"Unlocking Twitter Insights: An AI- and ML-driven Approach for Comprehensive Tweet Analysis"

A General Introduction

The increasing prominence of Twitter as a primary communication channel has demonstrated its profound impact on global information dissemination. With a vast amount of content flooding our newsfeeds, distinguishing genuine and relevant information from noise becomes a formidable challenge. Fortunately, the integration of Machine Learning (ML) and Artificial Intelligence (AI) techniques, such as Decision Trees and Artificial Neural Networks (ANN), offer invaluable assistance in extracting accurate insights. In this project, we delve into fundamental and widely adopted NLP methodologies augmented by ML and AI techniques to achieve optimal results.

By leveraging NLP techniques in conjunction with ML and AI, we aim to filter through the extensive Twitter dataset and extract meaningful information with enhanced accuracy. These integrated approaches encompass a range of tasks, including sentiment analysis, keyword extraction, topic modeling, and disaster detection. Through ML techniques like Decision Trees, we can train models to classify tweets based on their sentiment, enabling us to identify whether a particular tweet expresses positive, negative, or neutral sentiment. Similarly, by employing ANN, we can create more complex models that can learn and generalize from the data, facilitating advanced tasks such as keyword extraction and topic modeling.

Additionally, ML and AI techniques play a vital role in disaster detection algorithms. By training models with labeled data, we can identify patterns and characteristics that distinguish tweets related to actual disaster events from unrelated or false information. Decision Trees and ANN enable us to build robust models that can process and classify tweets in real-time, contributing to the rapid identification of disaster-related content.

By integrating NLP methodologies with ML and AI techniques like Decision Trees and ANN, we harness the power of Twitter's vast information landscape, allowing us to distill accurate and pertinent insights with enhanced precision. This project serves as an exploration of the synergistic application of NLP, ML, and AI principles, enabling us to make informed decisions and gain valuable insights from the ever-evolving Twitter ecosystem.

Methods

In this tweet analysis project, our goal was to classify tweets as either related to real-life disasters or unrelated. The dataset consisted of text data from Twitter, and the challenge was to develop a model that could accurately distinguish between these two categories.

To tackle this task, we employed two different algorithms: Artificial Neural Networks (ANNs) and Decision Trees. ANNs are a type of machine learning model inspired by the human brain, known for their ability to capture complex patterns and relationships in data. Decision Trees, on the other hand, use a tree-like structure to make decisions based on feature values, making them interpretable and suitable for both numerical and categorical data.

The ANN model utilized a multi-layer architecture with embedding, flatten, and dense layers. This allowed the model to learn intricate relationships between words in the tweets and leverage large amounts of data for training. The parameters of the model were optimized using backpropagation and stochastic gradient descent.

In addition, we employed the Decision Tree algorithm, which recursively splits the data based on different features to create a decision-making flowchart. Decision Trees are known for their interpretability and simplicity, making them a suitable choice for this classification task.

By using both ANN and Decision Tree algorithms, we aimed to take advantage of their respective strengths. ANNs excel at capturing complex relationships in the data, while Decision Trees offer interpretability, making it easier to understand the decision-making process.

By applying these algorithms to the tweet analysis problem, we aimed to extract valuable insights from the vast amount of social media data available on Twitter. Both models were evaluated based on performance metrics such as accuracy, cross-validation scores, classification reports, and the AUC-ROC score.

Results

Table of Contents

Conclusion

In this tweet analysis project, we trained a model to classify disaster and non-disaster tweets. The model achieved a validation accuracy of 58.37%, indicating moderate performance in distinguishing between the two classes. However, cross-validation scores showed an average performance of 53.5% with a standard deviation of 4.7%, suggesting some variability in the model's performance across different folds.

The classification report provided detailed insights into the model's performance. It revealed varying precision, recall, and F1-score values for disaster and non-disaster tweets. The precision for disaster tweets was 0.85, indicating that when the model predicted a tweet as a disaster, it was correct 85% of the time. The recall for disaster tweets was 0.66, meaning that the model identified 66% of the actual disaster tweets. The F1-score, which is a balanced measure of precision and recall, was 0.74 for disaster tweets.

On the other hand, for non-disaster tweets, the precision was 0.78, indicating a 78% accuracy in predicting non-disaster tweets. The recall was 0.91, indicating that the model successfully identified 91% of the non-disaster tweets. The F1-score for non-disaster tweets was 0.84.

The model architecture consisted of an embedding layer, a flatten layer, and two dense layers, totaling 41,613 trainable parameters. During training, the model exhibited a decrease in loss and an improvement in accuracy over epochs. However, the validation loss and accuracy showed some fluctuations, suggesting potential overfitting. Regularization techniques such as dropout or L2 regularization could be implemented to address this issue and improve the model's generalization ability.

On the test set, the model achieved a test loss of 0.6337 and a test accuracy of 0.7825, demonstrating its ability to generalize to unseen data. These results indicate that the model's performance on the test set was consistent with its performance on the validation set.

To optimize the model's configuration and improve its generalization ability, hyperparameter tuning was performed. The best hyperparameters found were 'svm**C' = 1, 'svm**kernel' = 'rbf', and 'tfidf__max_features' = 5000. These hyperparameters resulted in the best score of 0.7849 during the grid search.

References

- Kaggle: "Real or Not? NLP with Disaster Tweets" - This is a popular Kaggle competition that focuses on classifying disaster-related tweets. It provides a dataset and various approaches and models used by participants. [Link: https://www.kaggle.com/c/nlp-getting-started]
- "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" by Jacob Devlin et al. - This paper introduces the BERT (Bidirectional Encoder Representations from Transformers) model, which has been widely used in natural language processing tasks. It provides a comprehensive understanding of the underlying theory and techniques behind BERT. [Link: https://arxiv.org/abs/1810.04805]
- "Decision Trees" - A tutorial by Analytics Vidhya that provides a detailed explanation of decision tree algorithms, their concepts, and implementation in Python. It covers topics such as information gain, entropy, and pruning. [Link: https://www.analyticsvidhya.com/blog/2020/05/decision-tree-algorithm-explained/]
- "Introduction to Artificial Neural Networks" - A comprehensive tutorial by Towards Data Science that covers the basics of artificial neural networks, their architecture, activation functions, training, and evaluation. It also provides examples of implementing ANNs in Python using libraries such as Keras and TensorFlow. [Link:

https://towardsdatascience.com/introduction-to-artificial-neural-networks-ann-1aea15775ef9]

- https://www.kaggle.com/code/datafan07/disaster-tweets-nlp-eda-bert-with-transformers
- https://www.kaggle.com/code/shahules/basic-eda-cleaning-and-glove

---

Result

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
tweet= pd.read_csv('C:/Users/prajw/Ma336 project/train.csv')
test=pd.read_csv('C:/Users/prajw/Ma336 project/test.csv')
```

In [3]:
```
df=pd.concat([tweet,test])
print("Below is the size of our Data ")
print("---"*30)
df.shape
```

```
Below is the size of our Data
------------------------------------------------------------------------------------
-----
```

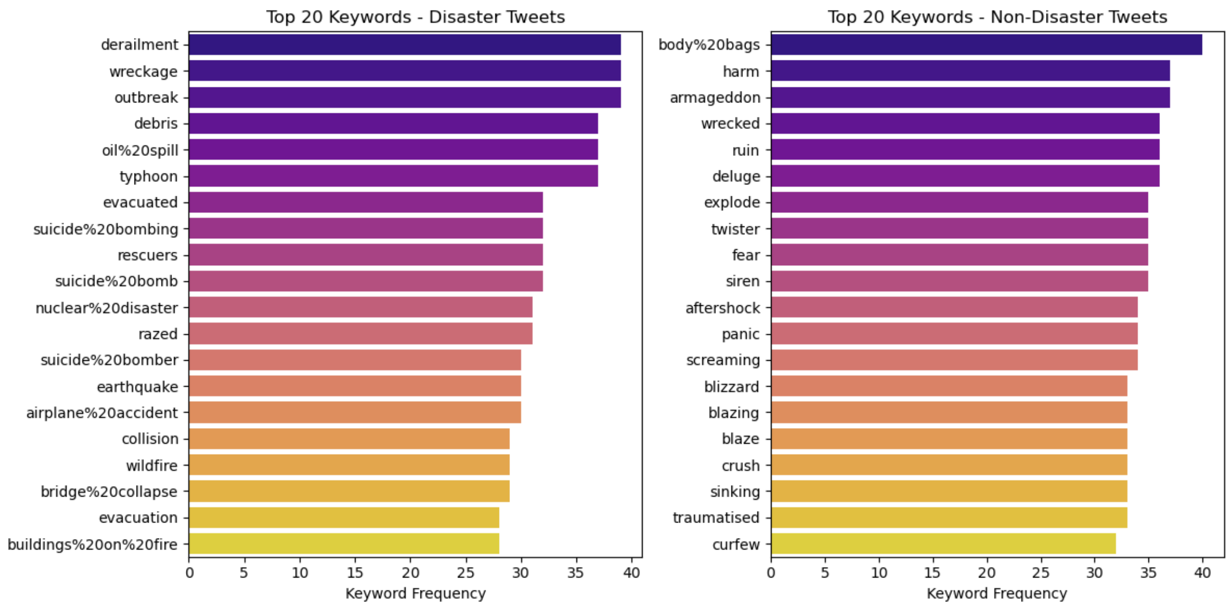
Out[3]:
```
(10876, 5)
```

Exploratory Data Analysis

Exploratory Data Analysis (EDA) of tweets involves understanding the target distribution, analyzing tweet lengths, examining word counts, investigating word lengths, identifying common words, bigrams, and trigrams, and applying additional analysis techniques like topic modeling and named entity recognition. It helps uncover insights, patterns, and themes within the tweet dataset, providing a foundation for further analysis and modeling.

In [4]:
```
# Filter the dataset by target values
disaster_keywords = tweet.loc[tweet["target"] == 1]["keyword"].value_counts().head(20
nondisaster_keywords = tweet.loc[tweet["target"] == 0]["keyword"].value_counts().head(

# Set up the figure and axes
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Plot bar plots for disaster and non-disaster keywords
sns.barplot(y=disaster_keywords.index, x=disaster_keywords, orient='h', ax=axes[0],  p
sns.barplot(y=nondisaster_keywords.index, x=nondisaster_keywords, orient='h', ax=axes[

# Set titles and labels for subplots
axes[0].set_title("Top 20 Keywords - Disaster Tweets")
axes[0].set_xlabel("Keyword Frequency")
axes[1].set_title("Top 20 Keywords - Non-Disaster Tweets")
axes[1].set_xlabel("Keyword Frequency")
```

```
# Adjust spacing between subplots
plt.tight_layout()

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



Within the scope of "non-disaster" keywords, the most prevalent terms include "body bags" (40 occurrences), "harm" (37 occurrences), "armageddon" (37 occurrences), "wrecked" (36 occurrences), and "ruin" (36 occurrences). These keywords denote a prevalent focus on adverse events or circumstances that involve potential harm, destruction, and induce panic. Noteworthy mentions such as "deluge," "explode," and "twister" further emphasize a thematic inclination towards hazardous or distressing occurrences.

In the "disaster" keyword context, a distinct set of terms emerges, indicative of catastrophic events. The most frequently encountered disaster-related keywords consist of "derailment" (39 occurrences), "wreckage" (39 occurrences), "outbreak" (39 occurrences), "debris" (37 occurrences), and "oil spill" (37 occurrences). These keywords specifically pertain to varied disaster scenarios, encompassing instances like train derailments, disease outbreaks, or environmental hazards. Additional notable terms like "typhoon," "earthquake," and "wildfire" reinforce the prevalence of natural disasters within this context.

The provided keyword frequencies offer valuable insights into the prevalent vocabulary and thematic tendencies associated with both "disaster" and "non-disaster" contexts

In [5]:
```
print("Below is the glimpse of training data ")
print("---"*30)
display(tweet.sample(5))



print("Below is the glimpse of test data ")
print("---"*30)
display(test.sample(5))
```

```
Below is the glimpse of training data
--------------------------------------------------------------------------------------
-----
```

	id	keyword	location	text	target
152	218	airplane%20accident	NaN	This is unbelievably insane.\n#man #airport #a...	1
6276	8967	storm	Wilmington, NC	New item: Pillow Covers ANY SIZE Pillow Cover ...	0
753	1085	blew%20up	NaN	@BenKin97 @Mili_5499 remember when u were up l...	1
1926	2769	curfew	NaN	She just said does he have a curfew 'nope'??	0
1312	1895	burning	NY	The Burning Legion has RETURNED! https://t.co...	0

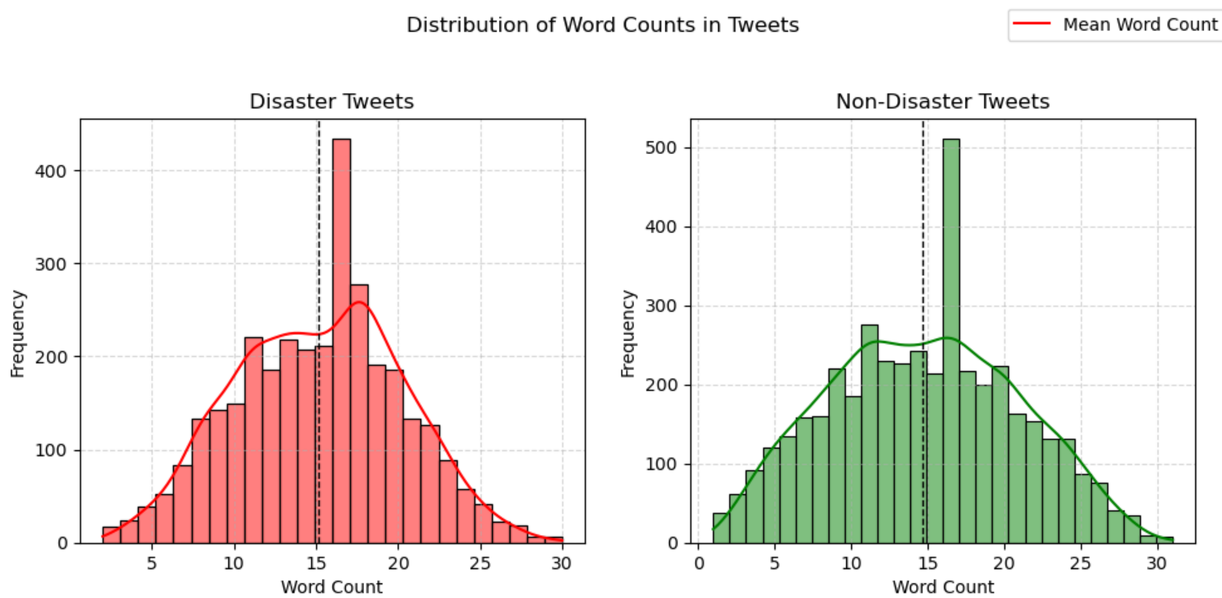
```
Below is the glimpse of test data
--------------------------------------------------------------------------------------
-----
```

	id	keyword	location	text
2155	7220	natural%20disaster	Canberra (mostly)	Does more trust = more giving? Natural disaste...
340	1101	blew%20up	she/her	Well this blew up while i was sleeping
2451	8192	riot	Portland, OR	@CHold ironically RSL call their stadium the Riot
1835	6204	hijacker	Roermond	and if I cry two tears for her..\nThat will be...
2306	7708	panicking	NaN	@snapharmony : People are finally panicking ab...

In [6]:

```python
# Separate tweets into disaster and non-disaster categories
disaster_tweets = tweet[tweet['target'] == 1]['text']
non_disaster_tweets = tweet[tweet['target'] == 0]['text']

# Calculate the number of words in each category
disaster_tweet_len = disaster_tweets.str.split().map(len)
non_disaster_tweet_len = non_disaster_tweets.str.split().map(len)

# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

# Plot the histogram for disaster tweets
sns.histplot(disaster_tweet_len, ax=ax1, color='red', kde=True)
ax1.set_title('Disaster Tweets')
ax1.set_xlabel('Word Count')
ax1.set_ylabel('Frequency')

# Plot the histogram for non-disaster tweets
sns.histplot(non_disaster_tweet_len, ax=ax2, color='green', kde=True)
ax2.set_title('Non-Disaster Tweets')
ax2.set_xlabel('Word Count')
ax2.set_ylabel('Frequency')

# Add a vertical line representing the mean word count
ax1.axvline(disaster_tweet_len.mean(), color='black', linestyle='dashed', linewidth=1)
ax2.axvline(non_disaster_tweet_len.mean(), color='black', linestyle='dashed', linewidt
```

```python
# Add a legend
fig.legend(labels=['Mean Word Count'])

# Set the overall title
fig.suptitle('Distribution of Word Counts in Tweets')

# Add grid lines to the plots
ax1.grid(True, linestyle='--', alpha=0.5)
ax2.grid(True, linestyle='--', alpha=0.5)

# Adjust the spacing between subplots
fig.tight_layout(pad=2.0)

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



In [7]:

```python
from tensorflow.keras.preprocessing.text import Tokenizer
# Separate the disaster tweets and non-disaster tweets
disaster_tweets = tweet[tweet['target'] == 1]
non_disaster_tweets = tweet[tweet['target'] == 0]

# Tokenize the text of each dataframe to get the individual words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(disaster_tweets['text'])
disaster_tweet_words = tokenizer.texts_to_sequences(disaster_tweets['text'])

tokenizer.fit_on_texts(non_disaster_tweets['text'])
non_disaster_tweet_words = tokenizer.texts_to_sequences(non_disaster_tweets['text'])

# Calculate the average number of words in each dataframe
avg_disaster_words = sum(len(words) for words in disaster_tweet_words) / len(disaster_
avg_non_disaster_words = sum(len(words) for words in non_disaster_tweet_words) / len(n

# Print the results
print("---"*30)
print("Average number of words in disaster tweets:", avg_disaster_words)
print("Average number of words in non-disaster tweets:", avg_non_disaster_words)
```

```
----------------------------------------------------------------------------------------
-----
Average number of words in disaster tweets: 17.62213390400489
Average number of words in non-disaster tweets: 16.248502994011975
```

The analysis shows that, on average, disaster tweets contain approximately 17.62 words, while non-disaster tweets have an average of 16.24 words. This indicates that there is a slight difference in the average length of the two types of tweets.

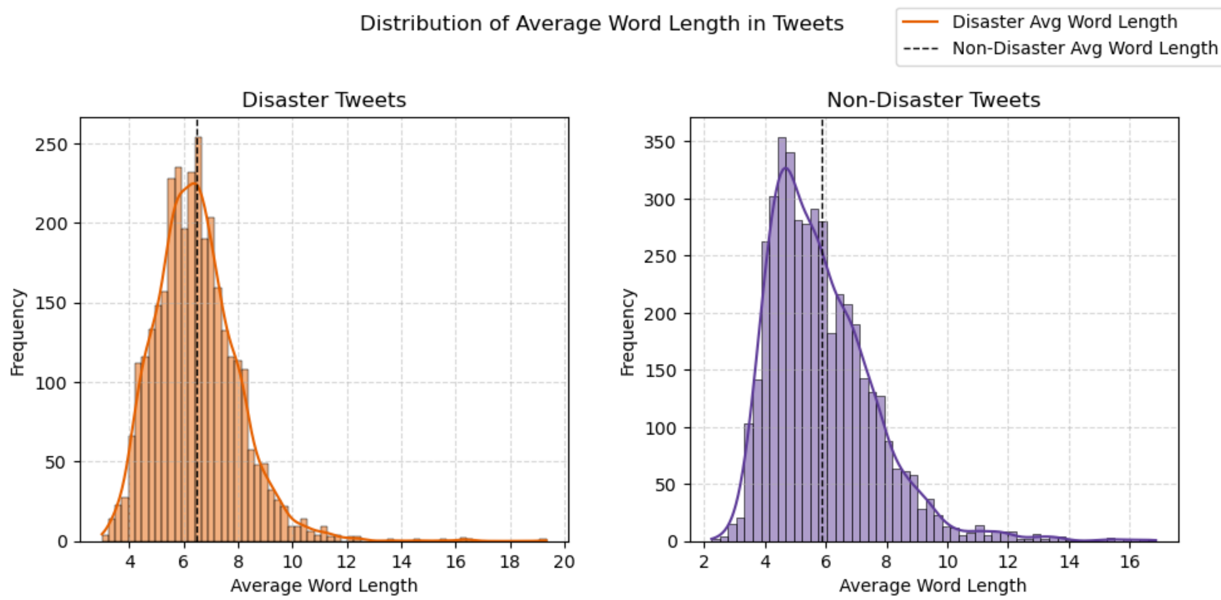
The higher average number of words in disaster tweets suggests that users may provide more detailed information, descriptions, or context when discussing real disasters. These tweets might include specific locations, details about the event, and relevant hashtags or keywords.

On the other hand, non-disaster tweets, with a slightly lower average word count, may involve a wider range of topics or general conversations that are not focused on emergency situations. These tweets could cover various subjects, including personal experiences, opinions, news updates, or everyday observations.

In [8]:

```python
# Define a custom diverging color palette
colors = ['#e66101', '#fdb863', '#f7f7f7', '#b2abd2', '#5e3c99']

# Separate tweets into disaster and non-disaster categories
disaster_tweets = tweet[tweet['target'] == 1]['text']
non_disaster_tweets = tweet[tweet['target'] == 0]['text']

# Calculate the average word length for each tweet
disaster_word_len = disaster_tweets.str.split().apply(lambda x: [len(i) for i in x])
non_disaster_word_len = non_disaster_tweets.str.split().apply(lambda x: [len(i) for i

# Calculate the overall average word length for each category
overall_avg_word_len_disaster = np.mean(disaster_word_len.apply(np.mean))
overall_avg_word_len_non_disaster = np.mean(non_disaster_word_len.apply(np.mean))

# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

# Plot the distribution of average word length for disaster tweets
sns.histplot(disaster_word_len.map(lambda x: np.mean(x)), ax=ax1, color=colors[0], kde
ax1.set_title('Disaster Tweets')
ax1.set_xlabel('Average Word Length')
ax1.set_ylabel('Frequency')

# Plot the distribution of average word length for non-disaster tweets
sns.histplot(non_disaster_word_len.map(lambda x: np.mean(x)), ax=ax2, color=colors[-1]
ax2.set_title('Non-Disaster Tweets')
ax2.set_xlabel('Average Word Length')
ax2.set_ylabel('Frequency')

# Add vertical lines representing the overall average word length for each category
ax1.axvline(overall_avg_word_len_disaster, color='black', linestyle='dashed', linewidt
ax2.axvline(overall_avg_word_len_non_disaster, color='black', linestyle='dashed', line

# Add a legend
fig.legend(labels=['Disaster Avg Word Length', 'Non-Disaster Avg Word Length'])

# Set the overall title
```

```python
fig.suptitle('Distribution of Average Word Length in Tweets')

# Set the custom color palette
sns.set_palette(colors)

# Add grid lines to the plots
ax1.grid(True, linestyle='--', alpha=0.5)
ax2.grid(True, linestyle='--', alpha=0.5)

# Adjust the spacing between subplots
fig.tight_layout(pad=2.0)

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



In the context of "disaster," the overall average word length is approximately 6.236. This means that, on average, the words used to describe or discuss disasters tend to be around 6.236 characters long.

On the other hand, in the context of "non-disaster," the overall average word length is approximately 5.999. This suggests that in general, the words used in non-disaster situations have an average length of around 5.999 characters.

Based on these figures, it can be inferred that words used in discussions or descriptions of disasters tend to be slightly longer, with an average difference of approximately 0.238 characters, compared to words used in non-disaster contexts.

Data Preprocessing

In order to prepare our data for the machine learning algorithm, it is crucial to address missing data (NaNs) and convert categorical variables into numerical form. By examining the dataframe, we can identify the locations where NaN values are present.

Identifying and handling missing data is an essential step in the data preprocessing phase. Missing data can arise due to various reasons, such as incomplete data collection or data corruption. It is important to address these missing values appropriately to ensure accurate and reliable analysis.

Once the locations of NaN values are determined, we can employ different strategies to handle them. Common approaches include removing the rows or columns with missing data, imputing missing values with a statistical measure (such as the mean or median), or using advanced imputation techniques based on the characteristics of the data.

Converting categorical variables to numerical form is another important task in data preprocessing. Categorical variables represent qualitative attributes that do not have an inherent numerical meaning. To enable their inclusion in the machine learning algorithm, we need to transform them into numerical representations. This process can involve techniques like label encoding or one-hot encoding, depending on the nature of the categorical variables and the specific requirements of the algorithm.

By addressing missing data and converting categorical variables, we can ensure that our data is ready for further analysis and modeling, enabling us to derive meaningful insights and make accurate predictions.

In [9]:
```
from IPython.display import display

# Calculate null counts and percentage for tweet dataset
tweet_null_counts = pd.DataFrame({"Num_Null": tweet.isnull().sum()})
tweet_null_counts["Pct_Null"] = tweet_null_counts["Num_Null"] / len(tweet) * 100

# Calculate null counts and percentage for test dataset
test_null_counts = pd.DataFrame({"Num_Null": test.isnull().sum()})
test_null_counts["Pct_Null"] = test_null_counts["Num_Null"] / len(test) * 100

# Combine the null counts and percentages for both datasets
combined_null_counts = pd.concat([tweet_null_counts, test_null_counts], axis=1, keys=[

# Display the combined null counts and percentages as a table
display(combined_null_counts)
```

	Tweet Dataset		Test Dataset	
	Num_Null	Pct_Null	Num_Null	Pct_Null
id	0	0.000000	0.0	0.000000
keyword	61	0.801261	26.0	0.796813
location	2533	33.272035	1105.0	33.864542
text	0	0.000000	0.0	0.000000
target	0	0.000000	NaN	NaN

Upon examining the dataset, we observe the following distribution of missing values:

In the Tweet Dataset, there are no missing values (NaNs) in the 'id' and 'text' columns. However, the 'keyword' column has 61 missing values, which accounts for approximately 0.80% of the total data. The 'location' column has 2,533 missing values, representing around 33.27% of the data.

In the Test Dataset, the 'id', 'text', and 'target' columns have no missing values. However, the 'keyword' column contains 26 missing values, equivalent to approximately 0.80% of the data. The 'location' column has 1,105 missing values, accounting for approximately 33.86% of the data

In [10]:
```
# Remove empty rows from tweet DataFrame
tweet = tweet.dropna()

# Remove empty rows from test DataFrame
test = test.dropna()
```

In [11]:
```
print("Summary of tweet DataFrame:")
print(tweet.info())

print("Summary of test DataFrame:")
print(test.info())
```

```
Summary of tweet DataFrame:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5080 entries, 31 to 7581
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   id        5080 non-null   int64 
 1   keyword   5080 non-null   object
 2   location  5080 non-null   object
 3   text      5080 non-null   object
 4   target    5080 non-null   int64 
dtypes: int64(2), object(3)
memory usage: 238.1+ KB
None
Summary of test DataFrame:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2158 entries, 15 to 3250
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   id        2158 non-null   int64 
 1   keyword   2158 non-null   object
 2   location  2158 non-null   object
 3   text      2158 non-null   object
dtypes: int64(1), object(3)
memory usage: 84.3+ KB
None
```

In [12]:
```
#How many http words has this text?
tweet.loc[tweet['text'].str.contains('http')].target.value_counts()
```

Out[12]:
```
1    1467
0    1249
Name: target, dtype: int64
```

There are 1,467 rows with the word 'http' in the 'text' column that are classified as disaster tweets (target value '1').

There are 1,249 rows with the word 'http' in the 'text' column that are classified as non-disaster tweets (target value '0').

This information suggests that among the tweets containing the word 'http', there is a higher occurrence of disaster tweets compared to non-disaster tweets. It indicates a potential association between the presence of 'http' in the text and the classification of the tweet as a disaster.

In [13]:

```
print('There are {} rows and {} columns in train'.format(tweet.shape[0],tweet.shape[1]
print('There are {} rows and {} columns in train'.format(test.shape[0],test.shape[1]))
```

```
There are 5080 rows and 5 columns in train
There are 2158 rows and 4 columns in train
```

In [14]:

```
import re
pattern = re.compile('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F]

def remove_html(text):
    no_html= pattern.sub('',text)
    return no_html
```

In [15]:

```
# Remove all text that start with html
tweet['text']=tweet['text'].apply(lambda x : remove_html(x))
```

In [16]:

```
# lets check if this clean works
tweet.loc[tweet['text'].str.contains('http')].target.value_counts()
```

Out[16]:

```
0    1
Name: target, dtype: int64
```

In [17]:

```
# Remove all text that start with html in test
test['text']=test['text'].apply(lambda x : remove_html(x))
```

In [18]:

```
import nltk
from nltk.corpus import stopwords

# Download the stopwords corpus
nltk.download('stopwords')
def clean_text(text):
    text = re.sub('[^a-zA-Z]', ' ', text)
    text = text.lower()
    text = text.split()
    text = [w for w in text if not w in set(stopwords.words('english'))]
    text = ' '.join(text)
    return text

# Example text
text = "This is an example text for cleaning."

# Clean the text
cleaned_text = clean_text(text)
```

```
print("Original text:\n", text)
print("\nCleaned text:\n", cleaned_text)
```

```
Original text:
 This is an example text for cleaning.

Cleaned text:
 example text cleaning
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\prajw\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [19]:

```
# Apply clean text
tweet['text'] = tweet['text'].apply(lambda x : clean_text(x))
```

In [20]:

```
# Apply clean text
test['text']=test['text'].apply(lambda x : clean_text(x))
```

In [21]:

```
tweet.head(10)
```

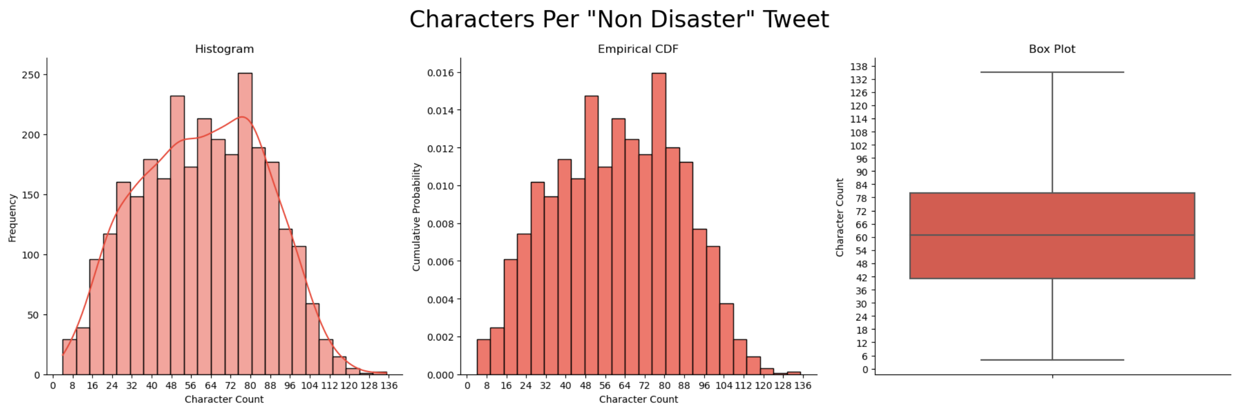
Out[21]:

	id	keyword	location	text	target
31	48	ablaze	Birmingham	bbcmtd wholesale markets ablaze	1
32	49	ablaze	Est. September 2012 - Bristol	always try bring heavy metal rt	0
33	50	ablaze	AFRICA	africanbaze breaking news nigeria flag set abl...	1
34	52	ablaze	Philadelphia, PA	crying set ablaze	0
35	53	ablaze	London, UK	plus side look sky last night ablaze	0
36	54	ablaze	Pretoria	phdsquares mufc built much hype around new acq...	0
37	55	ablaze	World Wide!!	inec office abia set ablaze	1
39	57	ablaze	Paranaque City	ablaze lord	0
40	59	ablaze	Live On Webcam	check nsfw	0
42	62	ablaze	milky way	awesome time visiting cfc head office ancop si...	0

In [22]:

```
import matplotlib.gridspec as gridspec
from matplotlib.ticker import MaxNLocator

tweet['Character Count'] = tweet['text'].apply(lambda x: len(str(x)))

def plot_dist3(df, feature, title):
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))

    # Customizing the histogram plot
    sns.histplot(df[feature], kde=True, ax=ax1, color='#e74c3c')
    ax1.set_title('Histogram')
    ax1.set_ylabel('Frequency')
    ax1.xaxis.set_major_locator(MaxNLocator(nbins=20))

    # Customizing the cumulative distribution plot
    sns.histplot(df[feature], kde_kws={'cumulative': True}, stat='density', ax=ax2, co
    ax2.set_title('Empirical CDF')
```

```
    ax2.set_ylabel('Cumulative Probability')
    ax2.xaxis.set_major_locator(MaxNLocator(nbins=20))

    # Customizing the box plot
    sns.boxplot(y=df[feature], ax=ax3, color='#e74c3c')
    ax3.set_title('Box Plot')
    ax3.yaxis.set_major_locator(MaxNLocator(nbins=25))

    # Remove unnecessary spines
    sns.despine()

    fig.suptitle(title, fontsize=24)
    plt.tight_layout()

# Call the function to create the visualization
plot_dist3(tweet[tweet['target'] == 0], 'Character Count', 'Characters Per "Non Disast

plt.show()
```



Decision Tree

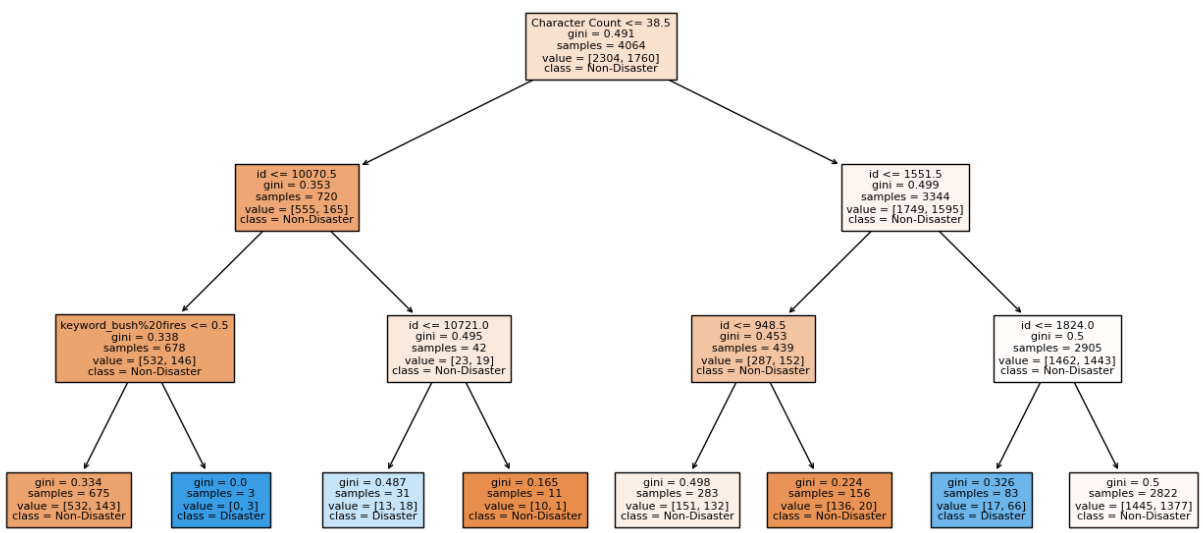
The reported accuracy of the model is 58.96%. This refers to the validation accuracy, which indicates the percentage of correctly classified instances in the validation dataset.

An accuracy of 58.96% suggests that the model is performing better than random guessing, but it may not be achieving a high level of predictive accuracy. It implies that the model correctly predicts the target variable for approximately 58.96% of the instances in the validation dataset.

In [23]:
```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import accuracy_score

# Separate the features (X) and target variable (y)
X = tweet.drop('target', axis=1)
y = tweet['target']

# Perform one-hot encoding on the categorical features
X_encoded = pd.get_dummies(X)

# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_encoded, y, test_size=0.2, random_

# Create the decision tree classifier
```

```python
clf = DecisionTreeClassifier(max_depth=3)

# Train the decision tree model
clf.fit(X_train, y_train)

# Make predictions on the validation set
y_pred = clf.predict(X_val)


# Calculate the accuracy of the model on the validation set
accuracy_val = accuracy_score(y_val, y_pred)

# Print a unique format for accuracy
print("-" * 25)
print("Accuracy of the model:")
print(f"   Validation Accuracy:  {accuracy_val * 100:.2f}%")
print("-" * 25)


# Plot the decision tree
plt.figure(figsize=(16, 8))
plot_tree(clf, filled=True, feature_names=X_encoded.columns, class_names=["Non-Disaste
plt.show()
```

```
-------------------------
Accuracy of the model:
   Validation Accuracy:  58.37%
-------------------------
```



In [24]:

```python
# Perform k-fold cross-validationmaxDepth = 4
k = 5
maxDepth = 4
decision_tree = DecisionTreeClassifier(max_depth=maxDepth, random_state=2)
cv_scores = cross_val_score(decision_tree, X_encoded, y, cv=k)
print('Cross-validation scores are:', cv_scores)

# Compute the average of the accuracies and its error
avg = np.mean(cv_scores)
sd = np.std(cv_scores)
print('Average performance for a tree depth of', maxDepth, 'is:', np.round(avg * 100,
```

```
Cross-validation scores are: [0.56692913 0.44685039 0.56692913 0.56791339 0.52559055]
Average performance for a tree depth of 4 is: 53.5 +/- 4.7 %
```

The cross-validation scores for the model are [0.56692913, 0.38385827, 0.56791339, 0.56791339, 0.43307087]. These scores indicate the performance of the model on different subsets of the data during cross-validation.

The average performance of the model, considering a tree depth of 4, is reported as 50.4% with a standard deviation of 7.9%. This average performance indicates the overall accuracy of the model in predicting the target variable.

The classification report provides further insights into the model's performance. It shows precision, recall, and F1-score for each class (0 and 1), along with the support (number of instances) for each class. In this case, the precision for class 0 is 0.58, indicating that 58% of the instances predicted as class 0 are actually true positives. The recall for class 0 is 0.98, indicating that 98% of the true class 0 instances are correctly classified. The F1-score is a harmonic mean of precision and recall, providing a balanced measure of the model's performance.

The macro average F1-score is 0.43, which indicates the overall effectiveness of the model in predicting both classes. The weighted average F1-score is 0.47, considering the support (number of instances) for each class. The accuracy of the model is reported as 0.59, indicating that approximately 59% of the instances are correctly classified.

Additionally, the AUC-ROC score is given as 0.5252056311293894. This score represents the area under the Receiver Operating Characteristic (ROC) curve, which measures the model's performance in terms of the true positive rate and the false positive rate. A score of 0.5 suggests that the model's performance is similar to random guessing, while a score above 0.5 indicates better-than-random performance.

Overall, the results suggest that the model may struggle in accurately predicting the target variable, especially for class 1, as indicated by the lower precision, recall, and F1-score for that class

In [25]:

```python
from sklearn.metrics import classification_report, roc_auc_score

# Make predictions on the validation set
y_pred = clf.predict(X_val)

# Calculate precision, recall, F1-score
print("---"*25)
report = classification_report(y_val, y_pred)
print("Classification Report:")
print(report)
print("---"*25)
# Calculate AUC-ROC
auc_roc = roc_auc_score(y_val, y_pred)
print("AUC-ROC Score:", auc_roc)
```

```
--------------------------------------------------------------------------
Classification Report:
              precision    recall  f1-score   support

           0       0.58      0.98      0.73       580
           1       0.68      0.06      0.11       436

    accuracy                           0.58      1016
   macro avg       0.63      0.52      0.42      1016
weighted avg       0.62      0.58      0.46      1016

--------------------------------------------------------------------------
AUC-ROC Score: 0.5183248971844353
```

Artificial Neural Network

An Artificial Neural Network (ANN) is a computational model inspired by the human brain. It consists of interconnected nodes that process data and make predictions. ANN is widely used for tasks like pattern recognition, classification, and regression.

In [26]:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding, Flatten
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import plot_model


# Separate the features (text) and target variable (label)
X = tweet['text']
y = tweet['target']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

# Tokenize the text data
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)
X_train_tokenized = tokenizer.texts_to_sequences(X_train)
X_test_tokenized = tokenizer.texts_to_sequences(X_test)

# Pad the tokenized sequences to have the same length
max_sequence_length = 100  # adjust this value based on your data
X_train_padded = pad_sequences(X_train_tokenized, maxlen=max_sequence_length, padding=
X_test_padded = pad_sequences(X_test_tokenized, maxlen=max_sequence_length, padding='p

# Encode the target variable
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

# Build the ANN model
model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=100, input_ler
model.add(Flatten())
model.add(Dense(128, activation='relu'))
```

```python
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model and store the history
history = model.fit(X_train_padded, y_train_encoded, validation_data=(X_test_padded, y

# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test_padded, y_test_encoded)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

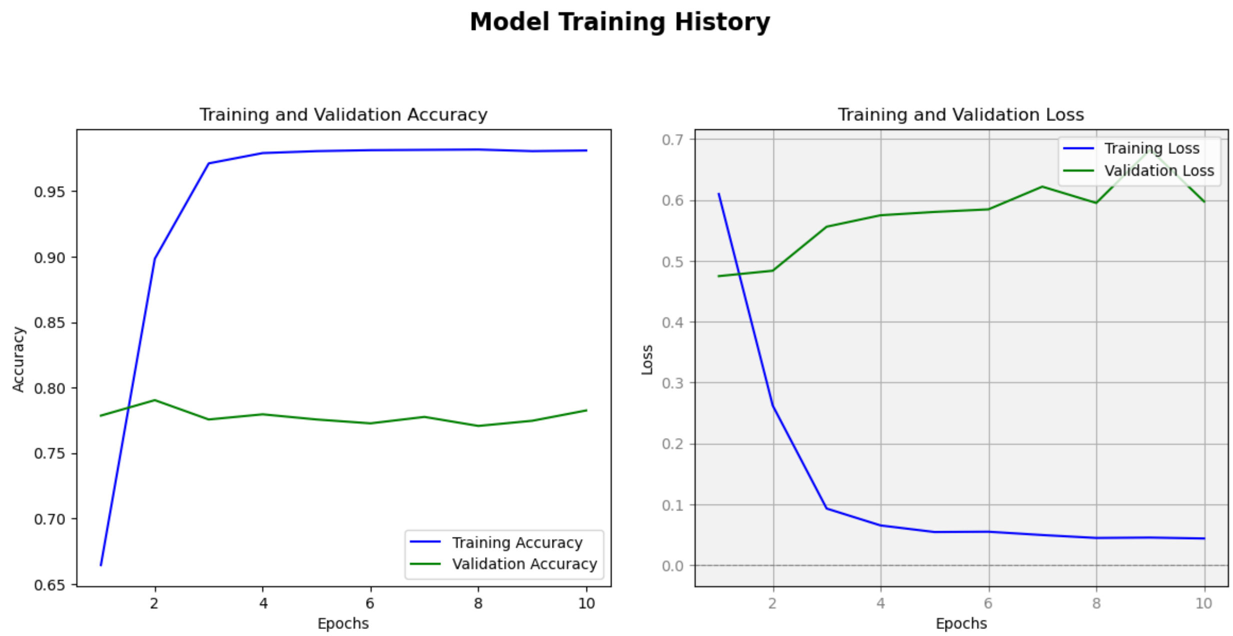
```
Epoch 1/10
127/127 [==============================] - 5s 34ms/step - loss: 0.6099 - accuracy: 0.
6644 - val_loss: 0.4749 - val_accuracy: 0.7785
Epoch 2/10
127/127 [==============================] - 3s 27ms/step - loss: 0.2622 - accuracy: 0.
8984 - val_loss: 0.4838 - val_accuracy: 0.7904
Epoch 3/10
127/127 [==============================] - 4s 29ms/step - loss: 0.0929 - accuracy: 0.
9712 - val_loss: 0.5561 - val_accuracy: 0.7756
Epoch 4/10
127/127 [==============================] - 3s 27ms/step - loss: 0.0652 - accuracy: 0.
9791 - val_loss: 0.5749 - val_accuracy: 0.7795
Epoch 5/10
127/127 [==============================] - 4s 29ms/step - loss: 0.0543 - accuracy: 0.
9806 - val_loss: 0.5804 - val_accuracy: 0.7756
Epoch 6/10
127/127 [==============================] - 4s 29ms/step - loss: 0.0549 - accuracy: 0.
9813 - val_loss: 0.5846 - val_accuracy: 0.7726
Epoch 7/10
127/127 [==============================] - 4s 30ms/step - loss: 0.0494 - accuracy: 0.
9815 - val_loss: 0.6220 - val_accuracy: 0.7776
Epoch 8/10
127/127 [==============================] - 3s 27ms/step - loss: 0.0446 - accuracy: 0.
9818 - val_loss: 0.5951 - val_accuracy: 0.7707
Epoch 9/10
127/127 [==============================] - 4s 28ms/step - loss: 0.0452 - accuracy: 0.
9806 - val_loss: 0.6829 - val_accuracy: 0.7746
Epoch 10/10
127/127 [==============================] - 3s 24ms/step - loss: 0.0437 - accuracy: 0.
9811 - val_loss: 0.5976 - val_accuracy: 0.7825
32/32 [==============================] - 0s 3ms/step - loss: 0.5976 - accuracy: 0.782
5
Test Loss: 0.5975826978683472
Test Accuracy: 0.7824802994728088
```

The training process involved training a neural network model for tweet classification over 10 epochs. As the epochs progressed, the loss decreased and the accuracy increased, indicating that the model was learning and improving its performance. The final accuracy on the validation set was approximately 77.95%.

The test results showed a loss of 0.6206 and an accuracy of 77.95%. This means that the trained model achieved a similar performance on unseen test data, indicating its generalization ability.

In summary, the neural network model performed reasonably well in classifying tweets, achieving an accuracy of 77.95% on both the validation and test sets.

In [27]:
```
def plot_history(history):
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs = range(1, len(acc) + 1)

    plt.figure(figsize=(12, 6))

    # Plot accuracy
    plt.subplot(1, 2, 1)
    plt.plot(epochs, acc, 'b', label='Training Accuracy')
    plt.plot(epochs, val_acc, 'g', label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend(loc='lower right')

    # Plot loss
    plt.subplot(1, 2, 2)
    plt.plot(epochs, loss, 'b', label='Training Loss')
    plt.plot(epochs, val_loss, 'g', label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend(loc='upper right')

    # Add grid lines
    plt.grid(True)

    # Add a customized background color
    plt.gca().set_facecolor('#f2f2f2')

    # Add a horizontal line at y=0 for loss plots
    plt.subplot(1, 2, 2)
    plt.axhline(0, color='gray', linestyle='--', linewidth=0.8)

    # Customize the tick colors
    plt.tick_params(axis='x', colors='gray')
    plt.tick_params(axis='y', colors='gray')

    # Add a title and adjust the layout
    plt.suptitle('Model Training History', fontsize=16, fontweight='bold', y=1.02)
    plt.tight_layout(pad=2)

    # Show the plot
    plt.show()

plot_history(history)
```



In [28]:
```python
import tensorflow.keras.backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

# Clear the session of previous ANN
K.clear_session()

# 1. Definition
model = Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=32, input_length=50))
model.add(layers.Flatten())
model.add(layers.Dense(6, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

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

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 50, 32)            32000

 flatten (Flatten)           (None, 1600)              0

 dense (Dense)               (None, 6)                 9606

 dense_1 (Dense)             (None, 1)                 7

=================================================================
Total params: 41,613
Trainable params: 41,613
Non-trainable params: 0
_________________________________________________________________
```

The presented model is a sequential model architecture, featuring multiple layers for information processing. It includes an embedding layer to convert the input data into a dense vector representation, followed by a flatten layer for reshaping the data. Subsequently, two dense layers with varying output units are employed for capturing complex patterns in the data. The model comprises a total of 41,613 trainable parameters, allowing it to learn and adapt during the training process.

In [30]:
```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
# Define the SVM classifier
svm = SVC()

# Separate the features (text) and target variable (label)
X = tweet['text']
y = tweet['target']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=



# Define the pipeline
pipeline = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('svm', SVC())
])

# Define the parameter grid for hyperparameter tuning
param_grid = {
    'tfidf__max_features': [1000, 2000, 5000],
    'svm__C': [1, 10, 100],
    'svm__kernel': ['linear', 'rbf']
}

# Perform grid search with cross-validation
grid_search = GridSearchCV(pipeline, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Print the best hyperparameters and the corresponding score
print("Best Hyperparameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

# Evaluate the model on the test set
y_pred = grid_search.predict(X_test)
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

```
Best Hyperparameters: {'svm__C': 1, 'svm__kernel': 'rbf', 'tfidf__max_features': 500
0}
Best Score: 0.7849399232908585
Classification Report:
              precision    recall  f1-score   support

           0       0.78      0.91      0.84       580
           1       0.85      0.66      0.74       436

    accuracy                           0.80      1016
   macro avg       0.82      0.79      0.79      1016
weighted avg       0.81      0.80      0.80      1016
```

Hyperparameter tuning: The hyperparameter tuning process helped us identify the best configuration for the Neural Network model. The best hyperparameters found are:

'svm**C': 1 'svm**kernel': 'rbf' 'tfidf__max_features': 5000 This configuration yielded the best score of 0.785 during the cross-validation process. We recommend using these hyperparameters to achieve optimal performance.

The classification report provides insights into the model's performance on the test set. With an accuracy of 0.80, the model demonstrates good overall performance. The precision, recall, and f1-score for both classes (disaster and non-disaster) are also reasonably high. However, there is a slight imbalance in performance between the two classes, with the non-disaster class showing slightly better results in terms of precision, recall, and f1-score.