NLP Disaster Tweets Kaggle Mini-Project

1. Introduction

1.1 Project Goal

Twitter is a powerful mobile communication with a wide audience. This is very useful in times of emergency. However, it can be challenging for a machine learning model to classify a text message as a "disaster," as some words can be used interchangeably in a non-disaster situation. Example of a disaster tweet: "70 years since we annihilated 100000 people instantly and became aware that we have the ability to annihilate the whole of humanity." Example of a non-disaster tweet: "Be annihilated for status education mba on behalf of a on easy street careen: eOvm http://t.co/e0pl0c54FF." Both tweets use the word "annihilate," which has a different meaning in a different context. Therefore, the goal of this project is to build a Recurrent Neural Network (RNN) machine learning model to predict if a tweet is about a real disaster.

The accuracy of the model will be determine by the F1 score between the predicted and expected answers.

1.2 Natural Language Processing (NLP)

NLP is training a machine to read and understand human language. In the project, it handles the necessary steps: cleaning, breaking down the text, and converting the words into the numbers the model needs. This preparation allows the RNN model to read the context and meaning to correctly classify the tweet.

In [62]:
```
# Load the Libraries
import os
import re
import string
import nltk
import string
import unicodedata
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time
from collections import Counter
from wordcloud import WordCloud
from nltk.corpus import stopwords
nltk.download('stopwords')
from collections import Counter
```

```python
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense, Dropout, LSTM, GRU
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc, confusion_matrix, classification_report
from sklearn.utils import class_weight
from tensorflow.keras import backend as K
```

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

1.1 Data Source

- The of Source of the dataset are taken from Kaggle: https://www.kaggle.com/c/nlp-getting-started/overview
- There are 2 main data file used in this project
  - `train.csv` - the training set
  - `test.csv` - the test set

In [63]:
```python
#TRAIN_DIR = '/kaggle/input/nlp-getting-started/train.csv'
#TEST_DIR  = '/kaggle/input/nlp-getting-started/test.csv'

TRAIN_DIR = '/mnt/d/Data/nlp-getting-started/train.csv'
TEST_DIR  = '/mnt/d/Data/nlp-getting-started/test.csv'

trained_labeled_df = pd.read_csv(TRAIN_DIR)
test_df = pd.read_csv(TEST_DIR)
```

1.2 Training Dataset `train.csv`

- 7613 rows
- 5 columns:
  - `id` : unique identifier in int64 type
  - `keyword` : keyword from the tweet in object type
  - `location` : location info in object type
  - `text` : tweet text in object in object type
  - `target` : 0 = Non-Disaster, 1 = Disaster in int64 type
- There are no nulls for `id` , `text` , `target` columns
- There are nulls in `keyword` and `location` columns which might impact the cleaning process and training of model
  - keyword: 61

- location: 2533

- The `text` column consist of abbrivation, symbols like #, =>, etc, so will need to be treated during the cleaning process

In [64]:
```
trained_labeled_df.head()
```

Out[64]:

	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

In [65]:
```
print('Train DataFrame Shape:', trained_labeled_df.shape)
```

```
Train DataFrame Shape: (7613, 5)
```

In [66]:
```
print('\nSummary of Train DataFrame:\n',trained_labeled_df.info())

# Shows the total null count for every column in the DataFrame
print('\nNulls in Train DataFrame:\n',trained_labeled_df.isnull().sum())
```

```
<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

Summary of Train DataFrame:
 None

Nulls in Train DataFrame:
 id             0
keyword       61
location    2533
text           0
target         0
dtype: int64
```

1.3 Test Dataset `test.csv`

- 3263 rows

- 4 columns:
    - `id` : unique identifier in int64 type
    - `keyword` : keyword from the tweet in object type
    - `location` : location info in object type
    - `text` : tweet text in object in object type
- Similar dataframe structure to `train.csv` except `target` column

- There are nulls in `keyword` and `location` columns which might impact the cleaning process and training of model

- The `text` column consist of abbrivation, symbols like #, =>, etc, so will need to be treated during the cleaning process

In [67]:
```
test_df.head()
```

Out[67]:

	id	keyword	location	text
0	0	NaN	NaN	Just happened a terrible car crash
1	2	NaN	NaN	Heard about #earthquake is different cities, s...
2	3	NaN	NaN	there is a forest fire at spot pond, geese are...
3	9	NaN	NaN	Apocalypse lighting. #Spokane #wildfires
4	11	NaN	NaN	Typhoon Soudelor kills 28 in China and Taiwan

In [68]:
```
print('Test DataFrame shape :', test_df.shape)
```

```
Test DataFrame shape : (3263, 4)
```

In [69]:
```
print('\nSummary of Test DataFrame:\n',test_df.info())

# Shows the total null count for every column in the DataFrame
print('\nNulls in Test DataFrame:\n',test_df.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3263 entries, 0 to 3262
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   id        3263 non-null   int64 
 1   keyword   3237 non-null   object
 2   location  2158 non-null   object
 3   text      3263 non-null   object
dtypes: int64(1), object(3)
memory usage: 102.1+ KB

Summary of Test DataFrame:
 None

Nulls in Test DataFrame:
 id             0
keyword       26
location    1105
text           0
dtype: int64
```

2. Exploratory Data Analysis (EDA)

2.1 Target Distribution of the Train Dataframe

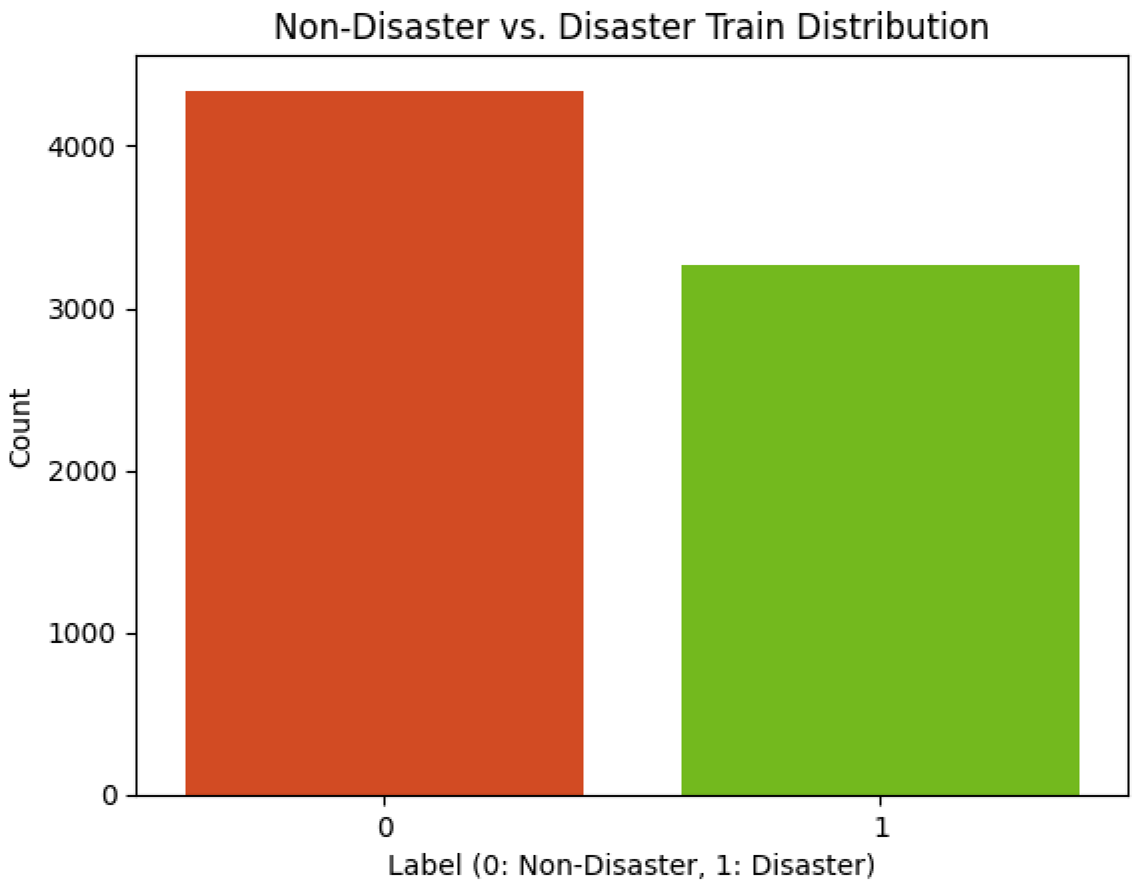
Target	%
0 = Non-Disaster	57%
1 = Disaster	42%

The slight imbalance would need to be take into consideration during the calculation for bias.

In [70]:
```
print('\nTarget distribution:\n', trained_labeled_df['target'].value_counts(normali
```

```
Target distribution:
 target
0    0.57034
1    0.42966
Name: proportion, dtype: float64
```

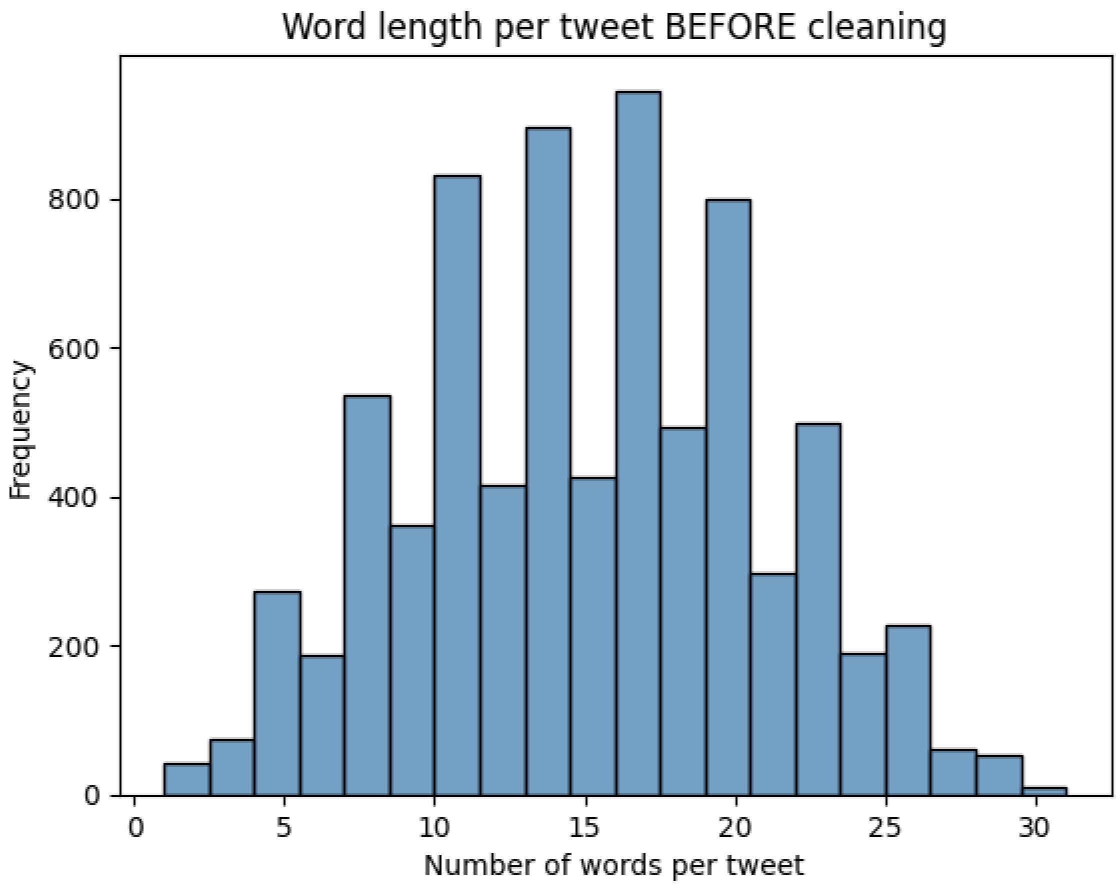
In [71]:
```
sns.countplot(
    x=trained_labeled_df['target'],
    palette=["#f33d05", "#7bd307"],
    hue=trained_labeled_df['target'],
    legend=False
    )
plt.title('Non-Disaster vs. Disaster Train Distribution', fontsize=12)
plt.xlabel('Label (0: Non-Disaster, 1: Disaster)')
plt.ylabel('Count')
plt.show()
```



2.2 Word Length Per Tweet Distribution

- Histogram of training dataset text shows a normal distribution, with most tweets having between 5 and 25 words
- The peak is at 17 words per tweet, with an occurrence of about 10,000 times
- For an RNN, every input sequence must have the same length
- The distribution will change after removing irrelevant words, and we will then decide whether padding or truncation is needed

In [72]:
```
def plot_word_length_distribution(df, text_col='text', bins=20, title='Word length
    df['len_words'] = df[text_col].astype(str).str.split().apply(len)
    sns.histplot(data=df, x='len_words', bins=bins, color='steelblue')
    plt.title(title)
    plt.xlabel('Number of words per tweet')
    plt.ylabel('Frequency')

    plt.show()

plot_word_length_distribution(trained_labeled_df, title='Word length per tweet BEFO
```



2.2 Keyword Frequencies

List and plot the top 10 keywords and their frequencies for both disaster and non-disaster tweets.

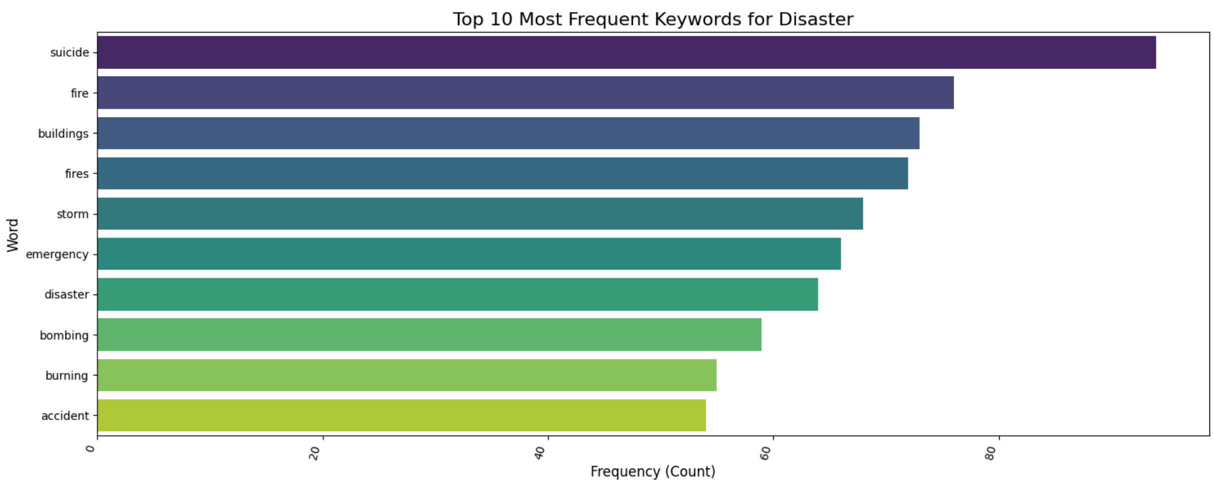
In [73]:

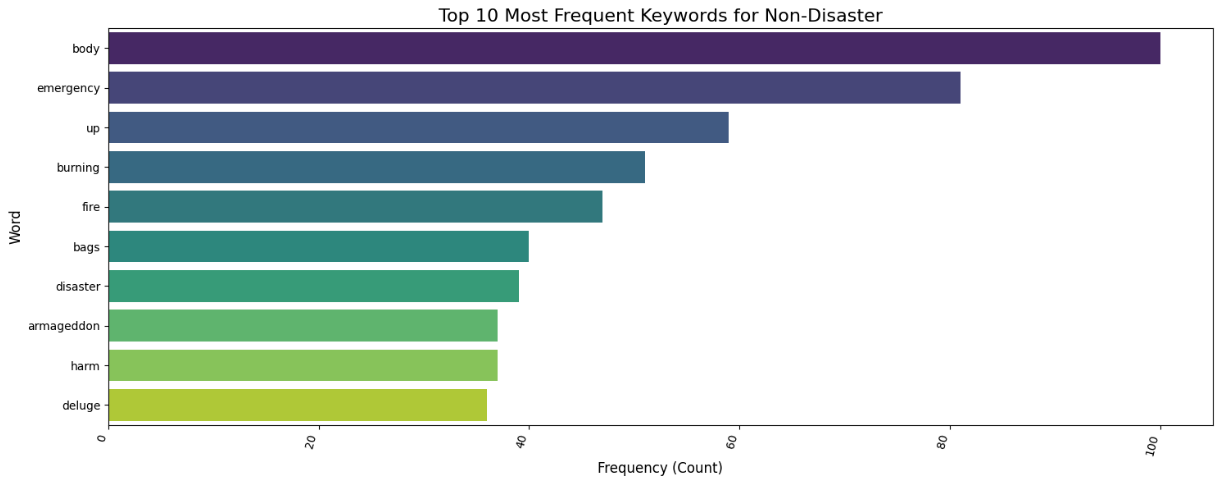
```
# Find and plot the top most frequent words
def word_count(df, text_column):
    all_words = []

    # Lowercase
    text_data = df[text_column].astype(str).str.lower()
    for text in text_data:
        # use regex to find each word
        words = re.findall(r'\b\w+\b', text)
        cleaned_words = [re.sub(r'^20', '', word) for word in words]
        all_words.extend(cleaned_words)

    # Count frequencies of each word to a dictionary
    word_counts = Counter(all_words)
    return word_counts


# print(word_counts)

def plot_top_words(word_counts, n=50, plot_size=(15, 6), title=''):

    # List top N to a new dataFrame
```

```
    top_n_words = word_counts.most_common(n)
    freq_df = pd.DataFrame(top_n_words, columns=['Word', 'Count'])

    # Plot with Barplot
    #plt.figure(figsize=(15, 6))
    plt.figure(figsize=plot_size)
    sns.barplot(
        x='Count',
        y='Word',
        data=freq_df,
        palette='viridis',
        hue='Word',
        legend=False
    )

    if not title:
        title = f'Top {n} Most Frequent Words'

    plt.title(title, fontsize=16)
    plt.xlabel('Frequency (Count)', fontsize=12)
    plt.ylabel('Word', fontsize=12)
    plt.xticks(rotation=75, ha='right', fontsize=10)
    plt.tight_layout()
    plt.show()
```

In [74]:

```
top_n = 10
plot_title = f'Top {top_n} Most Frequent Keywords for Disaster'
disaster_keyword_count_results = word_count(trained_labeled_df[trained_labeled_df.t
plot_top_words(disaster_keyword_count_results, n=top_n, title=plot_title)

plot_title = f'Top {top_n} Most Frequent Keywords for Non-Disaster'
non_disaster_keyword_count_results = word_count(trained_labeled_df[trained_labeled_
plot_top_words(non_disaster_keyword_count_results, n=top_n, title=plot_title)
```



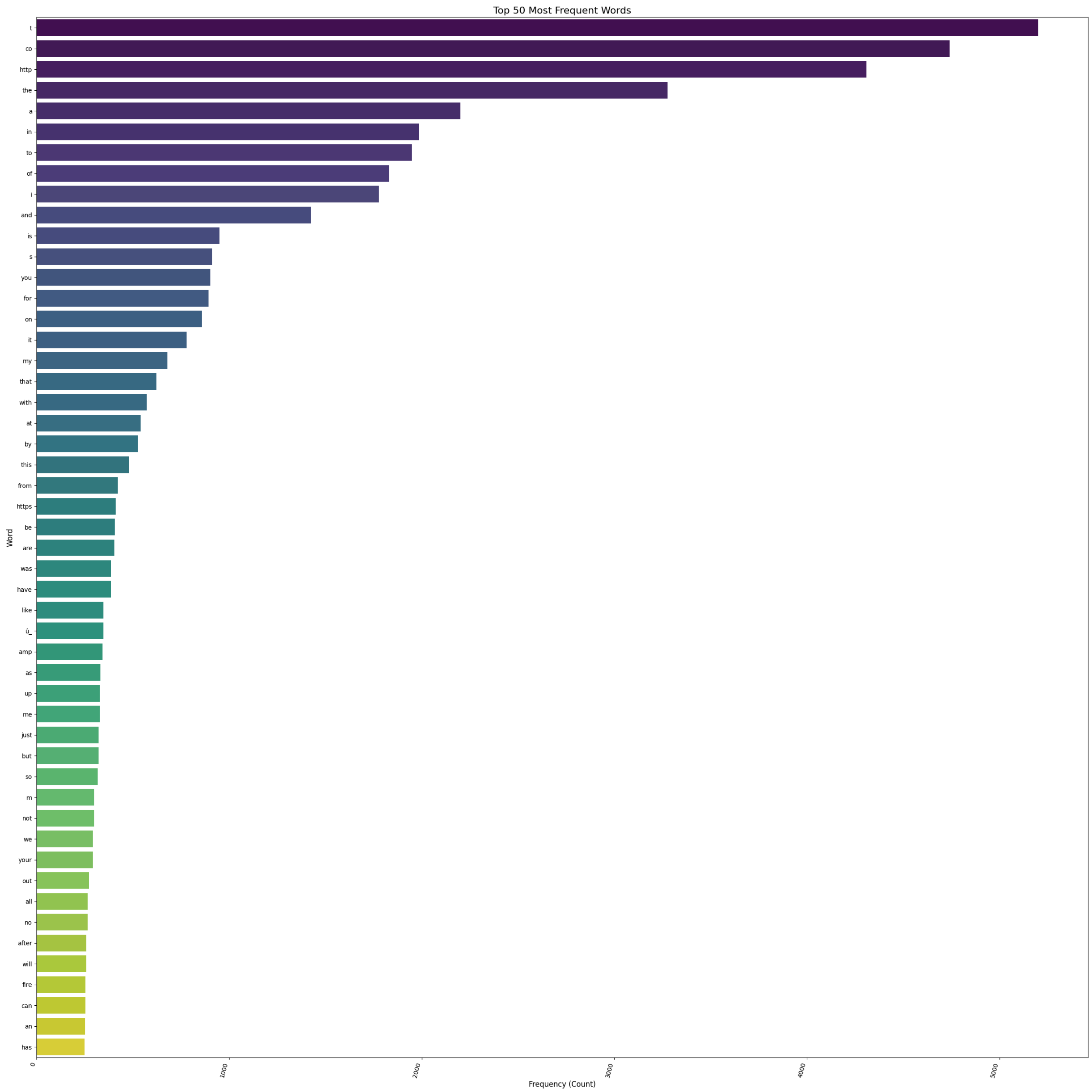


2.3 Visual Inspection of the Text

2.3.1 Word in Text Frequencies

- List and plot the top 50 words and their frequencies in tweets text.
- Usnually high amount of irrelevent words such as "t", "co", "http", etc. should be remove in the cleaning process

In [75]:
```
text_count_results = word_count(trained_labeled_df, 'text')
plot_top_words(text_count_results, n=50,  plot_size=(25, 25))
```



2.3.2 Word Cloud for Train Dataset

- A word cloud was used to visually inspect the top 100 words in the training dataset

In [76]:
```
# Word Cloud - Cleaned Text
def word_cloud_plot(df_text, plot_title):

    # Combine all tweets into one string
    combined_text = " ".join(df_text.astype(str))

    # Generate word cloud
    wc_data = WordCloud(width=800, height=400, background_color='white', max_words=

    if not plot_title:
        plot_title = f'Word Cloud'

    plt.figure(figsize=(10,5))
    plt.imshow(wc_data, interpolation='bilinear')
```

```python
    plt.axis('off')
    plt.title(plot_title)
    plt.show()
```

In [77]:
```python
plot_title = f'Word Cloud Training Dataset Text Before Cleaning'
word_cloud_plot(trained_labeled_df['text'], plot_title)
```

Word Cloud Training Dataset Text Before Cleaning

2.3.3 Word Cloud for Test Dataset

- A word cloud was also used to visually inspect the top 100 words in the test dataset.
- Based on observation, it can be assumed that similar words found in the training dataset are also in the test dataset.
- Therefore, the same cleaning logic can be applied to both datasets.

In [78]:
```python
plot_title = f'Word Cloud Test Dataset Text Before Cleaning'
word_cloud_plot(test_df['text'], plot_title)
```

Word Cloud Test Dataset Text Before Cleaning

2.4 Data Cleaning and Preprocessing

2.4.1 Data Cleaning

- Removed weird, accented characters
- Removed website links, Twitter handles, leftover HTML tags, and emojis
- Removed custom words, like "the" and "a," that were not relevant to the model's training

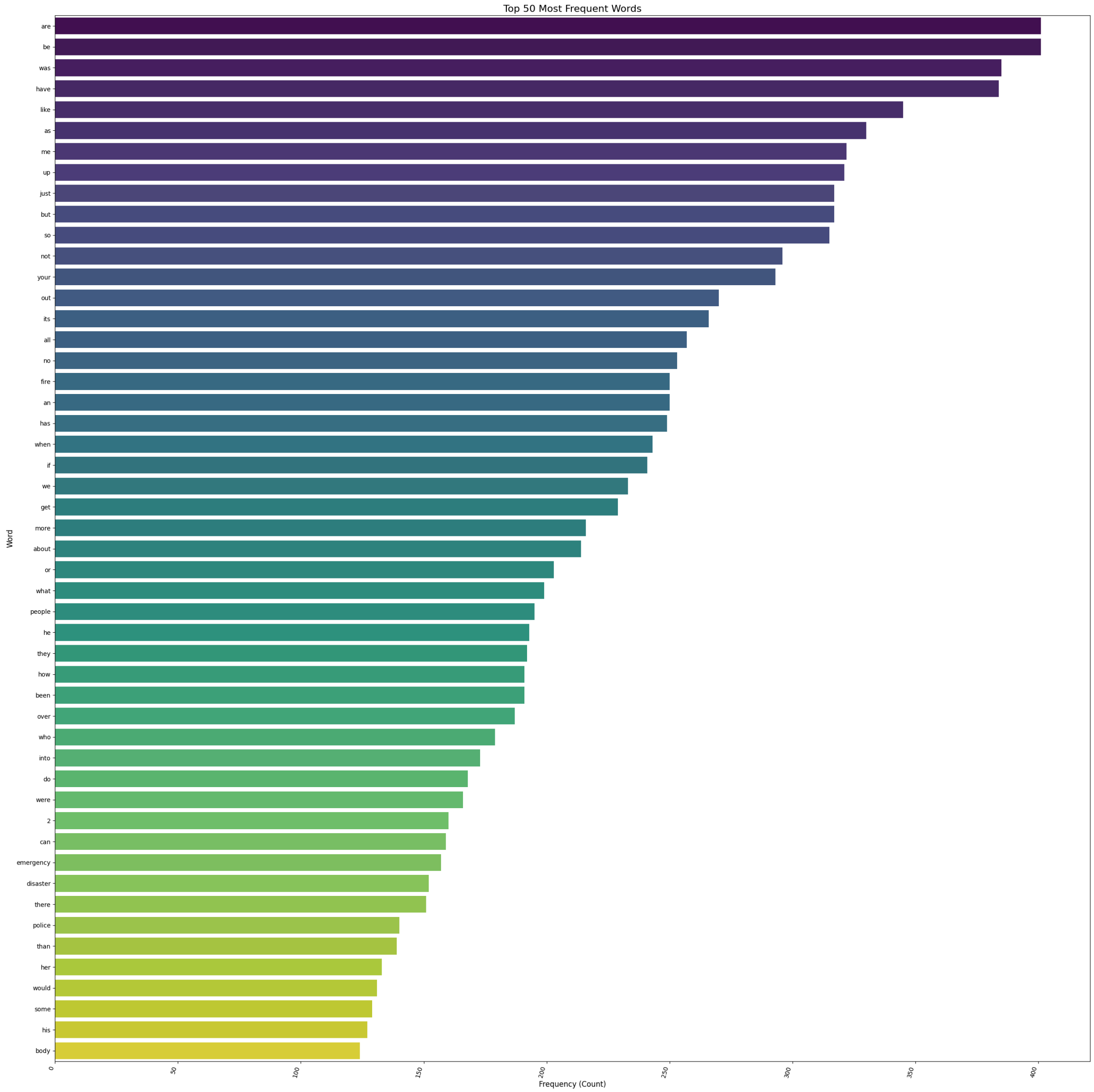
In [79]:
```
# Remove accented letters into base + accent, then drop non-ASCII
def remove_accents(text):
    text = unicodedata.normalize('NFKD', text)
    text = text.encode('ascii', 'ignore').decode('utf-8')
    return text


# Custom stopwords
CUSTOM_STOPWORDS = set([
    'the', 'in', 'a', 'of', 'to', 'on',
    'and','for','is', 'at', 'i', 'by', 'from', 'you',
    'with', 'that', 'this', 'after', 'it', 'my', 'will',
    'rt', 'via', 'amp', 'im', 'us', 'u', '…',
    'http', 'https', 'going', 't', 'CO',
    'love', 'amp', 'first', 'please', 'may', 'thats',
    'look','back','watch', 'come', 'said', 'feel',
    'think', 'video', 'even', 'say', 'way', 'day', 'see',
    'need', 'everyone', 'dont', 'new', 'lol', 'time', 'work'
    'run', 'one', 'make', 'car', 'let', 'great', 'cant', 'every',
    'got', 'still', 'now', 'never', 'summer', 'always', 'take',
    'guy', 'run', 'liked', 'little', 'things', 'latest',
    'photo', 'set', 'looks', 'theres', 'man', 'want', 'girl',
    'hes', 'families', 'know', 'news', 'today', 'road'
```

```
])

url_re = re.compile(r'https?://\S+|www\.\S+')
mention_re = re.compile(r'@\w+')
html_re = re.compile(r'<.*?>')
emoji_re = re.compile("[\U00010000-\U0010ffff]", flags=re.UNICODE)

def clean_text(s, remove_stopwords=True):
    if not isinstance(s, str):
        return ''
    s = s.lower()
    s = url_re.sub(' ', s)
    s = mention_re.sub(' ', s)
    s = html_re.sub(' ', s)
    s = emoji_re.sub(' ', s)
    s = s.replace('\n', ' ')
    s = s.translate(str.maketrans('', '', string.punctuation))
    s = re.sub('\s+', ' ', s).strip()

    if remove_stopwords:
        words = [w for w in s.split() if w not in CUSTOM_STOPWORDS]
        s = ' '.join(words)
    return s

# Clean train dataset
trained_labeled_df['text_clean'] = trained_labeled_df['text'].apply(remove_accents)
trained_labeled_df['text_clean'] = trained_labeled_df['text_clean'].apply(clean_tex

# Clean test dataset
test_df['text_clean'] = test_df['text'].apply(remove_accents)
test_df['text_clean']  = test_df['text_clean'].apply(clean_text)
```

2.4.2 Post Data Cleaning Inspection

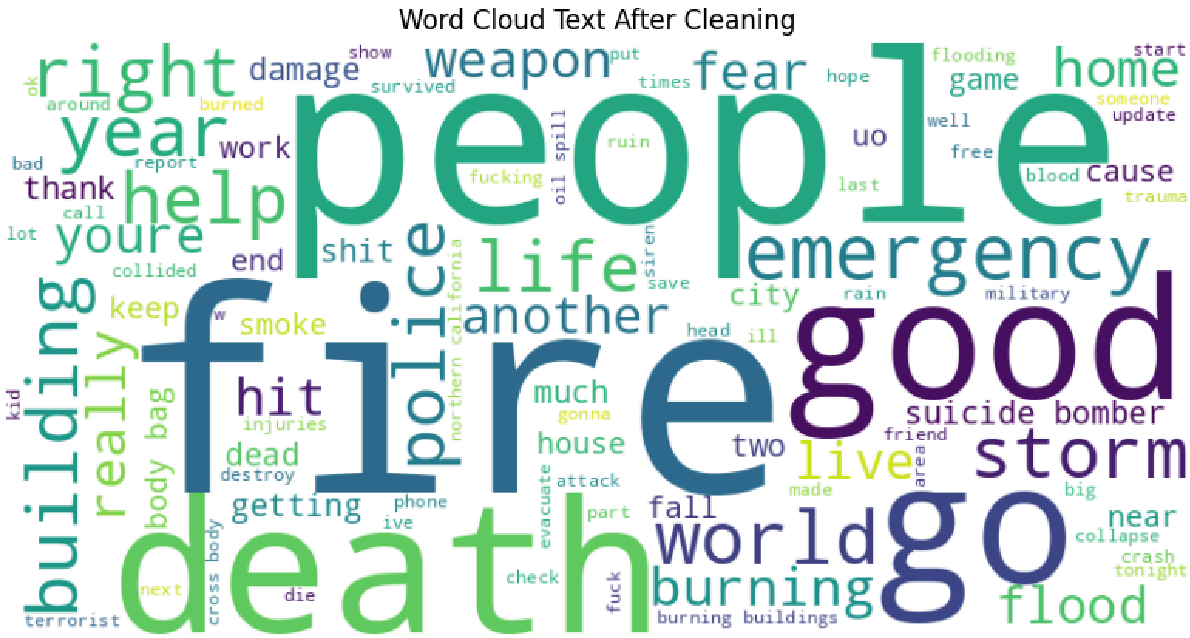
In [80]:

```
text_clean_count_results = word_count(trained_labeled_df, 'text_clean')
plot_top_words(text_clean_count_results, n=50,  plot_size=(25, 25))
```



In [81]:

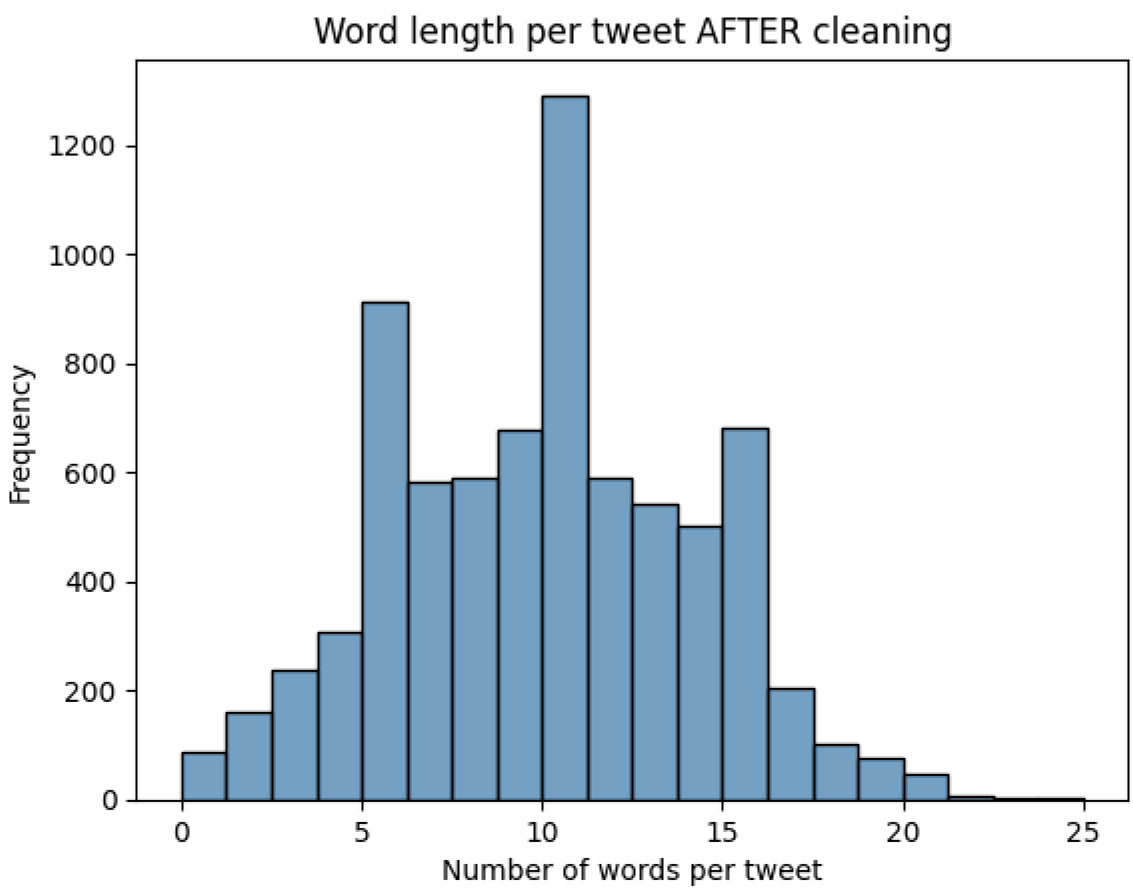
```
plot_title = f'Word Cloud Text After Cleaning'
word_cloud_plot(trained_labeled_df['text_clean'], plot_title)
```



2.4.3 Word Length Per Tweet

- After cleaning, for the training dataset, the peak is no longer at 17 words per tweet but 11 words per tweet, with an occurrence of about 13,000 times.
- However, 95% of cleaned tweets have a length of 17 words or less which mean that embedding max_len can be set to 17.

In [82]:
```
# inspect tweet lengths
plot_word_length_distribution(trained_labeled_df, text_col='text_clean', title='Wor
```



In [83]:
```
#trained_labeled_df.head()
trained_labeled_df['len_words'].quantile(0.95)
```

Out[83]: 17.0

2.4.4 Word Embedding and Analysis Plan

2.4.4.1 Word Embedding

`Word Embedding` is a process to convert the word in the text into numerical vectors. Each word is mapped to a unique vector of numbers which capture the semantic meaning of the words, i.e. words that mean the same thing, or are used in the same context, are positioned near each other in vector space. This is a critical to RNN as the model can only learn from numerical data.

2.4.4.2 Analysis Plan

Based on the EDA, the following parameters were chosen to prepare the data for the model:

- Maximum length (max_len) should be set to 17. This is because:
  - It covers 95% of the cleaned text data and will be an efficient use of data.
  - This will also reduce unnecessary padding which can be noisy for the model. In turn this can lead to faster training and better performance.
  - 5% of tweets longer than 17 words will be truncated, but this is an acceptable trade-off for the increased efficiency.

- The vocabulary size was limited to the 600 most frequent words to reduce noise and computational load. This choice was based on hyperparameter testing, which showed that larger vocabularies (e.g., 10000) did not significantly improve the model's F1 score.
- Model Selection: Three RNN architectures were chosen for testing: SimpleRNN, LSTM, and GRU. The best-performing model would then be selected for final tuning and evaluation.

2.4.5 Data Preprocessing

- The data are split into a training set (80%) and a validation set (20%) to test the model.
- Tested vocabulary sizes between 600 and 10,000 and found that a size of 600 gave the best accuracy and performance.
- Words are then tokenized to:
  - Limit the vocabulary to the 600 most common words.
  - Convert text into numerical sequences.
  - Standardize word length to 40 by either padding or truncation.
- A stop rule is used to terminate training early if the model's performance base on F1 is not improving.

In [84]:
```
# Prep the data
# feature variable: input cleaned text
X = trained_labeled_df['text_clean']
# target variable: labels to predict
y = trained_labeled_df['target']

# Split the data into 80% training set and 20% validation set.
X_train, X_val, y_train, y_val = train_test_split(X, y, stratify=y, test_size=0.2,

# Limit number of words to use in the vocabulary
max_words = 600
# Standardize word length
max_len = 17

# Tokenization
# set tokenization parameters and initialize it
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(X)

# convert the text into numbers and ensure all text per tweet are the same length
X_train_tokenized = pad_sequences(tokenizer.texts_to_sequences(X_train), maxlen=max
X_val_tokenized = pad_sequences(tokenizer.texts_to_sequences(X_val), maxlen=max_len
X_test_tokenized = pad_sequences(tokenizer.texts_to_sequences(test_df['text_clean']

# EarlyStopping stops the training early if the model val_f1_score isn't improving
# the best version was saved before it stopped
es = EarlyStopping(monitor='val_f1_calculator', patience=3, restore_best_weights=Tr
```

3. Model Architecture

3.1 Model Implementation

Comparing three RNN-family architectures, each using the same input embedding layer and a single dense output unit:

- SimpleRNN: A basic model that reads sequences step by step, carrying one hidden state in memory from the last step to the next.
- LSTM: An improved RNN that uses three gates (input, forget, output) and a cell state to decide what to remember or forget, making it better at handling long sequences.
- GRU: A simpler LSTM that uses two gates (reset and update) to control information flow, making it faster and more efficient.

Same settings across three models:

- Embedding layer: converts tokenized text into dense vectors with input_dim = max_words and output_dim = 32
- Dropout: 0.3 to reduce overfitting
- Dense output: 1 unit of sigmoid activation to predict either disaster or non-disaster
- Batch size: 64
- F1 Score: Utilizes a custom calculator for F1 score during compilation.

In [85]:

```
# Computes the F1-score
def f1_calculator(y_true, y_pred):

    y_true = tf.cast(y_true, tf.float32)

    #If the probability is 0.5 or greater, it rounds up to 1
    y_pred = tf.round(y_pred)

    # Get the True/False positive and negative
    tp = tf.reduce_sum(tf.cast(y_true * y_pred, 'float'), axis=0)
    fp = tf.reduce_sum(tf.cast((1 - y_true) * y_pred, 'float'), axis=0)
    fn = tf.reduce_sum(tf.cast(y_true * (1 - y_pred), 'float'), axis=0)

    # Precision
    # Adding K.epsilon() to prevent a "divide by zero" error
    p = tp / (tp + fp + K.epsilon())
    # Recall
    r = tp / (tp + fn + K.epsilon())

    # Calculate F1
    f1 = 2 * p * r / (p + r + K.epsilon())
    return K.mean(f1)
```

In [86]:

```
# model_rnn
model_rnn = Sequential([
    Embedding(input_dim=max_words, output_dim=32),
    SimpleRNN(32), #32
    Dropout(0.3), #0.3
    Dense(1, activation='sigmoid')
])
```

```
model_rnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'
model_rnn.summary()

history_model_rnn = model_rnn.fit(X_train_tokenized, y_train, validation_data=(X_va
#history_model_rnn = model_rnn.fit(X_train_tokenized, y_train, validation_data=(X_v
```

Model: "sequential_33"

Layer (type)	Output Shape	Param #
embedding_33 (Embedding)	?	0 (unbuilt)
simple_rnn_12 (SimpleRNN)	?	0 (unbuilt)
dropout_33 (Dropout)	?	0
dense_33 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.6348 - f1_calculator: 0.3224 -
loss: 0.6467 - val_accuracy: 0.7446 - val_f1_calculator: 0.6261 - val_loss: 0.5602
Epoch 2/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7826 - f1_calculator: 0.7105 -
loss: 0.4939 - val_accuracy: 0.7722 - val_f1_calculator: 0.6857 - val_loss: 0.4913
Epoch 3/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8146 - f1_calculator: 0.7592 -
loss: 0.4198 - val_accuracy: 0.7794 - val_f1_calculator: 0.7277 - val_loss: 0.4997
Epoch 4/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.8337 - f1_calculator: 0.7905 -
loss: 0.3845 - val_accuracy: 0.7728 - val_f1_calculator: 0.7020 - val_loss: 0.5221
Epoch 5/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.8547 - f1_calculator: 0.8178 -
loss: 0.3483 - val_accuracy: 0.7623 - val_f1_calculator: 0.7009 - val_loss: 0.5557
Epoch 6/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8721 - f1_calculator: 0.8438 -
loss: 0.3125 - val_accuracy: 0.7479 - val_f1_calculator: 0.6933 - val_loss: 0.5914
```

In [88]:

```
model_lstm = Sequential([
    Embedding(input_dim=max_words, output_dim=32),
    LSTM(32),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])

model_lstm.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
model_lstm.summary()

#history_model_lstm = model_lstm.fit(X_train_tokenized, y_train, validation_data=(X
history_model_lstm = model_lstm.fit(X_train_tokenized, y_train, validation_data=(X_
```

Model: "sequential_35"

Layer (type)	Output Shape	Param #
embedding_35 (Embedding)	?	0 (unbuilt)
lstm_4 (LSTM)	?	0 (unbuilt)
dropout_35 (Dropout)	?	0
dense_35 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - accuracy: 0.6140 - f1_calculator: 0.1996 -
loss: 0.6515 - val_accuracy: 0.7446 - val_f1_calculator: 0.6057 - val_loss: 0.5736
Epoch 2/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.7759 - f1_calculator: 0.7022 -
loss: 0.4899 - val_accuracy: 0.7741 - val_f1_calculator: 0.7259 - val_loss: 0.4854
Epoch 3/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.8043 - f1_calculator: 0.7528 -
loss: 0.4431 - val_accuracy: 0.7853 - val_f1_calculator: 0.7278 - val_loss: 0.4771
Epoch 4/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8122 - f1_calculator: 0.7645 -
loss: 0.4251 - val_accuracy: 0.7800 - val_f1_calculator: 0.7241 - val_loss: 0.4767
Epoch 5/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8143 - f1_calculator: 0.7648 -
loss: 0.4193 - val_accuracy: 0.7623 - val_f1_calculator: 0.7178 - val_loss: 0.4921
Epoch 6/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8140 - f1_calculator: 0.7689 -
loss: 0.4147 - val_accuracy: 0.7787 - val_f1_calculator: 0.7254 - val_loss: 0.4846
Epoch 7/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.8209 - f1_calculator: 0.7728 -
loss: 0.4074 - val_accuracy: 0.7781 - val_f1_calculator: 0.7174 - val_loss: 0.4819
Epoch 8/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.8251 - f1_calculator: 0.7802 -
loss: 0.4029 - val_accuracy: 0.7840 - val_f1_calculator: 0.7243 - val_loss: 0.4914
Epoch 9/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.8210 - f1_calculator: 0.7724 -
loss: 0.4060 - val_accuracy: 0.7833 - val_f1_calculator: 0.7184 - val_loss: 0.4872
Epoch 10/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8228 - f1_calculator: 0.7758 -
loss: 0.3982 - val_accuracy: 0.7840 - val_f1_calculator: 0.7061 - val_loss: 0.4920
```

In [ ]:

```
model_gru = Sequential([
    Embedding(input_dim=max_words, output_dim=32),
    GRU(32),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])

model_gru.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'
```

```
model_gru.summary()

#history_model_gru = model_gru.fit(X_train_tokenized, y_train, validation_data=(X_v
history_model_gru = model_gru.fit(X_train_tokenized, y_train, validation_data=(X_va
```

Model: "sequential_31"

Layer (type)	Output Shape	Param #
embedding_31 (Embedding)	?	0 (unbuilt)
gru_16 (GRU)	?	0 (unbuilt)
dropout_31 (Dropout)	?	0
dense_31 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - accuracy: 0.6110 - f1_calculator: 0.1923 - loss: 0.6507 - val_accuracy: 0.7380 - val_f1_calculator: 0.5940 - val_loss: 0.5668
Epoch 2/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.7793 - f1_calculator: 0.7140 - loss: 0.4854 - val_accuracy: 0.7741 - val_f1_calculator: 0.7295 - val_loss: 0.4844
Epoch 3/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8018 - f1_calculator: 0.7504 - loss: 0.4360 - val_accuracy: 0.7787 - val_f1_calculator: 0.7254 - val_loss: 0.4803
Epoch 4/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8103 - f1_calculator: 0.7617 - loss: 0.4238 - val_accuracy: 0.7761 - val_f1_calculator: 0.7221 - val_loss: 0.4833
Epoch 5/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8151 - f1_calculator: 0.7660 - loss: 0.4138 - val_accuracy: 0.7702 - val_f1_calculator: 0.7229 - val_loss: 0.4890
Epoch 6/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8154 - f1_calculator: 0.7672 - loss: 0.4091 - val_accuracy: 0.7794 - val_f1_calculator: 0.7095 - val_loss: 0.4937
Epoch 7/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8246 - f1_calculator: 0.7794 - loss: 0.4039 - val_accuracy: 0.7735 - val_f1_calculator: 0.7157 - val_loss: 0.4891
Epoch 8/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 19ms/step - accuracy: 0.8279 - f1_calculator: 0.7790 - loss: 0.3976 - val_accuracy: 0.7722 - val_f1_calculator: 0.7209 - val_loss: 0.4954
Epoch 9/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8309 - f1_calculator: 0.7830 - loss: 0.3897 - val_accuracy: 0.7761 - val_f1_calculator: 0.7162 - val_loss: 0.4959
Epoch 10/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.8317 - f1_calculator: 0.7873 - loss: 0.3867 - val_accuracy: 0.7833 - val_f1_calculator: 0.7157 - val_loss: 0.5014
```

3.2 Model Comparison Summary

Accuracy and Loss

- The SimpleRNN model performs the worst, as it starts overfitting quickly.
- The LSTM and GRU models handle the data better and do not overfit as much. Their performance is very close.

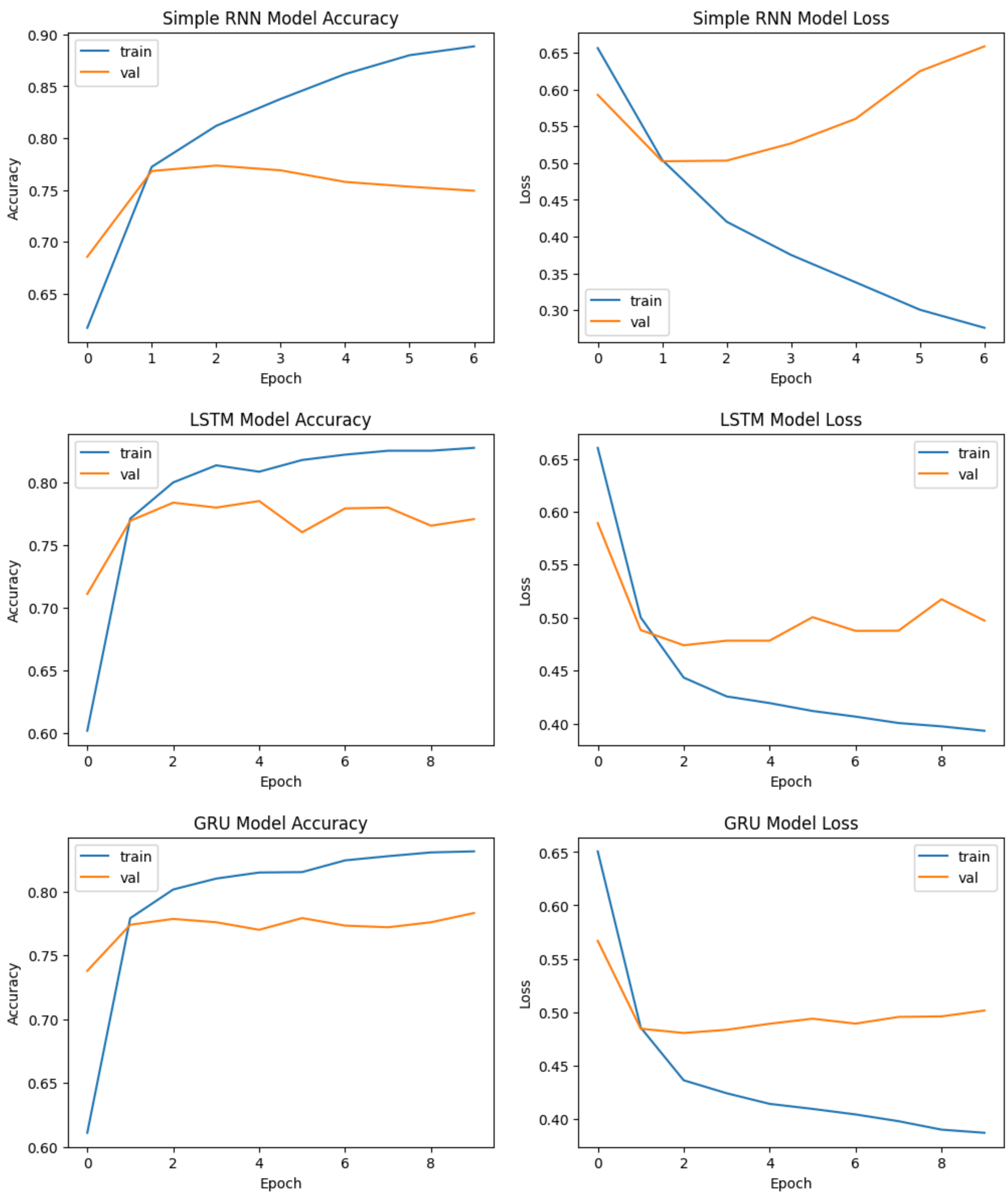
F1 Scores

Model	Validation F1 Score
SimpleRNN	0.716
LSTM	0.719
GRU	0.724

Overall

- The GRU model is the best choice for this project because it had the highest F1 score.
- It can remember important information over a long series of data points better than a SimpleRNN, similar to an LSTM.
- The GRU is more lightweight compared to an LSTM because it has fewer gates and parameters. This allows it to train faster and use less memory.

In [ ]:
```
# Plot training history
def plot_history(history, title):
    plt.figure(figsize=(12,4))
    plt.subplot(1,2,1)
    plt.plot(history.history['accuracy'], label='train')
    plt.plot(history.history['val_accuracy'], label='val')
    plt.title(f'{title} Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.subplot(1,2,2)
    plt.plot(history.history['loss'], label='train')
    plt.plot(history.history['val_loss'], label='val')
    plt.title(f'{title} Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.show()

plot_history(history_model_rnn, 'Simple RNN Model')
plot_history(history_model_lstm, 'LSTM Model')
plot_history(history_model_gru, 'GRU Model')
```



In [ ]:

```
val_preds_model_rnn_1 = (model_rnn.predict(X_val_tokenized) > 0.5).astype(int).rave
print('SimpleRNN Validation F1:', f1_score(y_val, val_preds_model_rnn_1))

val_preds_model_lstm = (model_lstm.predict(X_val_tokenized) > 0.5).astype(int).rave
print('LSTM Validation F1:', f1_score(y_val, val_preds_model_lstm))

val_preds_model_gru = (model_gru.predict(X_val_tokenized) > 0.5).astype(int).ravel(
print('GRU Validation F1:', f1_score(y_val, val_preds_model_gru))
```

```
48/48 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
SimpleRNN Validation F1: 0.715670436187399
48/48 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
LSTM Validation F1: 0.7192276749798874
48/48 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
GRU Validation F1: 0.7240802675585284
```

3.3 Hyperparameter Tuning

Since the GRU model achieved the best score among the tested architectures, hyperparameter tuning was performed on this model using the following key parameters:

- Batch sizes: [16, 32, 64]
- Dropout rates: [0.5, 0.3, 0.05]

Hyperparameter optimization results (sorted by highest validation accuracy):

Batch Size	Dropout Rate	Best Val F1 Score	Time Taken
32	0.05	0.7285	20.76
64	0.30	0.7231	8.71
64	0.05	0.7190	8.53
32	0.30	0.7189	22.81
32	0.50	0.7187	27.77
16	0.50	0.7178	20.77
64	0.50	0.7162	8.32
16	0.30	0.7145	20.34
16	0.05	0.7123	22.18

From the results, most hyperparameter combinations achieved similar validation scores around 0.72

However, the best hyperparameters for the GRU model, balancing accuracy and processing time, are:

- Batch size: 64
- Dropout rate: 0.3

Insights from Hyperparameter Tuning

- The GRU model consistently scored around 0.72 regardless of the hyperparameters.
- The top-performing model used a batch size of 32 and a dropout rate of 0.05, achieving a F1 Score of 0.7285.
- A model with a batch size of 64 and a dropout rate of 0.30 had a F1 Score of 0.7231, which is close to the top score, but it trained in about half the time. This is because a

larger batch size processes more samples at once, making each training step faster.

- The model with the worst performance had a batch size of 32 and a dropout rate of 0.30 with the longest training time at 27.77 seconds.
- Dropout rate did not significantly affect training time, but it did impact accuracy. Lower dropout rates, like 0.05, consistently produced better results than higher rates. This is because a low dropout rate ignores fewer neurons, which lets the model learn most of its connections and not oversimplify the data.

In [ ]:

```
# hyperparameters to tune
batch_sizes = [16, 32, 64]
dropout_rates = [0.5, 0.3, 0.05]

results = []

for batch in batch_sizes:
    for dropout in dropout_rates:
        print(f"\nbatch_size={batch} dropout={dropout}")

        # define model
        model_gru = Sequential([
            Embedding(input_dim=max_words, output_dim=32),
            GRU(32),
            Dropout(dropout),
            Dense(1, activation='sigmoid')
        ])

        model_gru.compile(optimizer='adam', loss='binary_crossentropy', metrics=['a

        # track time
        start_time = time.time()

        history = model_gru.fit(
            X_train_tokenized, y_train,
            validation_data=(X_val_tokenized, y_val),
            epochs=10,
            batch_size=batch,
            callbacks=[es],
            verbose=0
        )

        end_time = time.time()
        elapsed_time = end_time - start_time

        # get best F1
        best_val_f1 = max(history.history['val_f1_calculator'])

        # save results
        results.append({
            'Batch Size': batch,
            'Dropout Rate': dropout,
            'Best Val F1 Score': round(best_val_f1, 4),
            'Time Taken (s)': round(elapsed_time, 2)
        })
```

```
# tabulate
df_results = pd.DataFrame(results)
df_results = df_results.sort_values(by='Best Val F1 Score', ascending=False).reset_
df_results['Rank'] = df_results.index + 1

print("\nHyperparameter Tuning Results:")
print(df_results)
```

```
batch_size=16 dropout=0.5

batch_size=16 dropout=0.3

batch_size=16 dropout=0.05

batch_size=32 dropout=0.5

batch_size=32 dropout=0.3

batch_size=32 dropout=0.05

batch_size=64 dropout=0.5

batch_size=64 dropout=0.3

batch_size=64 dropout=0.05

Hyperparameter Tuning Results:
   Batch Size  Dropout Rate  Best Val F1 Score  Time Taken (s)  Rank
0          32          0.05             0.7285           20.76     1
1          64          0.30             0.7231            8.71     2
2          64          0.05             0.7190            8.53     3
3          32          0.30             0.7189           22.81     4
4          32          0.50             0.7187           27.77     5
5          16          0.50             0.7178           20.77     6
6          64          0.50             0.7162            8.32     7
7          16          0.30             0.7145           20.34     8
8          16          0.05             0.7123           22.18     9
```

4. Predict Test Dataset

4.1 Train with Best Hyperparameters

- Proceed to train GRU Model with the best hyperparameters combination:
  - Batch size: 64
  - Dropout rate: 0.3
- Note: This were the same hyperparameters used in the initial model comparison.
- class_weight function is used to balance the the target (42% disasters and 57% non-disaster).

4.2 Evaluation Training Process

- Training accuracy improved around 72% by epoch 4
- Validation accuracy stabilized around 75% as the model is learning without overfitting
- The F1 score on the validation set is 0.728,

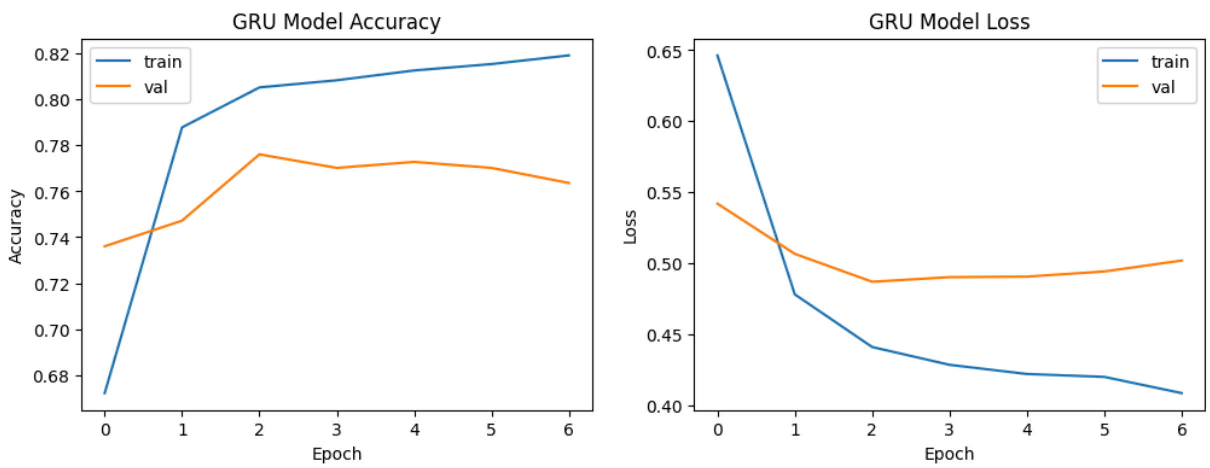
In [ ]:
```
# Compute class weights based on the training labels
class_weights_values = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train),
    y=y_train
)
class_weights_dict = dict(enumerate(class_weights_values))

# Build the GRU model
model_gru = Sequential([
    Embedding(input_dim=max_words, output_dim=32),
    GRU(32),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])

model_gru.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'
model_gru.summary()

# Train with class weights
history_model_gru = model_gru.fit(
    X_train_tokenized, y_train,
    validation_data=(X_val_tokenized, y_val),
    epochs=10,
    batch_size=64,
    class_weight=class_weights_dict,
    callbacks=[es]
)

# Plot training history
plot_history(history_model_gru, 'GRU Model')

# Validation predictions and F1 score
val_preds_model_gru = (model_gru.predict(X_val_tokenized) > 0.5).astype(int).ravel(
print('GRU Validation F1:', f1_score(y_val, val_preds_model_gru))
```

Model: "sequential_32"

Layer (type)	Output Shape	Param #
embedding_32 (Embedding)	?	0 (unbuilt)
gru_17 (GRU)	?	0 (unbuilt)
dropout_32 (Dropout)	?	0
dense_32 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)

```
Trainable params: 0 (0.00 B)
Non-trainable params: 0 (0.00 B)
Epoch 1/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6721 - f1_calculator: 0.4419 - loss: 0.6464 - val_accuracy: 0.7360 - val_f1_calculator: 0.7050 - val_loss: 0.5420
Epoch 2/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.7878 - f1_calculator: 0.7434 - loss: 0.4782 - val_accuracy: 0.7472 - val_f1_calculator: 0.7122 - val_loss: 0.5067
Epoch 3/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.8053 - f1_calculator: 0.7676 - loss: 0.4412 - val_accuracy: 0.7761 - val_f1_calculator: 0.7220 - val_loss: 0.4871
Epoch 4/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.8084 - f1_calculator: 0.7686 - loss: 0.4287 - val_accuracy: 0.7702 - val_f1_calculator: 0.7228 - val_loss: 0.4903
Epoch 5/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - accuracy: 0.8126 - f1_calculator: 0.7761 - loss: 0.4222 - val_accuracy: 0.7728 - val_f1_calculator: 0.7225 - val_loss: 0.4907
Epoch 6/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - accuracy: 0.8154 - f1_calculator: 0.7778 - loss: 0.4202 - val_accuracy: 0.7702 - val_f1_calculator: 0.7224 - val_loss: 0.4943
Epoch 7/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.8192 - f1_calculator: 0.7852 - loss: 0.4088 - val_accuracy: 0.7636 - val_f1_calculator: 0.7143 - val_loss: 0.5020
```




```
48/48 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step
GRU Validation F1: 0.7282608695652174
```

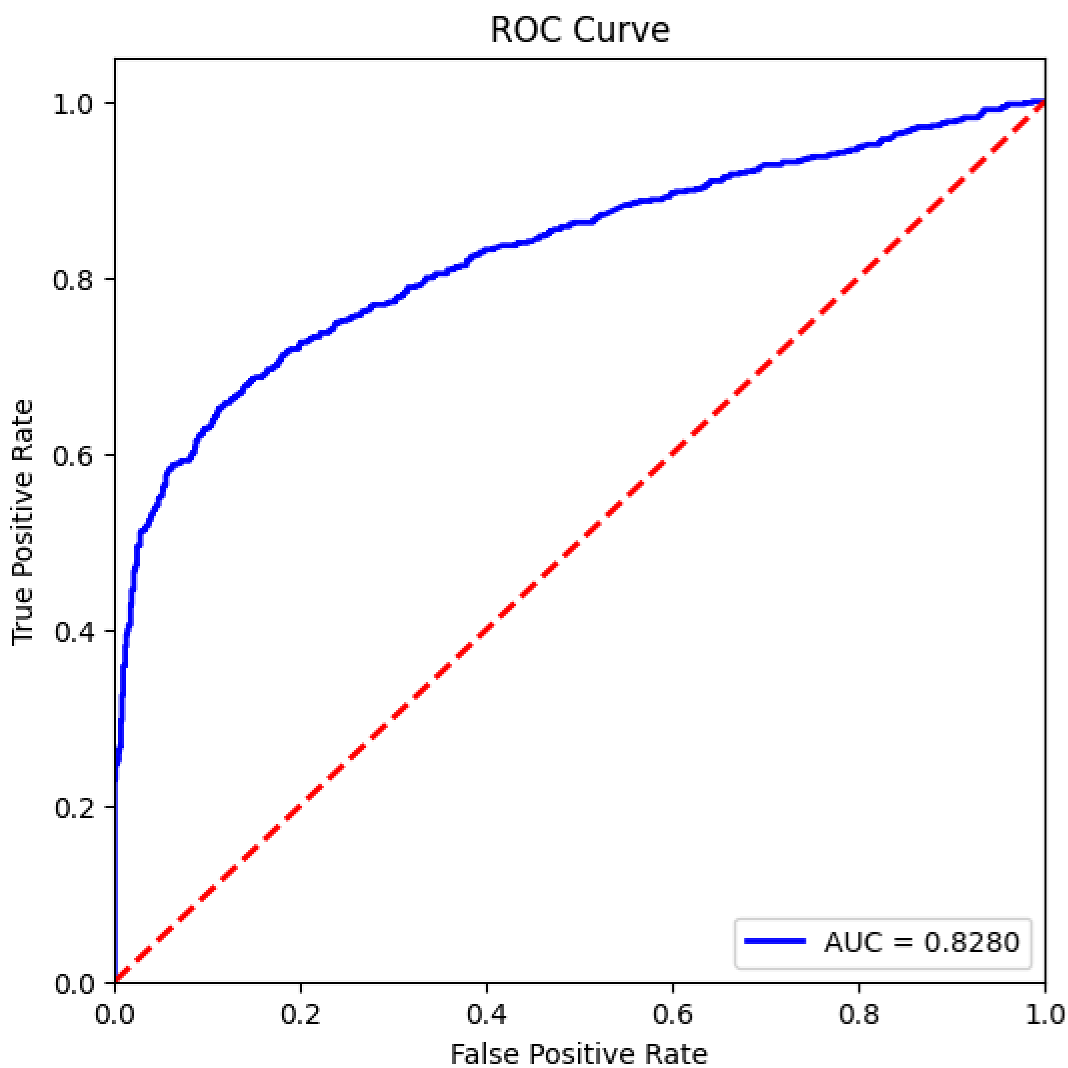
4.3 ROC Curve

The ROC curve shows GRU model has ability to distinguish between the classes. The curve is close to the top-left corner, which means the model has a high True Positive Rate and a low False Positive Rate. The Area Under the Curve (AUC) is 0.80 close to 1.

In [ ]:
```
# Get predictions as probabilities
prediction_probabilities = model_gru.predict(X_val_tokenized).ravel()

# ROC curve
false_positive_rates, true_positive_rates, thresholds = roc_curve(y_val, prediction
roc_auc = auc(false_positive_rates, true_positive_rates)
```

```
# Plot ROC
plt.figure(figsize=(6, 6))
plt.plot(false_positive_rates, true_positive_rates, color="blue", lw=2, label=f"AUC
plt.plot([0, 1], [0, 1], color="red", lw=2, linestyle="--")
plt.title("ROC Curve")
plt.legend(loc="lower right")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.show()
```

```
48/48 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
```



4.4 Predict Test Dataset

The trained GRU model was used to predict on the test dataset. The results were submitted to Kaggle and achieved a score of 0.77106.

In [ ]:

```
test_preds_rnn = (model_gru.predict(X_test_tokenized) > 0.5).astype(int).ravel()

# Save to submission.csv
```

```
submission = pd.DataFrame({
    "id": test_df['id'],
    "target": test_preds_rnn
})
submission.to_csv("submission.csv", index=False)

print("submission.csv saved with", len(submission), "entries")
print(submission.head())
```

```
102/102 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step
submission.csv saved with 3263 entries
   id  target
0   0       0
1   2       1
2   3       1
3   9       0
4  11       1
```

5. Conclusion

5.1 Model Performance Analysis

- The GRU is the best performing RNN-family model and fulfills the requirement.
- SimpleRNN works well but may underperform compared to LSTM and GRU.

5.2 Takeaways

- By adjusting the batch size and dropout rate to achieve a good balance between accuracy and training time.

- Cleaning text data is important for RNN models because it gets rid of noise. However, removing too much or too little data can hurt the model's performance.

5.3 Why Something Didn’t Work

- I spent a lot of time cleaning the text to see if removing more irrelevant words would improve the results. In fact, the F1 score got worse. So, I went back and only removed some of the stopwords. Removing too much data could lose important context or meaning to the text.

5.4 Future improvements

- To improve the model's performance in the future, I will use pre-trained GloVe embeddings with the GRU model. GloVe has already learned the meaning and relationships between millions of words from massive text sources like Wikipedia and Common Crawl. This will give the model a much better starting point than training word representations from scratch.