_n = 20
top_keywords = sorted_df['keyword'].value_counts().index[:top_n]
filtered_df = sorted_df[sorted_df['keyword'].isin(top_keywords)]

# Create a count plot with horizontal bars
sns.countplot(y='keyword', hue='target', data=filtered_df, order=top_keywords)

# Customize plot appearance
```

```
plt.tick_params(axis='x', labelsz=12)
plt.tick_params(axis='y', labelsz=10)
plt.legend(title='Target', loc='upper right')
plt.title('Distribution of Target Values by Keyword')

# Display the plot
plt.show()
```



Observaciones 💡 -->

- Se evidencia que para el target 0, los términos más frecuentes como "body bags" "harm" y "armageddon" están relacionados con eventos graves o de alta magnitud, pero en el contexto de la ausencia de un desastre, estos podrían estar mal clasificados o reflejar un contexto de preocupación general. Por otro lado, para el target 1, palabras como "evacuated" "debris" y "earthquake" se relacionan claramente con eventos de emergencia reales.
- En general, los bigramas o trigramas consisten en obtener los tokens o palabras consecutivas. Esta técnica está más orientada a aplicaciones como el análisis de sentimientos o la predicción de texto, pero se considera que podría llegar a aplicarse en este contexto. Para esto, no obstante, primero es importante adquirir ¡aún más información acerca de nuestro conjunto de datos! [\[Referencia\]](#)

(3) Análisis Visual Preliminar de los Datos

```
In [ ]: # Converting the column to a list
column_to_list = df['keyword'].unique().tolist()
# Convert all elements to strings to avoid TypeError
column_to_list_str = [str(keyword) for keyword in column_to_list]
print(f"Number of unique values in keyword = {len(column_to_list_str)}")
```

Number of unique values in keyword = 221

(1) Generación de Bigramas

```
In [ ]: from nltk.util import bigrams
from nltk.util import trigrams
from nltk.tokenize import word_tokenize
from collections import Counter
```

```
In [ ]: # Ensure that the necessary NLTK resources are downloaded
nltk.download('punkt')
```

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Package punkt is already up-to-date!

Out[]: True

```
In [ ]: # Tokenize text and generate bigrams
def generate_bigrams(text):
    tokens = word_tokenize(text)
    return list(bigrams(tokens)) # Generate bigrams

# Function to generate trigrams
def generate_trigrams(text):
    tokens = word_tokenize(text)
    return list(trigrams(tokens))
```

```
In [ ]: # Apply bigram generation to the 'text' column and store results in a separate DataFrame
n_gram = pd.DataFrame({
    'bigrams': df['text'].apply(generate_bigrams)
})

# Separate DataFrames for disaster and non-disaster tweets
disaster_df = df[df['target'] == 1].copy()
non_disaster_df = df[df['target'] == 0].copy()

# Add bigrams to the separated DataFrames
disaster_df['bigrams'] = n_gram.loc[disaster_df.index, 'bigrams']
non_disaster_df['bigrams'] = n_gram.loc[non_disaster_df.index, 'bigrams']

# Function to count bigrams
def count_bigrams(df):
    all_bigrams = [bigram for sublist in df['bigrams'] for bigram in sublist]
    return Counter(all_bigrams)

# Count bigrams for each category
disaster_bigram_counts = count_bigrams(disaster_df)
```

```

non_disaster_bigram_counts = count_bigrams(non_disaster_df)

# Convert bigram counts to DataFrame for plotting
def bigram_counts_to_df(bigram_counts):
    bigram_df = pd.DataFrame(bigram_counts.items(), columns=['Bigram', 'Count'])
    bigram_df['Bigram'] = bigram_df['Bigram'].apply(lambda x: ' '.join(x))
    return bigram_df.sort_values(by='Count', ascending=False).head(20) # Di

# Get top bigrams for each category
disaster_bigram_df = bigram_counts_to_df(disaster_bigram_counts)
non_disaster_bigram_df = bigram_counts_to_df(non_disaster_bigram_counts)

# Create plots
fig, axes = plt.subplots(ncols=2, figsize=(12, 5), dpi=100)
plt.tight_layout(pad=4.0)

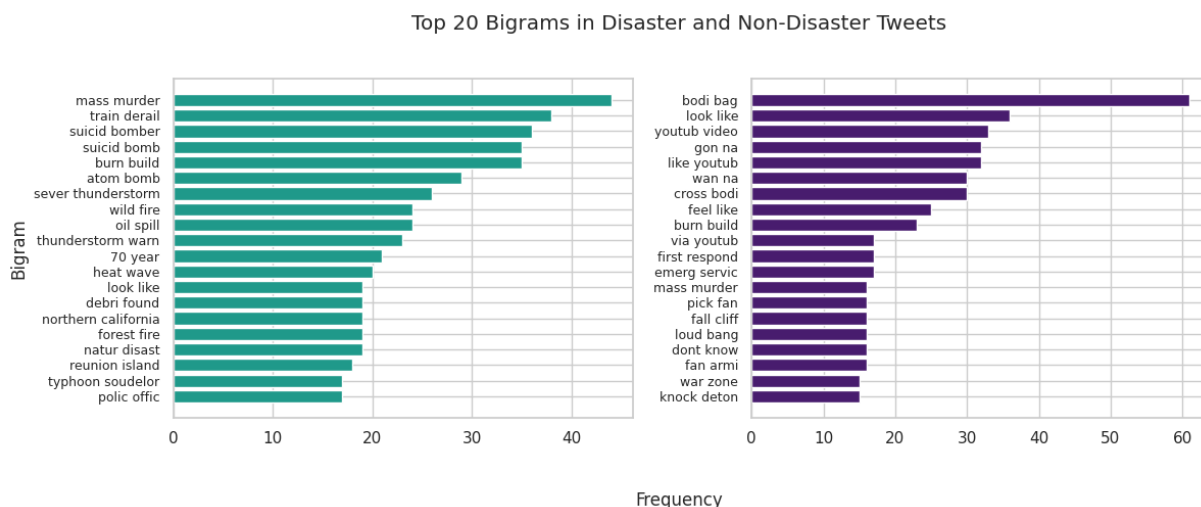
# Plot for disaster tweets
axes[0].barh(disaster_bigram_df['Bigram'], disaster_bigram_df['Count'], color='teal')
axes[0].set_ylabel('Bigram')
axes[0].invert_yaxis() # Invert y-axis to have the most frequent bigram on top
axes[0].tick_params(axis='y', labelsize=9) # Set fontsize for y-tick labels

# Plot for non-disaster tweets
axes[1].barh(non_disaster_bigram_df['Bigram'], non_disaster_bigram_df['Count'], color='purple')
axes[1].invert_yaxis() # Invert y-axis to have the most frequent bigram on top
axes[1].tick_params(axis='y', labelsize=9) # Set fontsize for y-tick labels

# Set a common title and x-axis label for the entire figure
fig.suptitle('Top 20 Bigrams in Disaster and Non-Disaster Tweets')
fig.supxlabel('Frequency', fontsize=12)

# Display the plots
plt.show()

```



(1) Generación de Trigramas

```

In [ ]: # Apply trigram generation to the 'text' column and store results in a separate DataFrame
n_gram = pd.DataFrame({
    'trigrams': df['text'].apply(generate_trigrams)
})

```

```
# Separate DataFrames for disaster and non-disaster tweets
disaster_df = df[df['target'] == 1].copy()
non_disaster_df = df[df['target'] == 0].copy()

# Add trigrams to the separated DataFrames
disaster_df['trigrams'] = n_gram.loc[disaster_df.index, 'trigrams']
non_disaster_df['trigrams'] = n_gram.loc[non_disaster_df.index, 'trigrams']

# Function to count trigrams
def count_trigrams(df):
    all_trigrams = [trigram for sublist in df['trigrams'] for trigram in sublist]
    return Counter(all_trigrams)

# Count trigrams for each category
disaster_trigram_counts = count_trigrams(disaster_df)
non_disaster_trigram_counts = count_trigrams(non_disaster_df)

# Convert trigram counts to DataFrame for plotting
def trigram_counts_to_df(trigram_counts):
    trigram_df = pd.DataFrame(trigram_counts.items(), columns=['Trigram', 'Count'])
    trigram_df['Trigram'] = trigram_df['Trigram'].apply(lambda x: ' '.join(x.split()))
    return trigram_df.sort_values(by='Count', ascending=False).head(20)

# Get top trigrams for each category
disaster_trigram_df = trigram_counts_to_df(disaster_trigram_counts)
non_disaster_trigram_df = trigram_counts_to_df(non_disaster_trigram_counts)

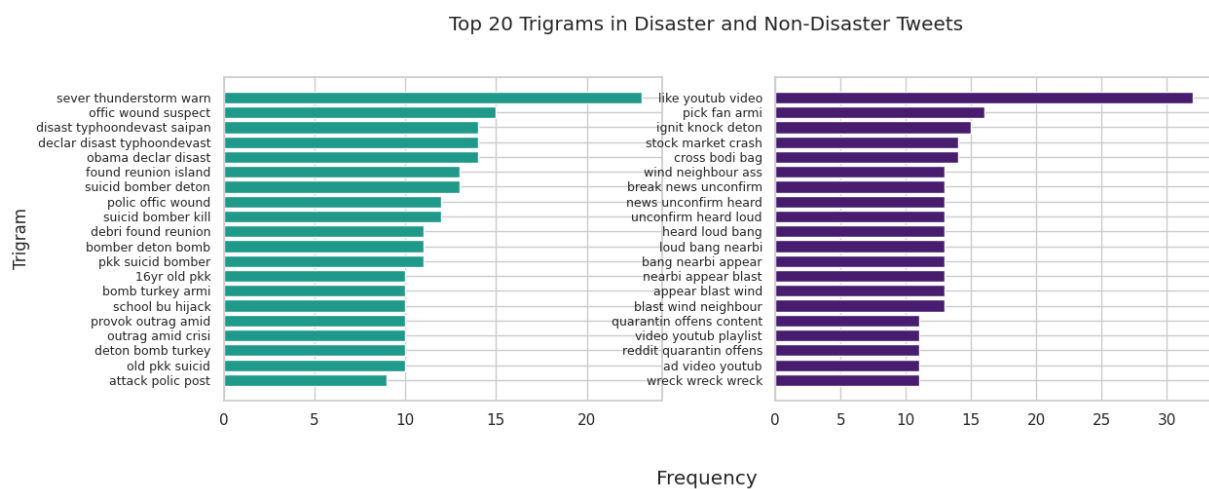
# Create plots
fig, axes = plt.subplots(ncols=2, figsize=(12, 5), dpi=100)
plt.tight_layout(pad=4.0)

# Plot for disaster tweets
axes[0].barh(disaster_trigram_df['Trigram'], disaster_trigram_df['Count'], color='red')
axes[0].set_ylabel('Trigram')
axes[0].invert_yaxis() # Invert y-axis to have the most frequent trigram on top
axes[0].tick_params(axis='y', labelsize=9) # Set fontsize for y-tick labels

# Plot for non-disaster tweets
axes[1].barh(non_disaster_trigram_df['Trigram'], non_disaster_trigram_df['Count'], color='blue')
axes[1].invert_yaxis() # Invert y-axis to have the most frequent trigram on top
axes[1].tick_params(axis='y', labelsize=9) # Set fontsize for y-tick labels

# Set a common title and x-axis label for the entire figure
fig.suptitle('Top 20 Trigrams in Disaster and Non-Disaster Tweets')
fig.supxlabel('Frequency')

# Display the plots
plt.show()
```



Observaciones 💡 -->

- En el análisis, se generaron bigramas y trigramas a partir de los tweets para estudiar las combinaciones más frecuentes de palabras en los mensajes clasificados como desastres y no desastres. Primero, se tokenizó el texto de cada tweet, dividiéndolo en palabras individuales. Luego, se formaron bigramas y trigramas, que son secuencias consecutivas de dos o tres palabras, respectivamente. Estos n-gramas se aplicaron a los tweets, y se contó la frecuencia de cada bigrama dentro de las categorías de tweets de desastre y no desastre.
- Posteriormente, se utilizó un contador para sumar las ocurrencias de cada bigrama en ambas categorías de tweets. Los bigramas más frecuentes se identificaron y se organizaron en un DataFrame, mostrando los 20 principales bigramas de cada categoría. Estos datos se visualizaron mediante gráficos de barras, lo que permitió comparar visualmente la prevalencia de ciertas combinaciones de palabras en tweets de desastres frente a tweets no relacionados con desastres.
- En cuanto a los resultados, observamos que el bigrama más frecuente es "mass murder," lo cual coincide con los resultados obtenidos en la clasificación de palabras clave, donde "mass murder" ocupa la décima posición de los más repetidos para tweets de desastre. Otros bigramas destacados incluyen "suicide bombing" (notando que las palabras se han reducido a su raíz) y "train derail," entre otros.
- Los resultados de los trigramas muestran una gran similitud con los obtenidos para los bigramas, siendo "severe thunderstorm warning" el más frecuente en los tweets de desastre. Aunque estos trigramas proporcionan información relevante sobre los desastres, es importante considerar que no añaden información adicional significativa respecto a los bigramas.

(4) Model Generation: LSTM Approach

- Para entrenar una red neuronal que clasifique datos secuenciales, se puede optar por una red LSTM (Long Short-Term Memory). Este tipo de red permite procesar datos en secuencia y realizar predicciones basadas en cada paso de la secuencia. Dado que el texto también es una forma de dato secuencial, las redes LSTM son herramientas muy efectivas para este tipo de situaciones, ya que son capaces de captar dependencias y contextos a lo largo de la secuencia. En base a esto se justifica la decisión de implementar un modelo LSTM [\[Referencia\]](#).

```
In [ ]: from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Embedding, Bidirectional, Dense, LSTM, D
        from tensorflow.keras.preprocessing.text import Tokenizer
        from tensorflow.keras.preprocessing.sequence import pad_sequences
        from sklearn.model_selection import train_test_split
```

(1) Embedding and Tokenization of Tweets

```
In [ ]: texts = df['text'].values
        labels = df['target'].values
```

```
In [ ]: # Tokenization and Padding
        max_words = 10000 # Maximum number of unique words
        max_len = 127 # Maximum length of sequences
```

Nota ! ¡Es aquí donde nos sirve nuestro pequeño análisis de la longitud de los textos!

```
In [ ]: tokenizer = Tokenizer(num_words=max_words)
        tokenizer.fit_on_texts(texts)
```

```
In [ ]: sequences = tokenizer.texts_to_sequences(texts)
```

```
In [ ]: X = pad_sequences(sequences, maxlen=max_len)
```

Observaciones 💡 -->

En este paso, se realiza un preprocesamiento del texto para que pueda ser ingresado al modelo. Este proceso incluye:

1. **Tokenización del Texto:** Primero, se crea un tokenizador que convierte el texto en una secuencia de números. Cada palabra en el texto es reemplazada por un número único, según su frecuencia en el conjunto de datos. Esto permite al modelo trabajar con datos numéricos en lugar de texto crudo.
2. **Conversión a Secuencias:** Una vez tokenizado el texto, se transforma en secuencias de números. Cada secuencia representa un texto en forma de una lista de números,

donde cada número corresponde a una palabra específica en el vocabulario.

3. **Relleno de Secuencias:** Las secuencias resultantes se ajustan para que todas tengan la misma longitud. Esto es necesario porque los modelos de aprendizaje automático requieren entradas de tamaño uniforme. Se agrega relleno a las secuencias más cortas y se recorta a las más largas para garantizar que todas tengan la misma dimensión.

Consideraciones Adicionales

Se ha decidido descartar la columna `keyword` del conjunto de datos. La razón es que la palabra en esta columna ya está incluida en el texto del tweet, por lo que no proporciona información adicional. Al no considerarla, se reduce la dimensionalidad del conjunto de datos sin perder la información relevante, ya que el texto del tweet ya contiene la información necesaria.

Además, el objetivo final es **"recibir el texto de un tweet sin preprocesar y determinar si se refiere a un desastre natural o no"**. En otras palabras, el problema se resuelve a partir de un texto plano, y por ende no tiene sentido incluir la columna `keyword`.

(2) Data Splitting Process

```
In [ ]: # Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2)
```

(3) Model Development and Creation

(1) Initial Iteration (LSTM)

```
In [ ]: epochs = 15 # For better result increase the epochs
batch_size = 64
```

```
In [ ]: # Define the model
model = Sequential()
model.add(Embedding(input_dim=max_words, output_dim=16, input_length=max_len))
model.add(Bidirectional(LSTM(16, return_sequences=True)))
model.add(Bidirectional(LSTM(16)))
model.add(Dense(18, activation='relu'))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Dense(9, activation='relu'))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Dense(1, activation='sigmoid')) # Binary classification

# Explicitly build the model
model.build(input_shape=(None, max_len))
```

```
In [ ]: # Define the optimizer
opt = keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.9, beta_2=0.999)
```

```
In [ ]: # Compile the model
model.compile(optimizer = opt, loss = "binary_crossentropy" , metrics=["accu
```

```
In [ ]: model.summary()
```

Model: "sequential"

Layer (type)	Output Shape
embedding (Embedding)	(None, 127, 16)
bidirectional (Bidirectional)	(None, 127, 32)
bidirectional_1 (Bidirectional)	(None, 32)
dense (Dense)	(None, 18)
dropout (Dropout)	(None, 18)
batch_normalization (BatchNormalization)	(None, 18)
dense_1 (Dense)	(None, 9)
dropout_1 (Dropout)	(None, 9)
batch_normalization_1 (BatchNormalization)	(None, 9)
dense_2 (Dense)	(None, 1)

Total params: 171,379 (669.45 KB)

Trainable params: 171,325 (669.24 KB)

Non-trainable params: 54 (216.00 B)

Observaciones 💡 -->

El modelo definido es una arquitectura para el procesamiento de datos secuenciales, especialmente diseñada para clasificación binaria. La red comienza con una capa de Embedding, que transforma las secuencias de números en vectores de dimensión 16, capturando las representaciones semánticas de las palabras en un espacio denso. Luego, se emplea una capa Bidirectional LSTM con 16 unidades, seguida de otra capa Bidirectional LSTM con la misma cantidad de unidades, para capturar dependencias tanto a corto como a largo plazo en las secuencias de texto. Estas capas LSTM son bidireccionales, lo que permite que la red considere la información de la secuencia en ambas direcciones.


Posteriormente, se añaden capas Dense con 18 y 9 neuronas, respectivamente, utilizando la activación ReLU para introducir no linealidades y ayudar al modelo a aprender representaciones complejas. Cada una de estas capas está seguida de una capa de Dropout con una tasa del 50% y una capa de BatchNormalization, lo que ayuda a prevenir


el sobreajuste y a normalizar las activaciones para mejorar la estabilidad del entrenamiento. Finalmente, la red culmina en una capa densa con una sola neurona y activación sigmoid, que realiza la clasificación binaria, determinando si el texto se refiere a un desastre natural o no. La estructura del modelo está explícitamente construida para manejar entradas de longitud fija, definida por `max_len`.


(1) Entrenamiento


```
In [ ]: from tensorflow.keras.callbacks import EarlyStopping
        early_stopping = EarlyStopping(monitor='val_loss', patience=6, restore_best_


In [ ]: # Fit the model
        history = model.fit(X_train, y_train,
                             epochs = epochs, validation_data = (X_test, y_test), bat
                             callbacks=[early_stopping])
```


Epoch 1/15
65/65  **23s** 210ms/step - accuracy: 0.5064 - loss: 0.8487
- val_accuracy: 0.5879 - val_loss: 0.6775


Epoch 2/15
65/65  **20s** 199ms/step - accuracy: 0.5104 - loss: 0.8387
- val_accuracy: 0.5879 - val_loss: 0.6834


Epoch 3/15
65/65  **20s** 182ms/step - accuracy: 0.5661 - loss: 0.7754
- val_accuracy: 0.5879 - val_loss: 0.6860


Epoch 4/15
65/65  **12s** 184ms/step - accuracy: 0.6075 - loss: 0.7270
- val_accuracy: 0.5879 - val_loss: 0.6859


Epoch 5/15
65/65  **14s** 208ms/step - accuracy: 0.6803 - loss: 0.6203
- val_accuracy: 0.5879 - val_loss: 0.6784


Epoch 6/15
65/65  **20s** 199ms/step - accuracy: 0.7065 - loss: 0.5864
- val_accuracy: 0.5879 - val_loss: 0.6663


Epoch 7/15
65/65  **13s** 203ms/step - accuracy: 0.7241 - loss: 0.5718
- val_accuracy: 0.6189 - val_loss: 0.6449


Epoch 8/15
65/65  **20s** 198ms/step - accuracy: 0.7456 - loss: 0.5384
- val_accuracy: 0.6679 - val_loss: 0.6190


Epoch 9/15
65/65  **20s** 197ms/step - accuracy: 0.7581 - loss: 0.5251
- val_accuracy: 0.6816 - val_loss: 0.6004


Epoch 10/15
65/65  **20s** 182ms/step - accuracy: 0.7739 - loss: 0.5114
- val_accuracy: 0.7269 - val_loss: 0.5658

Epoch 11/15
65/65  **21s** 192ms/step - accuracy: 0.7792 - loss: 0.4953
- val_accuracy: 0.7205 - val_loss: 0.5670

Epoch 12/15
65/65  **13s** 197ms/step - accuracy: 0.7893 - loss: 0.4907
- val_accuracy: 0.7579 - val_loss: 0.5314

Epoch 13/15
65/65  **19s** 181ms/step - accuracy: 0.7903 - loss: 0.4731
- val_accuracy: 0.7421 - val_loss: 0.5370

Epoch 14/15
65/65  **12s** 182ms/step - accuracy: 0.8023 - loss: 0.4632
- val_accuracy: 0.7406 - val_loss: 0.5408

Epoch 15/15
65/65  **21s** 195ms/step - accuracy: 0.7990 - loss: 0.4646
- val_accuracy: 0.7586 - val_loss: 0.5215

Observaciones 💡 -->

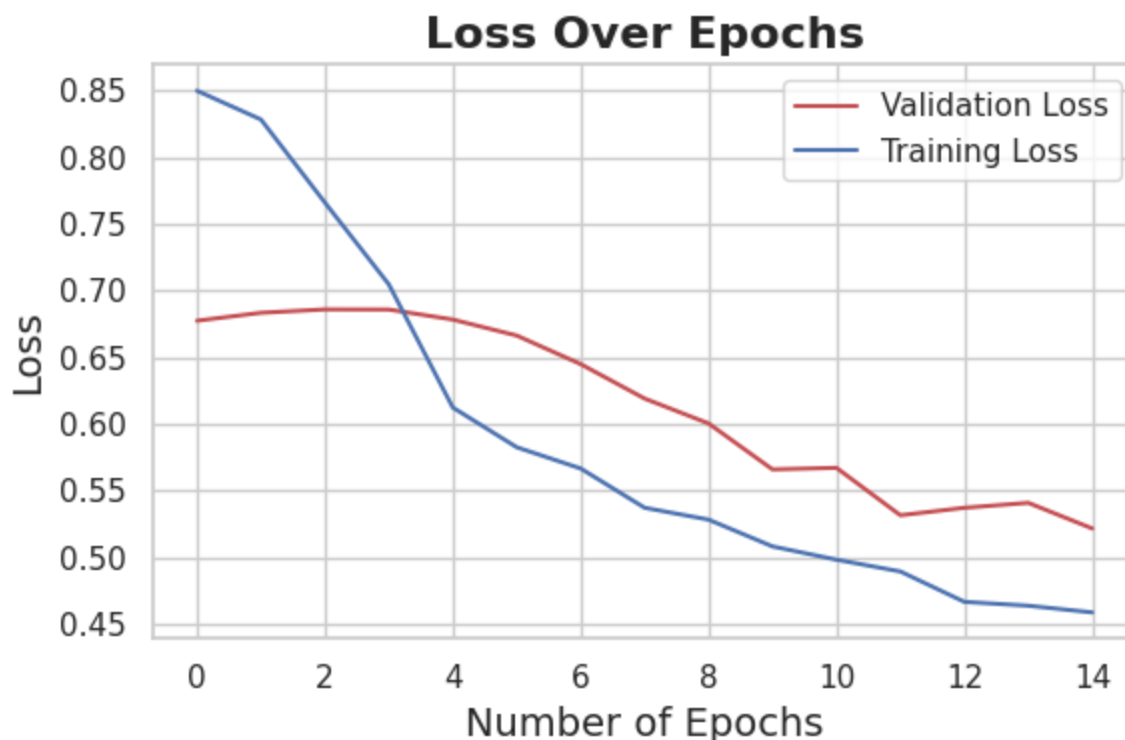
- Durante el entrenamiento del modelo a lo largo de 15 épocas, se observa una tendencia general positiva tanto en la precisión como en la pérdida. En las primeras épocas, la precisión de entrenamiento comienza en alrededor del 50% y mejora gradualmente, alcanzando un 79.9% al final del entrenamiento. La pérdida de entrenamiento también

muestra una disminución constante, comenzando en 0.8487 y reduciéndose a 0.4646, lo que indica que el modelo está aprendiendo a clasificar los datos de manera más efectiva a través del tiempo.

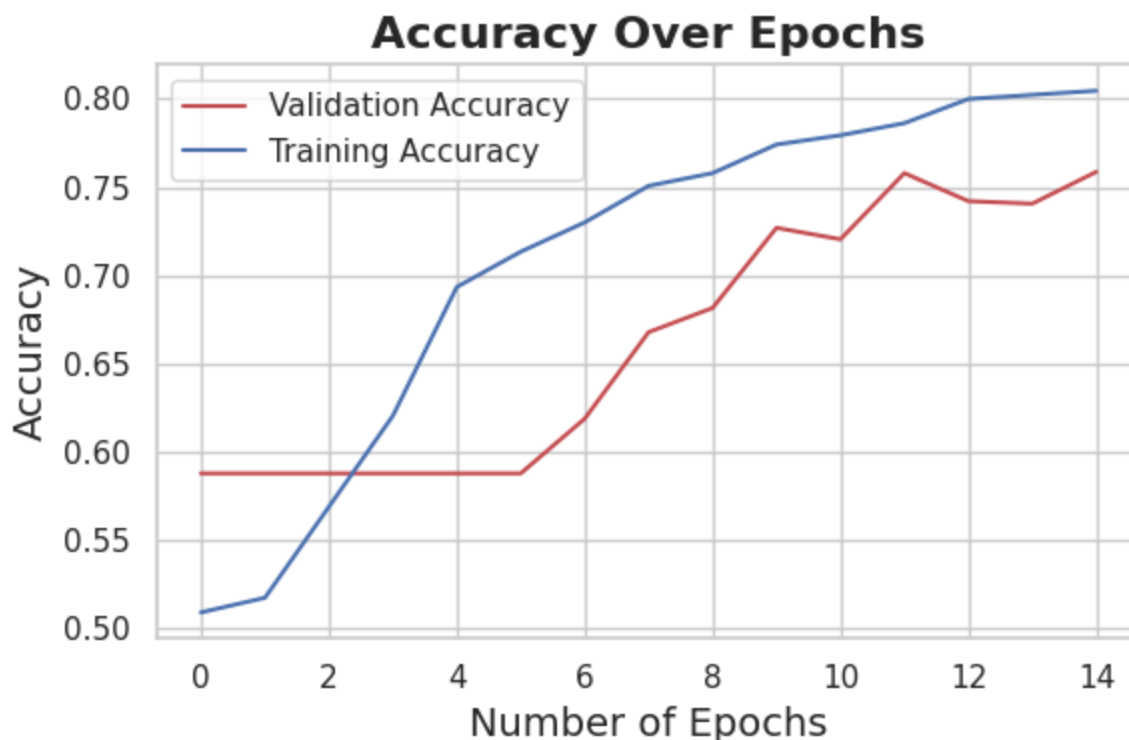
- En cuanto a la precisión y pérdida de validación, se nota una mejora significativa a partir de la séptima época. La precisión de validación comienza en 0.5879 y aumenta hasta un 75.86% al final, mientras que la pérdida de validación disminuye de 0.6775 a 0.5215. Esto permite **creer** que el modelo no solo está aprendiendo bien con los datos de entrenamiento, sino que también está generalizando efectivamente a los datos de validación. Sin embargo, es importante analizar otro tipo de métricas para llegar a conclusiones acerca del desempeño de nuestro modelo.

(2) Evaluación de Desempeño

```
In [ ]: plt.figure(figsize=(6, 4))
# Plot train and validation loss with a color palette
sns.lineplot(x=range(len(history.history['val_loss'])), y=history.history['v
sns.lineplot(x=range(len(history.history['loss'])), y=history.history['loss'
# Add titles and labels with improved styling
plt.title("Loss Over Epochs", fontsize=16, weight='bold')
plt.xlabel("Number of Epochs", fontsize=14)
plt.ylabel("Loss", fontsize=14)
# Add gridlines and legend
plt.grid(True)
plt.legend()
# Show the plot
plt.tight_layout()
plt.show()
```



```
In [ ]: plt.figure(figsize=(6, 4))
# Plot train and validation loss with a color palette
sns.lineplot(x=range(len(history.history['val_accuracy'])), y=history.history['val_accuracy'], color='red')
sns.lineplot(x=range(len(history.history['accuracy'])), y=history.history['accuracy'], color='blue')
# Add titles and labels with improved styling
plt.title("Accuracy Over Epochs", fontsize=16, weight='bold')
plt.xlabel("Number of Epochs", fontsize=14)
plt.ylabel("Accuracy", fontsize=14)
# Add gridlines and legend
plt.grid(True)
plt.legend()
# Show the plot
plt.tight_layout()
plt.show()
```



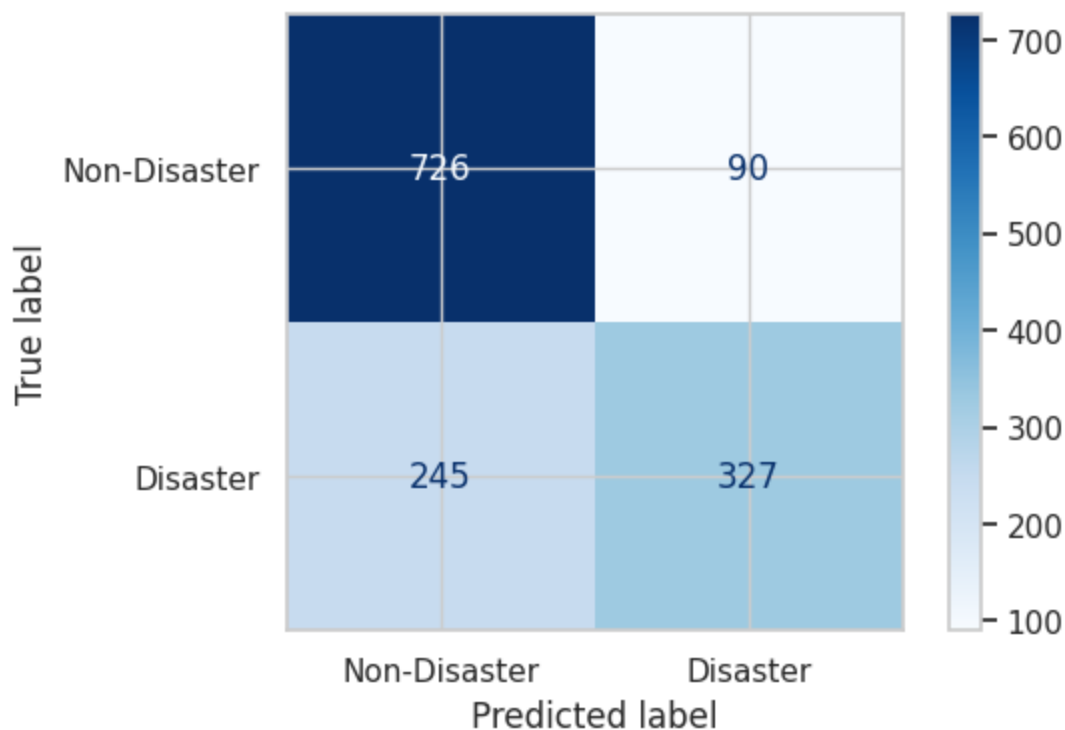
```
In [ ]: from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Generate predictions
y_pred_probs = model.predict(X_test) # Replace X_test with your test data
y_pred = (y_pred_probs > 0.5).astype(int) # Binary classification

# Compute confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Visualize confusion matrix
disp = ConfusionMatrixDisplay(conf_matrix, display_labels=['Non-Disaster', 'Disaster'])
disp.plot(cmap=plt.cm.Blues, values_format='d')

# Show the plot
plt.show()
```

44/44 ————— 3s 46ms/step



- El modelo ha identificado correctamente 726 casos como negativos (tweets de no desastre) y 327 casos como positivos (tweets de desastre).
- Sin embargo, ha cometido 90 errores al clasificar casos negativos como positivos (falsos positivos) y 245 errores al clasificar casos positivos como negativos (falsos negativos).

```
In [ ]: score = model.evaluate(X_train, y_train, verbose = 0)
        print('Accuracy over the training set:', round((score[1]*100), 2), '%')
```

Accuracy over the training set: 88.74 %

```
In [ ]: score = model.evaluate(X_test, y_test, verbose = 0)
        print('Accuracy over the test set:', round((score[1]*100), 2), '%')
```

Accuracy over the test set: 75.86 %

Observaciones 💡 -->

- El modelo muestra una buena precisión en el conjunto de entrenamiento y una precisión aceptable en el conjunto de prueba. La diferencia entre la precisión en los conjuntos de entrenamiento y prueba es significativa, lo que puede indicar la existencia de sobreajuste y la necesidad de ajustar el modelo para mejorar la generalización.
- Sabemos que hay un fuerte desbalance entre las clases, lo cual se refleja claramente en estos resultados. Aunque la precisión global del modelo

es relativamente alta, esto se debe en gran parte a su capacidad para identificar correctamente los casos negativos, con 726 aciertos. Sin embargo, los aciertos en los casos positivos son significativamente menores, con solo 327 casos correctamente clasificados frente a 245 errores de clasificación de positivos como negativos. Este desbalance en los datos está influyendo en la métrica de precisión, dando una impresión exagerada de su rendimiento general. Para abordar este problema y mejorar la capacidad del modelo para identificar correctamente los casos positivos, se implementarán técnicas de aumento de datos y regularización.

(3) Prueba de Clasificaciones

```
In [ ]: # Define a function to preprocess text
def preprocess_text(text, tokenizer, max_len):
    sequence = tokenizer.texts_to_sequences([text])
    padded_sequence = pad_sequences(sequence, maxlen=max_len)
    return padded_sequence
```

```
In [ ]: # Make predictions
def make_predictions(model, tweets):
    for text in tweets:
        padded_sequence = preprocess_text(text, tokenizer, max_len)
        prediction_prob = model.predict(padded_sequence)[0][0]
        prediction = (prediction_prob > 0.5).astype(int)
        print(f"Text: {text}")
        print(f"Probability: {prediction_prob:.4f}")
        print(f"Prediction: {'Disaster' if prediction == 1 else 'Non-Disaster'}
```

```
In [ ]: # Test inputs
tweets = [
    "Emergency services are on high alert due to the approaching storm",
    "Our team had a productive meeting this morning discussing new project i
    "He loves playing guitar and composing his own music",
    "Authorities are warning about severe thunderstorms in the area tonight"
    "Firefighters are working hard to contain the wildfires spreading across
]
```

```
In [ ]: make_predictions(model=model, tweets=tweets)
```

1/1  0s 37ms/step

Text: Emergency services are on high alert due to the approaching storm

Probability: 0.5126

Prediction: Disaster

1/1  0s 37ms/step

Text: Our team had a productive meeting this morning discussing new project ideas

Probability: 0.2515

Prediction: Non-Disaster

1/1  0s 36ms/step

Text: He loves playing guitar and composing his own music

Probability: 0.2515

Prediction: Non-Disaster

1/1  0s 39ms/step

Text: Authorities are warning about severe thunderstorms in the area tonight

Probability: 0.7977

Prediction: Disaster

1/1  0s 39ms/step

Text: Firefighters are working hard to contain the wildfires spreading across the region

Probability: 0.2515

Prediction: Non-Disaster

Observaciones 💡 -->

- El modelo muestra un buen desempeño en la mayoría de las predicciones, pero hay casos en los que la clasificación es errónea, especialmente para textos que claramente deberían ser clasificados como desastres. Esto se relaciona con todo lo discutido con anterioridad.

(2) Second Iteration (LSTM and Data Augmentation)

(1) Data Augmentation: Synonym Replacement and Random Swapping Techniques

```
In [ ]: import nlpaug.augmenter.char as nac # Character-level text augmentation
import nlpaug.augmenter.word as naw # Word-level text augmentation
import nlpaug.augmenter.sentence as nas # Sentence-level text augmentation
import nlpaug.flow as naf # Pipeline for combining multiple augmentations
from nlpaug.util import Action # Utilities and constants for augmentation a
```

```
In [ ]: # Upload files from the local system to the Colab environment
uploaded = files.upload()
# Iterate over the files uploaded by the user
for fn in uploaded.keys():
    # Print information about each uploaded file, including its name and size
    print('User uploaded file "{name}" with length {length} bytes'.format(
```

```

        name=fn, length=len(uploaded[fn])))
# Create the directory ~/.kaggle if it does not exist. This directory is where
!mkdir -p ~/.kaggle/
# Move the uploaded kaggle.json file into the ~/.kaggle/ directory. This file
!mv kaggle.json ~/.kaggle/
# Set the permissions of kaggle.json to be readable only by the owner. This
!chmod 600 ~/.kaggle/kaggle.json

```

No files selected.

Upload widget is only available when the cell has been

executed in the current browser session. Please rerun this cell to enable.

Saving kaggle.json to kaggle.json

User uploaded file "kaggle.json" with length 68 bytes

```

In [ ]: #!kaggle competitions download -c nlp-getting-started
#!kaggle datasets download -d rtatman/glove-global-vectors-for-word-represen

```

```

In [ ]: #!unzip /content/glove-global-vectors-for-word-representation.zip

```

```

In [ ]: # Initialize the WordEmbsAug augmenter with the specified model and parameters
aug_w2v = naw.WordEmbsAug(
    # Choose the type of word embedding model to use. Options include:
    # 'word2vec', 'glove', or 'fasttext'
    model_type='glove', # Currently set to 'glove'

    # Provide the path to the pre-trained word embeddings file.
    # The file should match the model_type selected.
    model_path='/content/glove.6B.100d.txt', # Path to the GloVe embeddings

    # Specify the action for augmentation. 'substitute' will replace words in
    # with their synonyms based on the word embeddings model.
    action="substitute" # Action to perform; 'substitute' replaces words with
)

```

```

In [ ]: from tqdm import tqdm
from sklearn.utils import shuffle

def augment_text(df, samples=600, pr=0.2):
    # Set the probability for word2vec augmentation
    aug_w2v.aug_p = pr
    new_text = []

    # Select the minority class samples (target == 1)
    df_n = df[df.target == 1].reset_index(drop=True)

    # Data augmentation loop
    for i in tqdm(np.random.randint(0, len(df_n), samples)):
        text = df_n.iloc[i]['text']
        augmented_text = aug_w2v.augment(text)
        new_text.append(augmented_text)

    # Create a DataFrame for the augmented data
    new_df = pd.DataFrame({'text': new_text, 'target': 1})

    # Combine the original and augmented data, and shuffle
    augmented_df = shuffle(pd.concat([df, new_df]).reset_index(drop=True))

```

```
return augmented_df
```

- **Referencia:** <https://neptune.ai/blog/data-augmentation-nlp>

```
In [ ]: augmented_df = augment_text(df)
```

```
100%|██████████| 600/600 [01:17<00:00, 7.78it/s]
```

```
In [ ]: # Create a figure with a single subplot
fig, ax = plt.subplots(figsize=(6, 4), dpi=100) # Use subplots() to get the
plt.tight_layout(pad=3.0) # Add padding around the plot

# Calculate target counts and percentages
target_counts = augmented_df['target'].value_counts()
total_count = target_counts.sum()
labels = [f'Non-Disaster Tweets ({target_counts[0]/total_count:.1%})',
          f'Disaster Tweets ({target_counts[1]/total_count:.1%})']

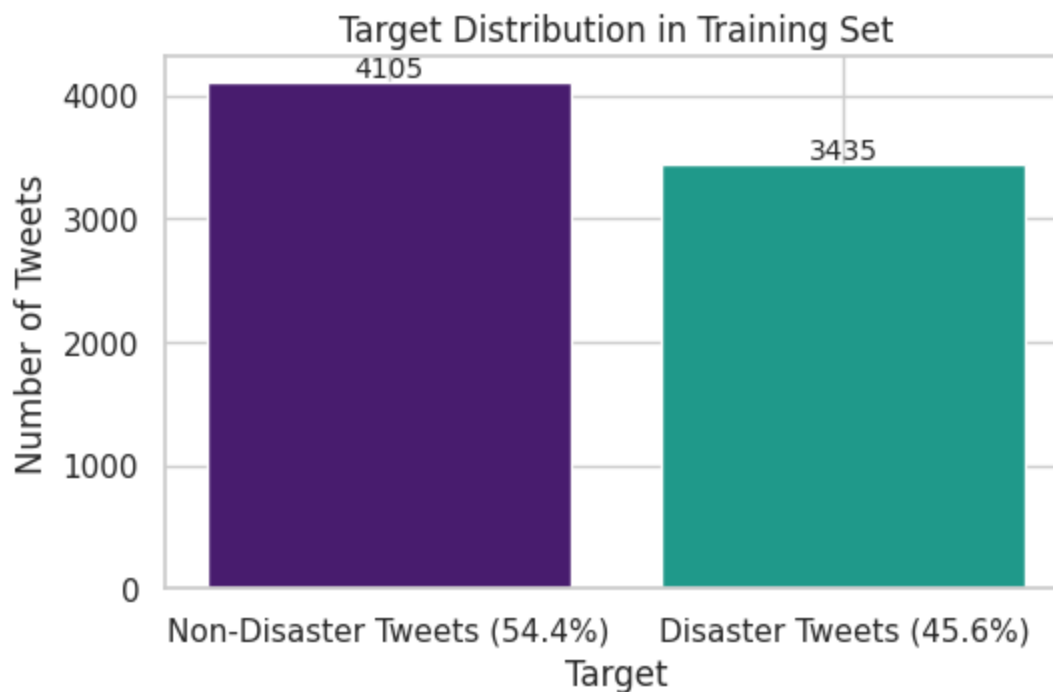
# Plot bar chart for target distribution with customized colors
bars = ax.bar(target_counts.index, target_counts, color=colors)

# Add labels above bars
for bar, label in zip(bars, target_counts):
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width() / 2, height, f'{label}',
            ha='center', va='bottom', fontsize=10)

# Set labels for x-axis and y-axis
ax.set_xticks(target_counts.index)
ax.set_xticklabels(labels)
ax.set_xlabel('Target')
ax.set_ylabel('Number of Tweets')

# Set the title
ax.set_title('Target Distribution in Training Set')

# Display the plot
plt.show()
```



(2) Embedding and Tokenization of Tweets

```
In [ ]: texts = augmented_df['text'].values
labels = augmented_df['target'].values
```

```
In [ ]: tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
X = pad_sequences(sequences, maxlen=max_len)
```

(3) Data Splitting Process

```
In [ ]: # Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2)
```

(4) Model Development and Creation

```
In [ ]: from tensorflow.keras.regularizers import l2

# Define the model
model = Sequential()

# Embedding layer
model.add(Embedding(input_dim=max_words, output_dim=32, input_length=max_len))

# Bidirectional LSTM layers with increased units
model.add(Bidirectional(LSTM(32, return_sequences=True, kernel_regularizer=l2(0.01))))
model.add(Bidirectional(LSTM(32, kernel_regularizer=l2(0.01)))) # Increased units

# Dense layers with regularization
model.add(Dense(36, activation='relu', kernel_regularizer=l2(0.01))) # Increased units
model.add(Dropout(0.5))
```

```

model.add(BatchNormalization())

model.add(Dense(18, activation='relu', kernel_regularizer=l2(0.01))) # Incr
model.add(Dropout(0.5))
model.add(BatchNormalization())

model.add(Dense(1, activation='sigmoid')) # Binary classification

# Explicitly build the model
model.build(input_shape=(None, max_len))

```

```

In [ ]: # Define the optimizer
opt = keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.9, beta_2=0.999)

```

```

In [ ]: # Compile the model
model.compile(optimizer = opt, loss = "binary_crossentropy" , metrics=["accu

```

```

In [ ]: # Model summary to check the architecture
model.summary()

```

Model: "sequential_4"

Layer (type)	Output Shape
embedding_4 (Embedding)	(None, 127, 32)
bidirectional_8 (Bidirectional)	(None, 127, 64)
bidirectional_9 (Bidirectional)	(None, 64)
dense_12 (Dense)	(None, 36)
dropout_8 (Dropout)	(None, 36)
batch_normalization_8 (BatchNormalization)	(None, 36)
dense_13 (Dense)	(None, 18)
dropout_9 (Dropout)	(None, 18)
batch_normalization_9 (BatchNormalization)	(None, 18)
dense_14 (Dense)	(None, 1)

Total params: 364,713 (1.39 MB)

Trainable params: 364,605 (1.39 MB)

Non-trainable params: 108 (432.00 B)

Observaciones 💡 -->

- Se añadió regularización L2 ($l_2 = 0.01$) a las capas Dense y LSTM para

ayudar disminuir el sobreajuste evidenciado en el modelo anterior.

- Se aumentaron la cantidad de neuronas en algunas capas para mejorar el rendimiento del modelo.

(5) Entrenamiento

```
In [ ]: # Fit the model
history = model.fit(X_train, y_train,
                    epochs = epochs, validation_data = (X_test, y_test), bat
                    callbacks=[early_stopping])
```



```
Epoch 1/15
65/65 ██████████ 38s 386ms/step - accuracy: 0.5137 - loss: 4.2520
- val_accuracy: 0.5597 - val_loss: 3.8527
Epoch 2/15
65/65 ██████████ 40s 365ms/step - accuracy: 0.5115 - loss: 3.9764
- val_accuracy: 0.5597 - val_loss: 3.6250
Epoch 3/15
65/65 ██████████ 24s 367ms/step - accuracy: 0.5107 - loss: 3.7285
- val_accuracy: 0.5597 - val_loss: 3.4170
Epoch 4/15
65/65 ██████████ 40s 354ms/step - accuracy: 0.5339 - loss: 3.4952
- val_accuracy: 0.5597 - val_loss: 3.2248
Epoch 5/15
65/65 ██████████ 24s 360ms/step - accuracy: 0.5531 - loss: 3.2847
- val_accuracy: 0.6200 - val_loss: 3.0459
Epoch 6/15
65/65 ██████████ 40s 346ms/step - accuracy: 0.5951 - loss: 3.0452
- val_accuracy: 0.6001 - val_loss: 2.8904
Epoch 7/15
65/65 ██████████ 39s 322ms/step - accuracy: 0.6407 - loss: 2.7912
- val_accuracy: 0.5869 - val_loss: 2.7427
Epoch 8/15
65/65 ██████████ 41s 316ms/step - accuracy: 0.6960 - loss: 2.5937
- val_accuracy: 0.6810 - val_loss: 2.5992
Epoch 9/15
65/65 ██████████ 41s 325ms/step - accuracy: 0.7097 - loss: 2.4589
- val_accuracy: 0.7480 - val_loss: 2.4450
Epoch 10/15
65/65 ██████████ 41s 337ms/step - accuracy: 0.7374 - loss: 2.2986
- val_accuracy: 0.7533 - val_loss: 2.3097
Epoch 11/15
65/65 ██████████ 41s 334ms/step - accuracy: 0.7556 - loss: 2.1777
- val_accuracy: 0.7527 - val_loss: 2.1682
Epoch 12/15
65/65 ██████████ 41s 331ms/step - accuracy: 0.7891 - loss: 2.0459
- val_accuracy: 0.7606 - val_loss: 2.0596
Epoch 13/15
65/65 ██████████ 22s 338ms/step - accuracy: 0.8052 - loss: 1.9389
- val_accuracy: 0.7527 - val_loss: 1.9592
Epoch 14/15
65/65 ██████████ 40s 330ms/step - accuracy: 0.8228 - loss: 1.8181
- val_accuracy: 0.7427 - val_loss: 1.8900
Epoch 15/15
65/65 ██████████ 42s 346ms/step - accuracy: 0.8257 - loss: 1.7363
- val_accuracy: 0.7447 - val_loss: 1.8097
```

Observaciones 💡 -->

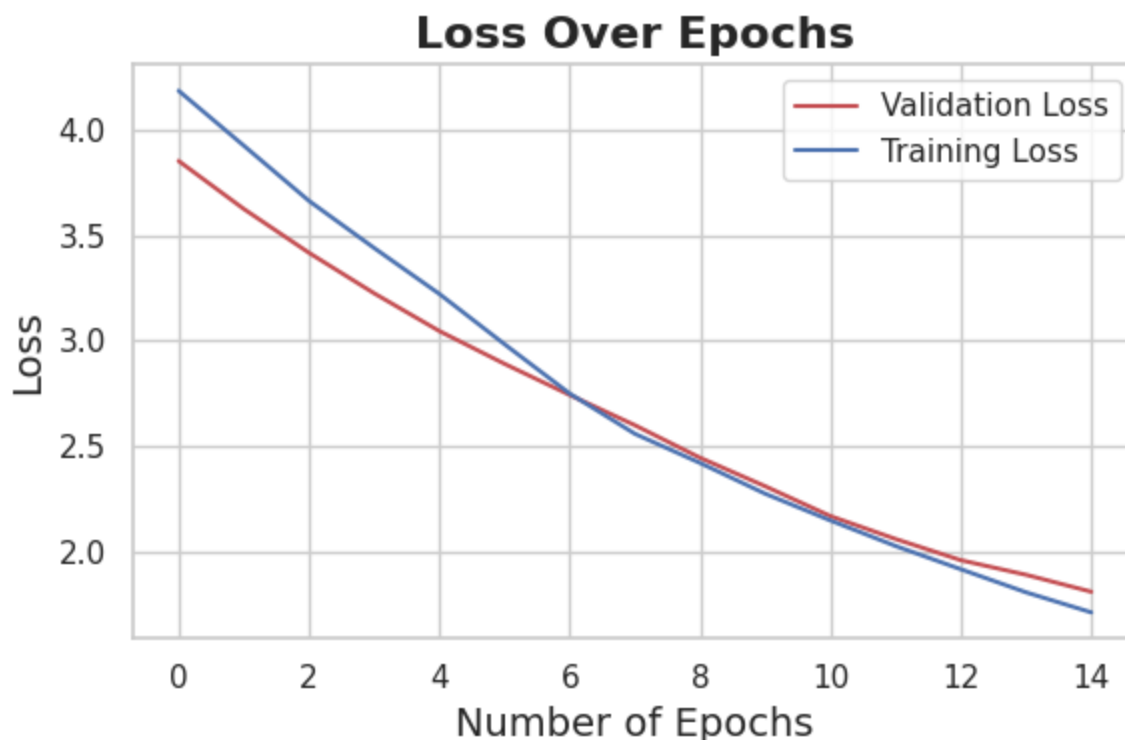
- Los resultados del entrenamiento para este modelo muestran una mejora constante en el rendimiento, tanto en el conjunto de entrenamiento como en el de validación. Al inicio del entrenamiento, la precisión es relativamente baja, con un valor de 51.37% y una pérdida considerablemente alta. Sin embargo, a medida que avanzan las épocas,

la precisión en el conjunto de entrenamiento aumenta gradualmente, alcanzando un 82.57% en la última época, mientras que la pérdida disminuye de 4.25 a 1.74.

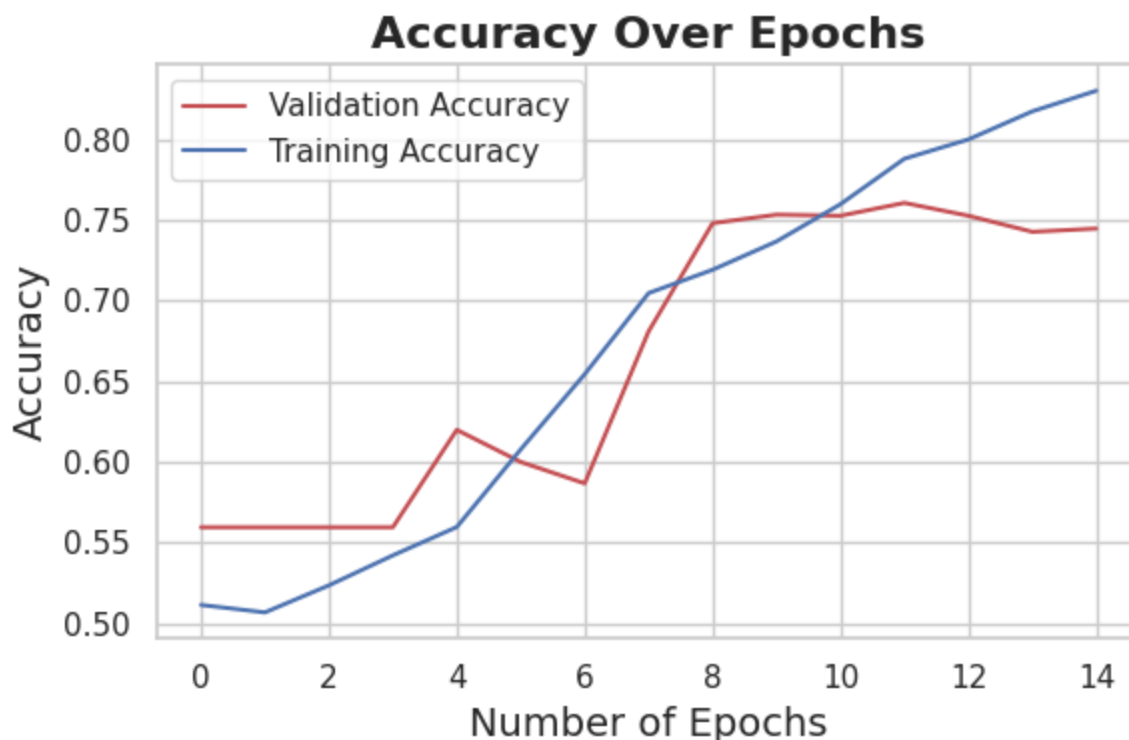
- El desempeño en el conjunto de validación también muestra una tendencia positiva. La precisión en el conjunto de validación empieza en 55.97% y mejora a 74.47% al final del entrenamiento. La pérdida en el conjunto de validación disminuye de 3.85 a 1.81 durante el mismo período.
- Este modelo está aprendiendo a generalizar mejor a partir de los datos de entrenamiento y validación. A pesar de las fluctuaciones en la precisión de validación entre algunas épocas, la tendencia general es positiva, lo que indica un buen ajuste del modelo. Sin embargo, se debe tener en cuenta que la precisión en el conjunto de validación, aunque ha mejorado, aún es menor que la precisión en el conjunto de entrenamiento, lo que sigue indicando una ligera tendencia al sobreajuste.

(6) Evaluación de Desempeño

```
In [ ]: plt.figure(figsize=(6, 4))
# Plot train and validation loss with a color palette
sns.lineplot(x=range(len(history.history['val_loss'])), y=history.history['v
sns.lineplot(x=range(len(history.history['loss'])), y=history.history['loss'
# Add titles and labels with improved styling
plt.title("Loss Over Epochs", fontsize=16, weight='bold')
plt.xlabel("Number of Epochs", fontsize=14)
plt.ylabel("Loss", fontsize=14)
# Add gridlines and legend
plt.grid(True)
plt.legend()
# Show the plot
plt.tight_layout()
plt.show()
```



```
In [ ]: plt.figure(figsize=(6, 4))
# Plot train and validation loss with a color palette
sns.lineplot(x=range(len(history.history['val_accuracy'])), y=history.history['val_accuracy'], color='red')
sns.lineplot(x=range(len(history.history['accuracy'])), y=history.history['accuracy'], color='blue')
# Add titles and labels with improved styling
plt.title("Accuracy Over Epochs", fontsize=16, weight='bold')
plt.xlabel("Number of Epochs", fontsize=14)
plt.ylabel("Accuracy", fontsize=14)
# Add gridlines and legend
plt.grid(True)
plt.legend()
# Show the plot
plt.tight_layout()
plt.show()
```



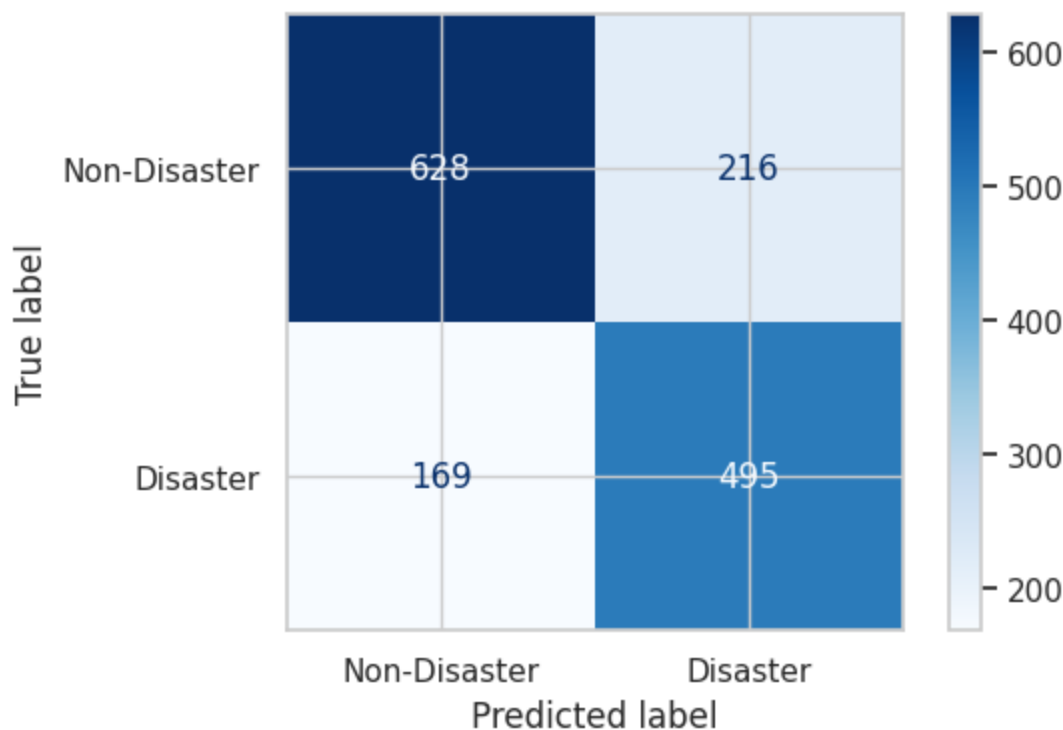
```
In [ ]: # Generate predictions
y_pred_probs = model.predict(X_test) # Replace X_test with your test data
y_pred = (y_pred_probs > 0.5).astype(int) # Binary classification

# Compute confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Visualize confusion matrix
disp = ConfusionMatrixDisplay(conf_matrix, display_labels=['Non-Disaster', 'Disaster'])
disp.plot(cmap=plt.cm.Blues, values_format='d')

# Show the plot
plt.show()
```

48/48 ————— 3s 72ms/step



- El modelo ha identificado correctamente 628 casos como negativos (tweets que no se refieren a desastres) y 495 casos como positivos (tweets que se refieren a desastres).
- No obstante, ha cometido 216 errores al clasificar casos negativos como positivos (falsos positivos) y 169 errores al clasificar casos positivos como negativos (falsos negativos).
- ES importante notar que parece ser que gracias al balanceo de clases, el modelo ahora es capaz de reconocer más tweets de desastres que antes.

```
In [ ]: score = model.evaluate(X_train, y_train, verbose = 0)
        print('Accuracy over the training set:', round((score[1]*100), 2), '%')
```

Accuracy over the training set: 91.94 %

```
In [ ]: score = model.evaluate(X_test, y_test, verbose = 0)
        print('Accuracy over the test set:', round((score[1]*100), 2), '%')
```

Accuracy over the test set: 74.47 %

Observaciones 💡 -->

En general, el nuevo modelo muestra un buen desempeño en el conjunto de entrenamiento, pero su precisión en el conjunto de prueba es más baja. Esto sugiere que, aunque el modelo está bien ajustado a los datos de entrenamiento, podría beneficiarse de técnicas adicionales para mejorar su capacidad de generalización. Sin embargo, es evidente que ha mejorado grandemente su capacidad de clasificar correctamente los

tweets de desastre, comparado con el modelo anterior.

```
In [ ]: make_predictions(model=model, tweets=tweets)
```

1/1  0s 39ms/step

Text: Emergency services are on high alert due to the approaching storm

Probability: 0.8125

Prediction: Disaster

1/1  0s 36ms/step

Text: Our team had a productive meeting this morning discussing new project ideas

Probability: 0.1439

Prediction: Non-Disaster

1/1  0s 36ms/step

Text: He loves playing guitar and composing his own music

Probability: 0.2297

Prediction: Non-Disaster

1/1  0s 37ms/step

Text: Authorities are warning about severe thunderstorms in the area tonight

Probability: 0.7734

Prediction: Disaster

1/1  0s 36ms/step

Text: Firefighters are working hard to contain the wildfires spreading across the region

Probability: 0.6358

Prediction: Disaster

Observaciones 💡 -->

- Vemos como el nuevo modelo muestra una mejora en la identificación de eventos relacionados con desastres y mantiene una buena precisión en la clasificación de textos no relacionados con desastres.
- Dado que la precisión disminuye por tan solo menos de un 1%, y ahora se clasifica más frecuentemente los tweets de desastre de manera correcta, **se cree que este es el mejor modelo de ambos creados.**

Nota: Basándose en lo anteriormente expuesto, se concluye que el mejor modelo para la clasificación de tweets de desastre es el modelo **LSTM con Regularización y Data Augmentation**.

(5) Testing Classification Capabilities with Our Custom Function

```
In [ ]: # Test inputs
tweets = [
    "Authorities are investigating a tragic mass murder that occurred in the",
    "Authorities are investigating a tragic bombing that occurred in the cit",
    "I had a great time at the concert last night with my friends",
    "Local volunteers are organizing a charity event to support victims of r",
    "Health officials are advising caution due to a recent outbreak of flu",
    "Students at the university are preparing for their upcoming exams and f",
]
```

```
In [ ]: make_predictions(model=model, tweets=tweets)
```

1/1 ————— 0s 38ms/step

Text: Authorities are investigating a tragic mass murder that occurred in the city center

Probability: 0.7014

Prediction: Disaster

1/1 ————— 0s 42ms/step

Text: Authorities are investigating a tragic bombing that occurred in the city center

Probability: 0.3910

Prediction: Non-Disaster

1/1 ————— 0s 38ms/step

Text: I had a great time at the concert last night with my friends

Probability: 0.1704

Prediction: Non-Disaster

1/1 ————— 0s 36ms/step

Text: Local volunteers are organizing a charity event to support victims of recent disasters

Probability: 0.2942

Prediction: Non-Disaster

1/1 ————— 0s 36ms/step

Text: Health officials are advising caution due to a recent outbreak of flu

Probability: 0.7593

Prediction: Disaster

1/1 ————— 0s 37ms/step

Text: Students at the university are preparing for their upcoming exams and final projects

Probability: 0.3712

Prediction: Non-Disaster

Observaciones 💡 -->

- En general, el nuevo modelo muestra un desempeño más sólido en la clasificación de eventos graves en comparación con el modelo anterior. Por ejemplo, predice correctamente eventos como "Authorities are investigating a tragic mass murder that occurred in the city center" con

una probabilidad alta de 0.7014, indicando una buena capacidad para clasificar desastres graves.

- Sin embargo, también muestra algunas debilidades; por ejemplo, clasifica incorrectamente un texto sobre una tragedia de bombardeo como "No Desastre" con una probabilidad de 0.3910, sugiriendo que el modelo puede estar subestimando la gravedad de ciertos eventos.
- Finalmente, el modelo identifica correctamente eventos no relacionados con desastres, como actividades recreativas y académicas, con probabilidades bajas, lo que refuerza su capacidad para distinguir entre texto relevante y no relevante para desastres.

Aunque se ha mejorado en términos de precisión y capacidad de identificación, aún presentan desafíos en la evaluación precisa de ciertos tweets así como un leve sesgo hacia clasificar los tweets como "Non-Disaster" como consecuencia del sobreajuste.