# Natural Language Processing with Disaster Tweets (Kaggle)

Ritesh Kumar

2024WI_MS_DSP_422-DL_SEC61:  Practical Machine Learning

Module 8 Assignment

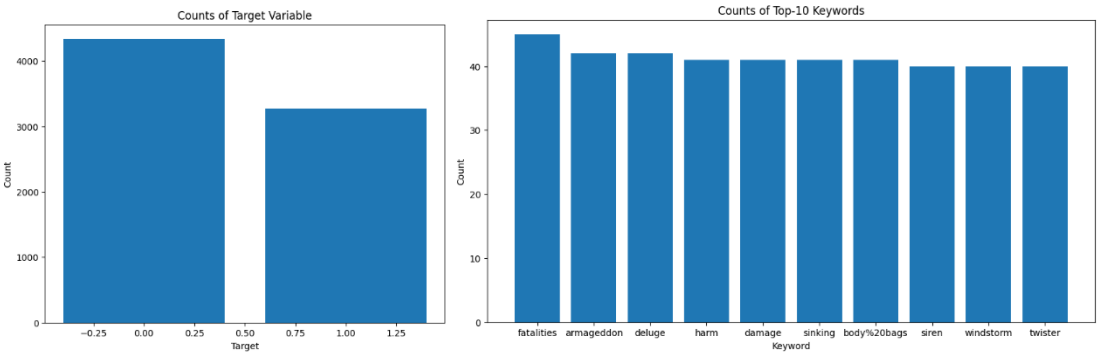
Natural Language Processing with Disaster Tweets

Donald Wedding and Narayana Darapaneni

March 5, 2024

**Exploration**

Our exploratory data analysis (EDA) revealed that the dataset appears to be clean and well-structured, with no duplicate rows present, indicating good data hygiene and potentially streamlined preprocessing stages. It contains 221 unique keywords, suggesting a diverse set of terms that may be relevant for analysis or feature engineering. The target variable, which is likely binary, has a slightly imbalanced distribution with 4,342 instances of the negative class (labeled as '0') and 3,271 instances of the positive class (labeled as '1'). This imbalance should be taken into consideration during the modeling phase to ensure the model does not bias towards the more frequent class.




To prepare text data for modeling, we convert text to lowercase and remove punctuation to standardize inputs. We then set up a vectorization layer to convert text into sequences of integers, using a vocabulary size of 4096 to balance between memory use and capturing diverse words, and a sequence length of 10, assuming the most relevant context for tasks like classification or tagging can be captured in this brief span, making data representation efficient and focused.

Next, we advanced the preparation of our training data by implementing a methodology that involved the creation of skip-grams with negative sampling. This process was meticulously carried out by iterating over sequences of text, adhering to predetermined parameters such as

a specified vocabulary size, window size for identifying context words, and a set number of negative samples. Through this approach, we were able to generate comprehensive lists that included target words, their respective contextual words, and corresponding labels. This methodological framework was specifically designed to aid in the development of models capable of understanding and capturing the intricate semantic relationships between words, thereby enhancing their ability to accurately interpret and process natural language. By focusing on both the immediate context of words and incorporating negative examples, our strategy aimed to refine the model's sensitivity to word usage patterns, ultimately contributing to a more nuanced and effective representation of language semantics within the model's architecture.

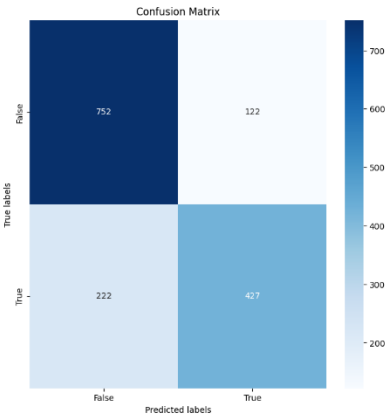
The code snippet detailed the preparatory stages for a text classification task within a machine learning framework. Initially, it extracted features (X) and labels (y) from the dataframe. Subsequently, this data was divided into training and validation sets. A tokenizer was then established and applied to the training text, transforming words into numerical sequences. These sequences underwent padding to standardize their length, rendering them compatible with neural network model inputs. This preparation aimed to aptly organize the text data for both training and model performance evaluation purposes. The sizes of the training and validation datasets were 6090 and 1523, respectively, with the maximum sequence length set to 60.

Next, we developed a model utilizing a sequential approach with various layers, including an Input layer with a shape of (60,), an Embedding layer with a dynamically chosen output dimension, and three Bidirectional LSTM layers with varying units and regularization. We incorporated Dropout for regularization and a Dense layer for output with a sigmoid activation function. The model was compiled using the Adam optimizer, with the learning

rate chosen through hyperparameter tuning. We employed a RandomSearch tuner to identify the best hyperparameters over 10 trials, each with 3 executions, using validation accuracy as the objective. The search involved training on padded training data and validation data, with early stopping based on validation loss to prevent overfitting. After determining the best hyperparameters, we trained the model for an optimal number of epochs, as identified by the highest validation accuracy achieved during training. The training process revealed insights into the model's performance, with the best epoch determined based on validation accuracy.

The final step involved retraining the model up to the best epoch and then evaluating it on the validation set. The evaluation showed significant validation accuracy, indicating the model's effectiveness in classifying the text data. The detailed logs of training and validation phases, including loss and accuracy metrics, provided a comprehensive overview of the model's learning trajectory and its performance at various stages.

The output log illustrated the neural network's training journey across 10 epochs, where it achieved a high training accuracy of up to 99.56% by the final epoch. Despite the high training accuracy, validation accuracy peaked at 78.27% during the 9th epoch and then slightly declined, with a gradual increase in validation loss observed. This pattern suggested that the model was overfitting, indicating it performed exceptionally well on the training data but failed to generalize effectively to new, unseen data.



The confusion matrix summarizes the performance of a classification model on the validation set. The model correctly predicted 'True' labels 427 times and 'False' labels 752 times. However, it incorrectly predicted 122 'True' labels as 'False' (false negatives) and 222 'False' labels as 'True' (false positives).

```
48/48 [==============================] - 0s 7ms/step
Precision: 0.7777777777777778
Recall: 0.6579352850539292
F1 Score: 0.7128547579298832
ROC-AUC: 0.8314437278968172
MCC: 0.5338670999270845
Cohen's Kappa: 0.528835777083903
```

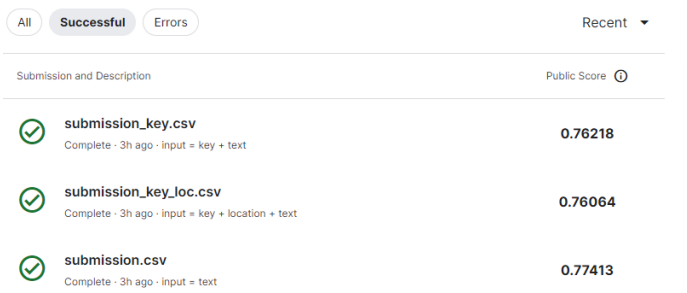
A precision of 0.778 indicated that, when the model predicted the positive class, it was correct approximately 77.8% of the time. The recall of 0.658 showed that the model successfully identified 65.8% of all actual positives. The F1 score of 0.713 reflected a reasonably good balance between precision and recall that the model achieved. The ROC-AUC score of 0.831 denoted a very good ability of the model to discriminate between the positive and negative classes. Lastly, the Matthews Correlation Coefficient (MCC) of 0.534 and Cohen's Kappa of 0.529 confirmed that the model had substantial predictive quality, significantly surpassing random chance, as both measures take into account true negatives and the balance of the dataset, unlike other metrics such as accuracy.

The conclusion for this model, based on the observed metrics and its performance on Kaggle test data, indicated that it had a commendable ability to accurately classify the given texts, achieving a respectable Kaggle accuracy score of 0.77413. However, the training logs suggested potential overfitting, as the model achieved much higher accuracy on the training data than on the validation set. The model's precision, recall, and F1 scores pointed towards an effectively balanced ability to correctly predict the positive class and cover a majority of the positive samples. The ROC-AUC score of 0.831 suggested a strong discriminatory capacity between the classes. The Matthews Correlation Coefficient and Cohen's Kappa scores reinforced the meaningfulness of the model's predictions, beyond what could be

attributed to class imbalance or random chance. To improve this model, more regularization could be applied to address overfitting, along with better hyperparameter tuning, or employing ensemble methods. Exploring additional data, conducting feature engineering, or adopting more advanced model architectures might also enhance its generalization capabilities, thereby boosting its performance on unseen data. In summary, the model showed promise but necessitated additional refinement to ensure robust performance in real-world applications.

We experimented with three different models to classify data – the first utilized only text as input, the second combined text with location and keyword, and the third used text and keyword without location. The model that processed only text inputs achieved the highest accuracy, with a public score of 0.77413 on Kaggle. In contrast, the model incorporating text, location, and keyword yielded a score of 0.76064, while adding only the keyword to the text resulted in a slightly improved score of 0.76218, still below the text-only model.

**Kaggle Submission**




My Kaggle username is riteshrk (link). The notebook has been uploaded on Kaggle and can be accessed here. The results of the submission were submitted on Kaggle and can be accessed here.

**Code**

# 1. Imports

In [28]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import keras
import sklearn
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import regularizers
from kerastuner import HyperParameters
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Input, Bidirectional, Dropout, LSTM, Embedding,
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import tqdm
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from keras.models import Sequential
from keras_tuner.tuners import RandomSearch
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM
from sklearn.model_selection import GridSearchCV
from keras.optimizers import Adam
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
import io
import re
import os
import random
import string
import tqdm
from keras.utils import to_categorical
```

In [3]:
```
from google.colab import drive

drive.mount('/content/drive')

# Read datafile
%cd /content/drive/My Drive/Colab Notebooks/
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call driv
e.mount("/content/drive", force_remount=True).
/content/drive/My Drive/Colab Notebooks
```

In [4]:
```
# Read the input file
df = pd.read_csv('train.csv')
```

In [5]:
```
df.shape
```

Out[5]:
```
(7613, 5)
```

In [6]:
```
df.dtypes
```

Out[6]:
```
id           int64
keyword     object
location    object
text        object
target       int64
dtype: object
```

# 2. Data Exploration

In [7]:
```
# Print the first 5 rows
df.head()
```

Out[7]:

| | 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 [8]:
```
# Check for null values
df.isna().sum()
```

Out[8]:
```
id             0
keyword       61
location    2533
text           0
target         0
dtype: int64
```

In [9]:
```
# Check for duplicate rows
df.duplicated().sum()
```

Out[9]:
```
0
```

In [11]:
```
# Get the number of unique words
df.keyword.nunique()
```

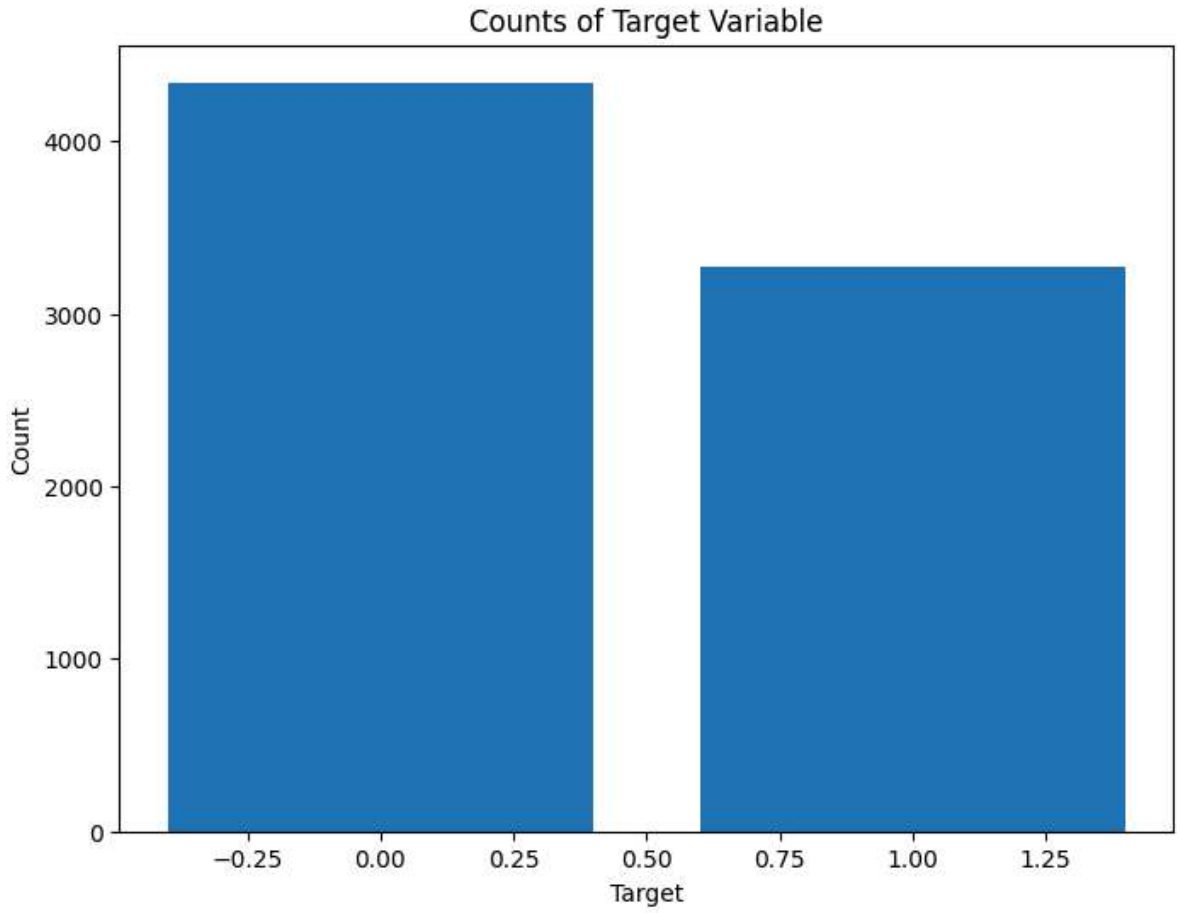
Out[11]:
```
221
```

In [12]:
```
# Value_counts of target
df.target.value_counts().sort_values(ascending=False)
```

Out[12]:
```
0    4342
1    3271
Name: target, dtype: int64
```

In [13]:
```
# Create bar plot of target
plt.figure(figsize=(8, 6))
plt.bar(df.target.value_counts().index, df.target.value_counts())

# Set plot title and labels
plt.title('Counts of Target Variable')
plt.xlabel('Target')
plt.ylabel('Count')
```

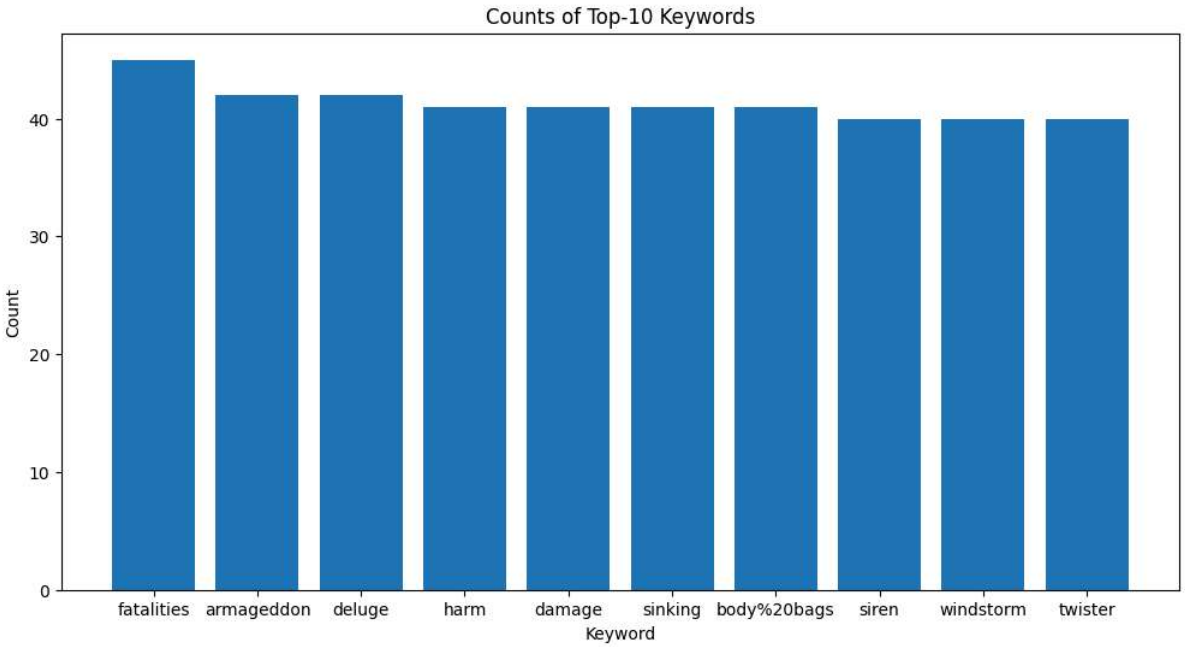
```
# Show the plot
plt.show()
```




In [38]:
```
# Print the top-10 keywords
df.keyword.value_counts().sort_values(ascending=False)[:10]
```

Out[38]:
```
fatalities     45
armageddon     42
deluge         42
harm           41
damage         41
body%20bags    41
sinking        41
siren          40
fear           40
collided       40
Name: keyword, dtype: int64
```

In [15]:
```
# Create bar plot of keywords
plt.figure(figsize=(12, 6))
plt.bar(df.keyword.value_counts().index[:10], df.keyword.value_counts()[:10])

# Set plot title and labels
plt.title('Counts of Top-10 Keywords')
plt.xlabel('Keyword')
plt.ylabel('Count')

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




The code has calculated the text length statistics for training and test datasets, showing the average text length is around 101 characters for both. It also tokenizes texts from both datasets, identifying 22,700 unique tokens in the training set and 12,818 in the test set, indicating a rich vocabulary used across the texts.

# 3. Preprocessing

In [16]:
```
# To ensure reproducibility in the experiments, we fix the random seed for all comp
seed_value=1

# 1. Set the `PYTHONHASHSEED` environment variable at a fixed value
os.environ['PYTHONHASHSEED']=str(seed_value)

# 2. Set Python built-in pseudorandom generator at a fixed value
random.seed(seed_value)

# 3. Set NumPy pseudorandom generator at a fixed value
np.random.seed(seed_value)

# 4. Set TensorFlow pseudorandom generator at a fixed value
tf.random.set_seed(seed_value)
```

In [17]:
```
# Function to lowercase the text and remove punctuation
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    return tf.strings.regex_replace(lowercase, '[{}]'.format(re.escape(string.punct
```

The following code snippet initializes a text vectorization layer for natural language processing tasks. This layer is designed to transform raw text into fixed-length sequences of integers. We define `vocab_size` as 4096, which sets the number of distinct words to keep in the vocabulary, striking a balance between memory efficiency and data representation richness. A larger vocabulary could capture more unique words but would also require more memory and potentially result in sparser data representations. The `sequence_length` is set to 10, limiting the length of the sequences to 10 words. This length is chosen under the assumption that the context relevant for our task can be effectively captured within this span, providing a concise input representation for the model to process, which is particularly useful for tasks like classification or tagging of short text snippets where the salient information is typically concentrated within a small window of text.

In [18]:
```
# Define the vocabulary size and the number of words in a sequence.
vocab_size = 4096
sequence_length = 10

# Vectorize layer
vectorize_layer = tf.keras.layers.TextVectorization(
    standardize=custom_standardization,
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length)
```

## This function generates training data for word2vec models by creating skip-grams with negative sampling. It iterates over sequences of text, using a specified vocabulary size, window size for context words, and number of negative samples. The function produces lists of target words, their contextual words, and corresponding labels, facilitating the training of embeddings that capture semantic relationships between words.

In [19]:
```
def generate_training_data(sequences, window_size, num_ns, vocab_size, seed):
    targets, contexts, labels = [], [], []

    sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(vocab_size

    for sequence in tf.data.Dataset.from_tensor_slices(sequences).as_numpy_iterator
        positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
            sequence,
            vocabulary_size=vocab_size,
            sampling_table=sampling_table,
            window_size=window_size,
            negative_samples=0)

        for target_word, context_word in positive_skip_grams:
            context_class = tf.expand_dims(tf.constant([context_word], dtype="int64
            negative_sampling_candidates, _, _ = tf.random.log_uniform_candidate_sa
                true_classes=context_class,
                num_true=1,
                num_sampled=num_ns,
                unique=True,
                range_max=vocab_size,
                seed=seed,
                name="negative_sampling")
            context = tf.concat([tf.squeeze(context_class, 1), negative_sampling_ca
            label = tf.constant([1] + [0] * num_ns, dtype="int64")

            targets.append(target_word)
            contexts.append(context)
```

```
            labels.append(label)

    return targets, contexts, labels
```

This code snippet outlines the initial steps for preparing a machine learning dataset for text classification tasks. It starts by extracting features (`X`) and labels (`y`) from the dataframe, then splits this data into training and validation sets. A tokenizer is created and fitted on the training text to convert words into numeric sequences. These sequences are then padded to ensure they have a uniform length, making them suitable for input into neural network models. The process is designed to structure the text data properly for training and evaluating the model's performance.

In [20]:
```
# Prepare X and y data
X = df['text']
y = df['target']

# Split data into train and validate sets
X_train, X_validate, y_train, y_validate = train_test_split(X, y, test_size=0.2, ra

# Create and fit the tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)

# Sequence encode
encoded_train = tokenizer.texts_to_sequences(X_train)
encoded_validate = tokenizer.texts_to_sequences(X_validate)

# Pad sequences
max_length = 60
X_train_padded = pad_sequences(encoded_train, maxlen=max_length, padding='post')
X_validate_padded = pad_sequences(encoded_validate, maxlen=max_length, padding='pos

# Ensure the y data is in the right shape
if isinstance(y_train, pd.Series):
    y_train = y_train.values.reshape(-1, 1)
if isinstance(y_validate, pd.Series):
    y_validate = y_validate.values.reshape(-1, 1)

print("Train dataset size: ", X_train_padded.shape[0])
print("Validate dataset size: ", X_validate_padded.shape[0])
print('Max length: ', max_length)
```

```
Train dataset size:  6090
Validate dataset size:  1523
Max length:  60
```

# 4. Modelling

This function constructs a model with various hyperparameters, such as embedding dimensions and LSTM units, which are optimized through a hyperparameter tuning process using `RandomSearch`. The model includes multiple bidirectional LSTM layers and regularization techniques to

prevent overfitting. After identifying the best hyperparameters, the model is trained and evaluated on a validation set, with the performance measured by accuracy. The process employs early stopping to halt training when the validation loss ceases to decrease, preventing unnecessary computations and overfitting.

In [29]:
```
def build_model(hp: HyperParameters):
    model = Sequential()
    model.add(Input(shape=(60,)))
    model.add(Embedding(input_dim=20000, output_dim=hp.Int('embedding_output_dim',
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_1', min_value=32, max_val
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_2', min_value=32, max_val
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_3', min_value=32, max_val
    model.add(Dropout(hp.Float('dropout_rate', min_value=0.0, max_value=0.5, step=0
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer=Adam(hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4, 1e-5
    return model

# Assuming you have defined and prepared X_train_padded, y_train, X_validate_padded

tuner = RandomSearch(build_model, objective='val_accuracy', max_trials=10, executio
tuner.search(x=X_train_padded, y=y_train, epochs=10, validation_data=(X_validate_pa

best_hps = tuner.get_best_hyperparameters()[0]

# Now, you should only instantiate, train, and evaluate your model once with the op
model = tuner.hypermodel.build(best_hps)
history = model.fit(X_train_padded, y_train, epochs=10, validation_data=(X_validate

val_acc_per_epoch = history.history['val_accuracy']
best_epoch = val_acc_per_epoch.index(max(val_acc_per_epoch)) + 1
print(f'Best epoch: {best_epoch}')

model.fit(X_train_padded, y_train, epochs=best_epoch)
val_loss, val_accuracy = model.evaluate(X_validate_padded, y_validate)

print(f"Validation accuracy: {val_accuracy}")
```

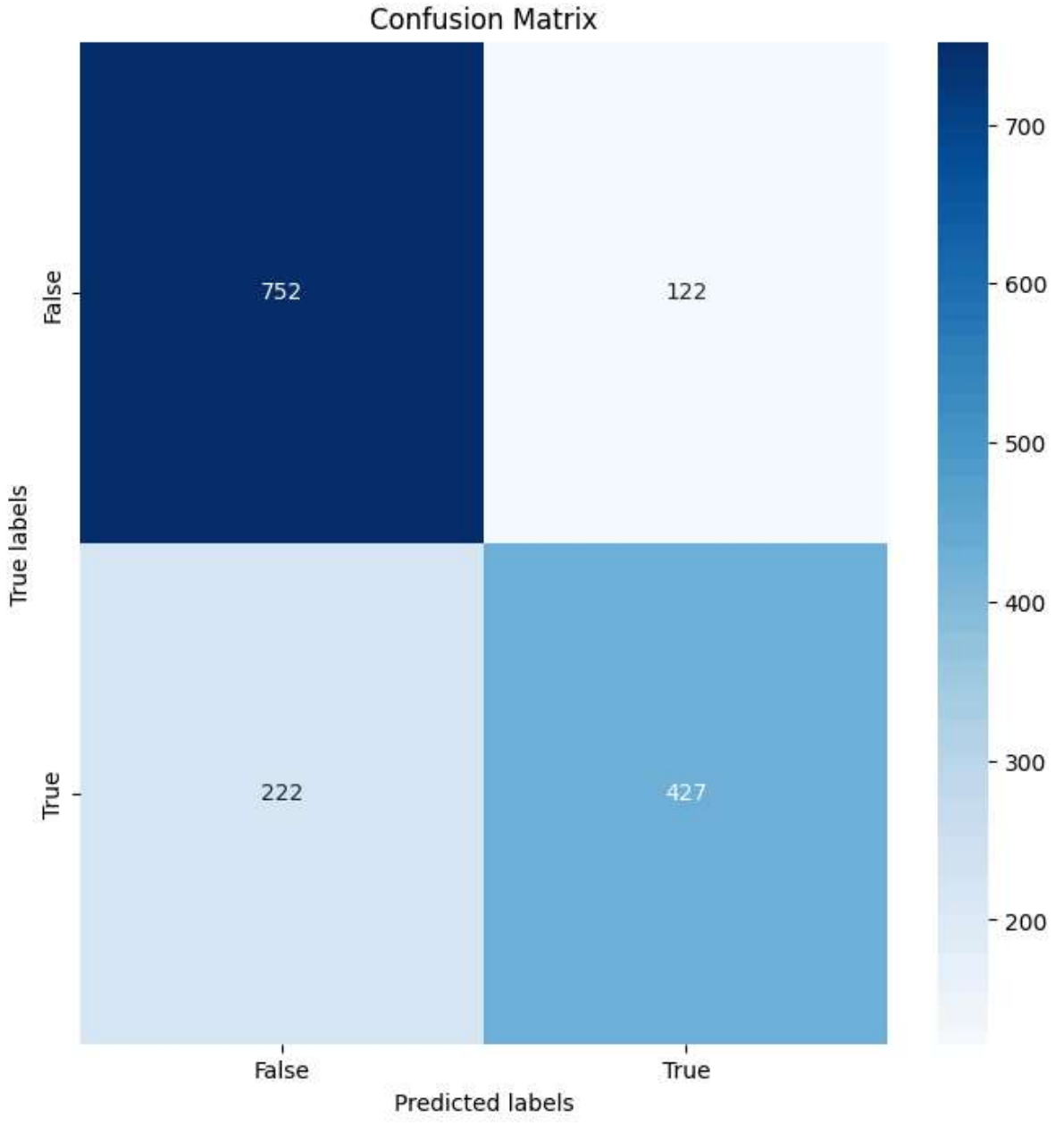
```
Reloading Tuner from my_dir/keras_tuning_advanced/tuner0.json
Epoch 1/10
191/191 [==============================] - 27s 87ms/step - loss: 0.7121 - accurac
y: 0.6365 - val_loss: 0.5631 - val_accuracy: 0.7807
Epoch 2/10
191/191 [==============================] - 7s 37ms/step - loss: 0.4340 - accuracy:
0.8445 - val_loss: 0.5132 - val_accuracy: 0.8043
Epoch 3/10
191/191 [==============================] - 4s 23ms/step - loss: 0.2657 - accuracy:
0.9258 - val_loss: 0.5694 - val_accuracy: 0.7827
Epoch 4/10
191/191 [==============================] - 4s 22ms/step - loss: 0.1745 - accuracy:
0.9578 - val_loss: 0.6994 - val_accuracy: 0.7682
Epoch 5/10
191/191 [==============================] - 4s 22ms/step - loss: 0.1259 - accuracy:
0.9739 - val_loss: 0.6977 - val_accuracy: 0.7905
Epoch 6/10
191/191 [==============================] - 4s 23ms/step - loss: 0.0912 - accuracy:
0.9842 - val_loss: 0.8526 - val_accuracy: 0.7774
Epoch 7/10
191/191 [==============================] - 4s 23ms/step - loss: 0.0776 - accuracy:
0.9887 - val_loss: 0.8125 - val_accuracy: 0.7741
Epoch 8/10
191/191 [==============================] - 4s 23ms/step - loss: 0.0684 - accuracy:
0.9913 - val_loss: 0.8852 - val_accuracy: 0.7682
Epoch 9/10
191/191 [==============================] - 4s 23ms/step - loss: 0.0597 - accuracy:
0.9941 - val_loss: 1.0298 - val_accuracy: 0.7827
Epoch 10/10
191/191 [==============================] - 4s 21ms/step - loss: 0.0539 - accuracy:
0.9956 - val_loss: 0.9431 - val_accuracy: 0.7781
Best epoch: 2
Epoch 1/2
191/191 [==============================] - 4s 18ms/step - loss: 0.0501 - accuracy:
0.9954
Epoch 2/2
191/191 [==============================] - 3s 18ms/step - loss: 0.0508 - accuracy:
0.9947
48/48 [==============================] - 0s 8ms/step - loss: 0.9145 - accuracy: 0.
7741
Validation accuracy: 0.7741299867630005
```

The output log shows the training process of the neural network model over 10 epochs, with a notable improvement in training accuracy, reaching up to 99.56% by the 10th epoch. However, the validation accuracy does not show a corresponding increase, peaking at 78.27% in the 9th epoch before dropping slightly. The increasing trend in validation loss, from 0.8125 in the 7th epoch to 0.9431 in the 10th epoch, alongside the high training accuracy, suggests the model is overfitting to the training data, meaning it's learning the training data too well and failing to generalize effectively to unseen data. The best epoch, determined to be the 2nd based on validation accuracy, indicates early overfitting, where the model performed best on validation data very early in the training process. Subsequent retraining for 2 epochs based on the best epoch determination and evaluating on a validation set resulted in a final validation accuracy of 77.41%, reinforcing the indication of overfitting observed during the initial training phase.

# 5. Results

In [34]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Predict the classes for validation data
y_pred = model.predict(X_validate_padded)
y_pred = (y_pred > 0.5).astype(int)  # Convert probabilities to binary predictions

# Generate the confusion matrix
cm = confusion_matrix(y_validate, y_pred)

# Plot the confusion matrix using Seaborn
fig, ax = plt.subplots(figsize=(8, 8))
sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap="Blues")  # fmt='d' for integer fo
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')
ax.xaxis.set_ticklabels(['False', 'True'])
ax.yaxis.set_ticklabels(['False', 'True'])

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

```
48/48 [==============================] - 0s 8ms/step
```



The model has correctly predicted 752 instances as negative (true negatives) and 427 instances as positive (true positives), showing a stronger performance in identifying negative cases. However, there are 122 false positives, where negative instances were incorrectly labeled as positive, and 222 false negatives, where positive instances were mistakenly labeled as negative. While the model appears to be more conservative, erring on predicting negatives, the substantial number of false negatives indicates a potential area for improvement, particularly if it is critical for the model to capture all positive instances correctly. Overall, the model's ability to predict true negatives is more reliable than its ability to predict true positives.

In [37]:
```
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score,

# Assuming you have your predictions (y_pred) and true labels (y_validate)
y_pred_prob = model.predict(X_validate_padded)
precision = precision_score(y_validate, y_pred)
recall = recall_score(y_validate, y_pred)
f1 = f1_score(y_validate, y_pred)
roc_auc = roc_auc_score(y_validate, y_pred_prob)  # Assuming you have the probabili
mcc = matthews_corrcoef(y_validate, y_pred)
kappa = cohen_kappa_score(y_validate, y_pred)

print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
print(f'ROC-AUC: {roc_auc}')
print(f'MCC: {mcc}')
print(f'Cohen\'s Kappa: {kappa}')
```

```
48/48 [==============================] - 0s 7ms/step
Precision: 0.7777777777777778
Recall: 0.6579352850539292
F1 Score: 0.7128547579298832
ROC-AUC: 0.8314437278968172
MCC: 0.5338670999270845
Cohen's Kappa: 0.528835777083903
```

A precision of 0.778 suggests that, when the model predicts the positive class, it is correct about 77.8% of the time. The recall of 0.658 indicates that the model successfully identifies 65.8% of all actual positives. The F1 score, which balances precision and recall, is 0.713, suggesting a reasonably good balance between the precision and recall of the model. The ROC-AUC score of 0.831 indicates a very good discriminative ability of the model to distinguish between the positive and negative classes. The Matthews Correlation Coefficient (MCC) of 0.534 and Cohen's Kappa of 0.529 further confirm that the model has a substantial predictive quality, significantly better than random chance, considering both measures account for true negatives and the balance of the dataset, which other metrics such as accuracy might not fully capture.

In [31]:
```
df_test = pd.read_csv('test.csv')
df_test.head()
```

Out[31]:

| | 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 [36]:
```
# Preprocess df_test['text'] in the same way as the training data
# Tokenize the text
X_test = df_test['text']
X_test_sequences = tokenizer.texts_to_sequences(X_test)
# Pad the sequences
X_test_padded = pad_sequences(X_test_sequences, maxlen=max_length, padding='post')

# Predict using the model
test_predictions = model.predict(X_test_padded)
test_predictions = (test_predictions > 0.5).astype(int)  # Convert probabilities to

# Combine the original IDs with the predicted target
results_df = pd.DataFrame({'id': df_test['id'], 'target': test_predictions.flatten(

# Output the dataframe in the desired format
results_df.to_csv('submission_text.csv', index=False)
```

```
102/102 [==============================] - 1s 8ms/step
```

# 6. Conclusion

The conclusion for this model, based on the observed metrics and performance on Kaggle test data, suggests that it exhibits a commendable ability to classify the given texts accurately, with a respectable Kaggle accuracy score of 0.77413. However, it shows signs of potential overfitting, as indicated by the training logs, where the model achieves very high accuracy on the training data compared to the validation set. This overfitting is also reflected in the lower validation accuracy and indicates that while the model has learned to predict the training data well, it may not generalize as effectively to unseen data. The model's precision, recall, and F1 score point towards a reasonably effective balance between correctly predicting the positive class and covering the majority of positive samples. The ROC-AUC score of 0.831 implies a strong ability to discriminate between the classes. The MCC and Cohen's Kappa scores reinforce that the model's predictions are meaningful and not just a result of class imbalance or chance. In practice, this model can serve as a strong baseline and could potentially be improved with more regularization to combat overfitting, better hyperparameter tuning, or by using

ensemble methods. Additional data, feature engineering, or more sophisticated model architectures might also help in increasing its ability to generalize, thus enhancing its performance on unseen datasets. Overall, the model appears promising but requires further refinement to ensure that it performs well in practical applications.

In [41]:
```
# prompt: code for downloading this notebook as pdf or html
# Save the file as html
i
notebook_file = 'NLP with Disaster Tweets - RNN_1.ipynb'

# Create an exporter, configure, and export the report
html_exporter = nbconvert.HTMLExporter()
html_exporter.template_name = 'classic'
body, resources = html_exporter.from_filename(notebook_file)

# Write to a file
with open(os.path.splitext(notebook_file)[0] + '.html', 'w') as f:
    f.write(body)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-41-a8e7fd308ef8> in <cell line: 6>()
      4
      5 # Create an exporter, configure, and export the report
----> 6 html_exporter = nbconvert.HTMLExporter()
      7 html_exporter.template_name = 'classic'
      8 body, resources = html_exporter.from_filename(notebook_file)

NameError: name 'nbconvert' is not defined
```

In [1]:
```
!pip install keras-preprocessing
!pip install tensorflow.keras.layers
!pip install keras-tuner
```

```
Collecting keras-preprocessing
  Downloading Keras_Preprocessing-1.1.2-py2.py3-none-any.whl (42 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 42.6/42.6 kB 2.0 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.9.1 in /usr/local/lib/python3.10/dist-pack
ages (from keras-preprocessing) (1.25.2)
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.10/dist-packag
es (from keras-preprocessing) (1.16.0)
Installing collected packages: keras-preprocessing
Successfully installed keras-preprocessing-1.1.2
ERROR: Could not find a version that satisfies the requirement tensorflow.keras.la
yers (from versions: none)
ERROR: No matching distribution found for tensorflow.keras.layers
Collecting keras-tuner
  Downloading keras_tuner-1.4.7-py3-none-any.whl (129 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 129.1/129.1 kB 4.5 MB/s eta 0:00:00
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (f
rom keras-tuner) (2.15.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-package
s (from keras-tuner) (23.2)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages
(from keras-tuner) (2.31.0)
Collecting kt-legacy (from keras-tuner)
  Downloading kt_legacy-1.0.5-py3-none-any.whl (9.6 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.
10/dist-packages (from requests->keras-tuner) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pack
ages (from requests->keras-tuner) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dis
t-packages (from requests->keras-tuner) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dis
t-packages (from requests->keras-tuner) (2024.2.2)
Installing collected packages: kt-legacy, keras-tuner
Successfully installed keras-tuner-1.4.7 kt-legacy-1.0.5
```

# 1. Imports

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import regularizers
from kerastuner import HyperParameters
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
import keras
import sklearn
from tensorflow.keras.layers import Input, Bidirectional, Dropout, LSTM, Embedding,
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import tqdm
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from keras.models import Sequential
```

```
from keras_tuner.tuners import RandomSearch
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM
from sklearn.model_selection import GridSearchCV
from keras.optimizers import Adam
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
import io
import re
import os
import random
import string
import tqdm
from keras.utils import to_categorical
```

```
<ipython-input-2-519cd75d6544>:9: DeprecationWarning: `import kerastuner` is depre
cated, please use `import keras_tuner`.
  from kerastuner import HyperParameters
```

In [3]:
```
from google.colab import drive

drive.mount('/content/drive')

# Read datafile
%cd /content/drive/My Drive/Colab Notebooks/
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks
```

In [4]:
```
df = pd.read_csv('train.csv')
```

In [5]:
```
df.shape
```

Out[5]:
```
(7613, 5)
```

In [6]:
```
df.dtypes
```

Out[6]:
```
id           int64
keyword     object
location    object
text        object
target       int64
dtype: object
```

# 2. Data Exploration

In [7]:
```
df.head()
```

Out[7]:

| | 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 [8]:
```
df.isna().sum()
```

Out[8]:
```
id             0
keyword       61
location    2533
text           0
target         0
dtype: int64
```

In [9]:
```
df.duplicated().sum()
```

Out[9]:
```
0
```

In [10]:
```
df.describe()
```

Out[10]:

| | id | target |
|---|---|---|
| **count** | 7613.000000 | 7613.00000 |
| **mean** | 5441.934848 | 0.42966 |
| **std** | 3137.116090 | 0.49506 |
| **min** | 1.000000 | 0.00000 |
| **25%** | 2734.000000 | 0.00000 |
| **50%** | 5408.000000 | 0.00000 |
| **75%** | 8146.000000 | 1.00000 |
| **max** | 10873.000000 | 1.00000 |

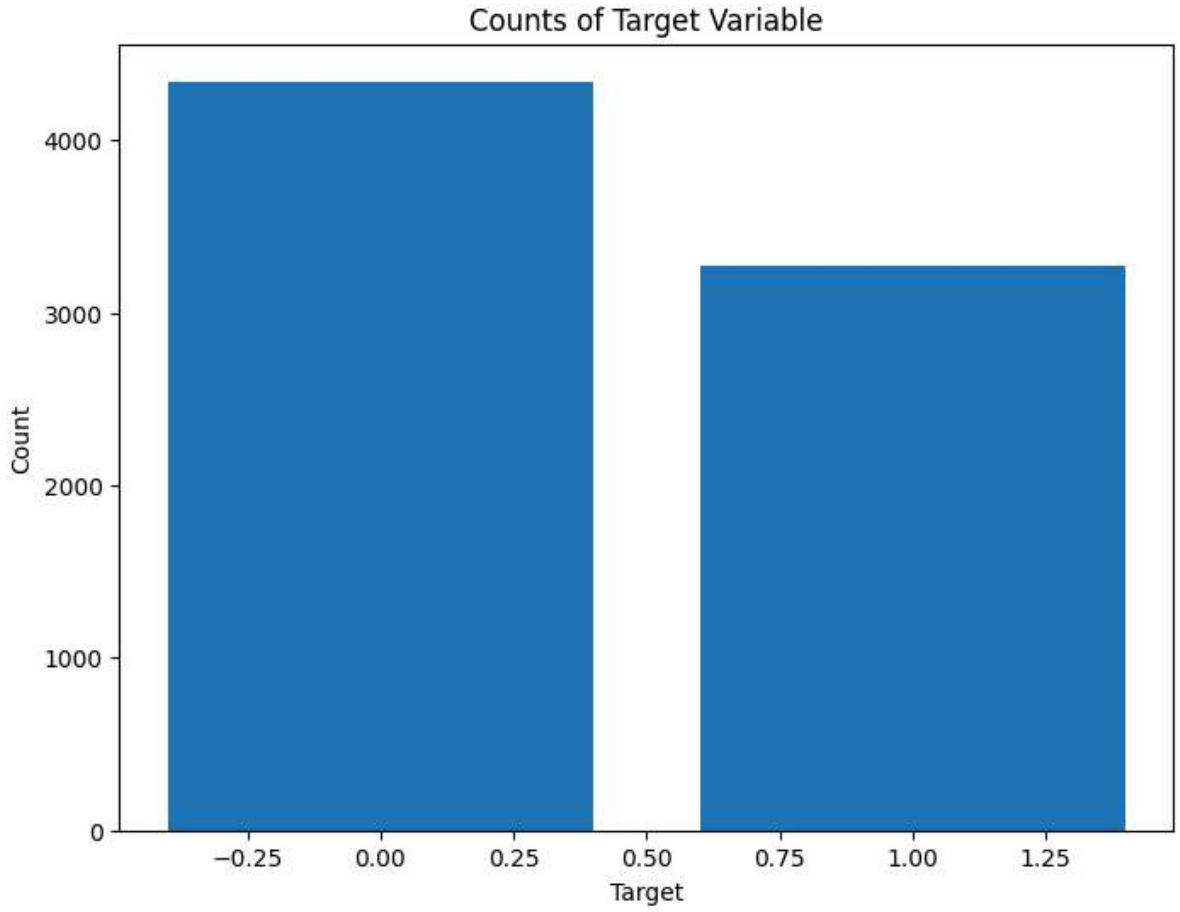
In [11]:
```
df.keyword.nunique()
```

Out[11]:
```
221
```

In [12]:
```
df.target.value_counts().sort_values(ascending=False)
```

Out[12]:
```
0    4342
1    3271
Name: target, dtype: int64
```

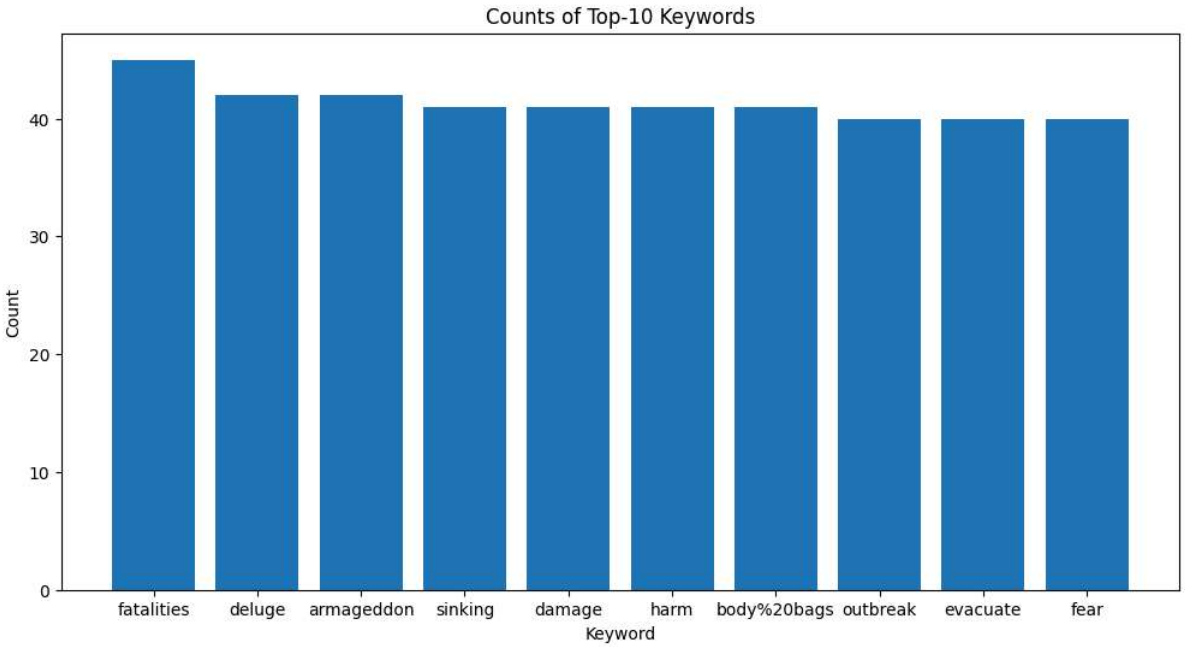
In [13]:
```
# Create bar plot
plt.figure(figsize=(8, 6))
plt.bar(df.target.value_counts().index, df.target.value_counts())

# Set plot title and labels
plt.title('Counts of Target Variable')
plt.xlabel('Target')
plt.ylabel('Count')

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




In [14]:
```
df.keyword.value_counts().sort_values(ascending=False)
```

Out[14]:
```
fatalities              45
armageddon              42
deluge                  42
sinking                 41
damage                  41
                        ..
forest%20fire           19
epicentre               12
threat                  11
inundation              10
radiation%20emergency    9
Name: keyword, Length: 221, dtype: int64
```

In [15]:
```
# Create bar plot
plt.figure(figsize=(12, 6))
plt.bar(df.keyword.value_counts().index[:10], df.keyword.value_counts()[:10])

# Set plot title and labels
plt.title('Counts of Top-10 Keywords')
plt.xlabel('Keyword')
plt.ylabel('Count')

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




The code has calculated the text length statistics for training and test datasets, showing the average text length is around 101 characters for both. It also tokenizes texts from both datasets, identifying 22,700 unique tokens in the training set and 12,818 in the test set, indicating a rich vocabulary used across the texts.

# 3. Preprocessing

In [16]:
```
# To ensure reproducibility in the experiments, we fix the random seed for all comp
seed_value=1

# 1. Set the `PYTHONHASHSEED` environment variable at a fixed value
os.environ['PYTHONHASHSEED']=str(seed_value)

# 2. Set Python built-in pseudorandom generator at a fixed value
random.seed(seed_value)

# 3. Set NumPy pseudorandom generator at a fixed value
np.random.seed(seed_value)

# 4. Set TensorFlow pseudorandom generator at a fixed value
tf.random.set_seed(seed_value)
```

In [17]:
```
# Function to lowercase the text and remove punctuation
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    return tf.strings.regex_replace(lowercase, '[{}]'.format(re.escape(string.punct
```

In [18]:
```
# Define the vocabulary size and the number of words in a sequence.
vocab_size = 4096
sequence_length = 10

# Vectorize layer
vectorize_layer = tf.keras.layers.TextVectorization(
    standardize=custom_standardization,
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length)
```

In [19]:
```
def generate_training_data(sequences, window_size, num_ns, vocab_size, seed):
    targets, contexts, labels = [], [], []

    sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(vocab_size

    for sequence in tf.data.Dataset.from_tensor_slices(sequences).as_numpy_iterator
        positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
            sequence,
            vocabulary_size=vocab_size,
            sampling_table=sampling_table,
            window_size=window_size,
            negative_samples=0)

        for target_word, context_word in positive_skip_grams:
            context_class = tf.expand_dims(tf.constant([context_word], dtype="int64
            negative_sampling_candidates, _, _ = tf.random.log_uniform_candidate_sa
                true_classes=context_class,
                num_true=1,
                num_sampled=num_ns,
                unique=True,
                range_max=vocab_size,
                seed=seed,
                name="negative_sampling")
            context = tf.concat([tf.squeeze(context_class, 1), negative_sampling_ca
            label = tf.constant([1] + [0] * num_ns, dtype="int64")

            targets.append(target_word)
            contexts.append(context)
            labels.append(label)

    return targets, contexts, labels
```

In [21]:
```
from sklearn.preprocessing import OneHotEncoder

max_length = 60  # Maximum length of text sequences
# One-hot encode the 'keyword' and 'location' features
onehot_encoder = OneHotEncoder(sparse=False)
keywords_encoded = onehot_encoder.fit_transform(df[['keyword']])
locations_encoded = onehot_encoder.fit_transform(df[['location']])

# Prepare the text data as before
X_text = df['text']
y = df['target']

# Tokenize the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_text)
X_text_sequences = tokenizer.texts_to_sequences(X_text)
X_text_padded = pad_sequences(X_text_sequences, maxlen=max_length, padding='post')

# Split data into train and validate sets
X_train_padded, X_validate_padded, y_train, y_validate, keywords_train, keywords_va
    X_text_padded, y, keywords_encoded, locations_encoded, test_size=0.2, random_st

# Ensure the y data is in the right shape
if isinstance(y_train, pd.Series):
    y_train = y_train.values.reshape(-1, 1)
if isinstance(y_validate, pd.Series):
    y_validate = y_validate.values.reshape(-1, 1)

# Concatenate the text data with the encoded keyword and location features
X_train = np.concatenate([X_train_padded, keywords_train, locations_train], axis=1)
X_validate = np.concatenate([X_validate_padded, keywords_validate, locations_valida
```

```
print("Train dataset size: ", X_train.shape[0])
print("Validate dataset size: ", X_validate.shape[0])
print('Max length (text sequences + keyword + location): ', X_train.shape[1])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: Fu
tureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be re
moved in 1.4. `sparse_output` is ignored unless you leave `sparse` to its default
value.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: Fu
tureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be re
moved in 1.4. `sparse_output` is ignored unless you leave `sparse` to its default
value.
  warnings.warn(
Train dataset size:  6090
Validate dataset size:  1523
Max length (text sequences + keyword + location):  3624
```

In [22]:

```
def build_model(hp: HyperParameters):
    model = Sequential()
    model.add(Input(shape=(60,)))
    model.add(Embedding(input_dim=20000, output_dim=hp.Int('embedding_output_dim',
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_1', min_value=32, max_val
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_2', min_value=32, max_val
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_3', min_value=32, max_val
    model.add(Dropout(hp.Float('dropout_rate', min_value=0.0, max_value=0.5, step=0
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer=Adam(hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4, 1e-5
    return model

# Assuming you have defined and prepared X_train_padded, y_train, X_validate_padded

tuner = RandomSearch(build_model, objective='val_accuracy', max_trials=10, executio
tuner.search(x=X_train_padded, y=y_train, epochs=10, validation_data=(X_validate_pa

best_hps = tuner.get_best_hyperparameters()[0]

# Now, you should only instantiate, train, and evaluate your model once with the op
model = tuner.hypermodel.build(best_hps)
history = model.fit(X_train_padded, y_train, epochs=10, validation_data=(X_validate

val_acc_per_epoch = history.history['val_accuracy']
best_epoch = val_acc_per_epoch.index(max(val_acc_per_epoch)) + 1
print(f'Best epoch: {best_epoch}')

model.fit(X_train_padded, y_train, epochs=best_epoch)
val_loss, val_accuracy = model.evaluate(X_validate_padded, y_validate)

print(f"Validation accuracy: {val_accuracy}")
```

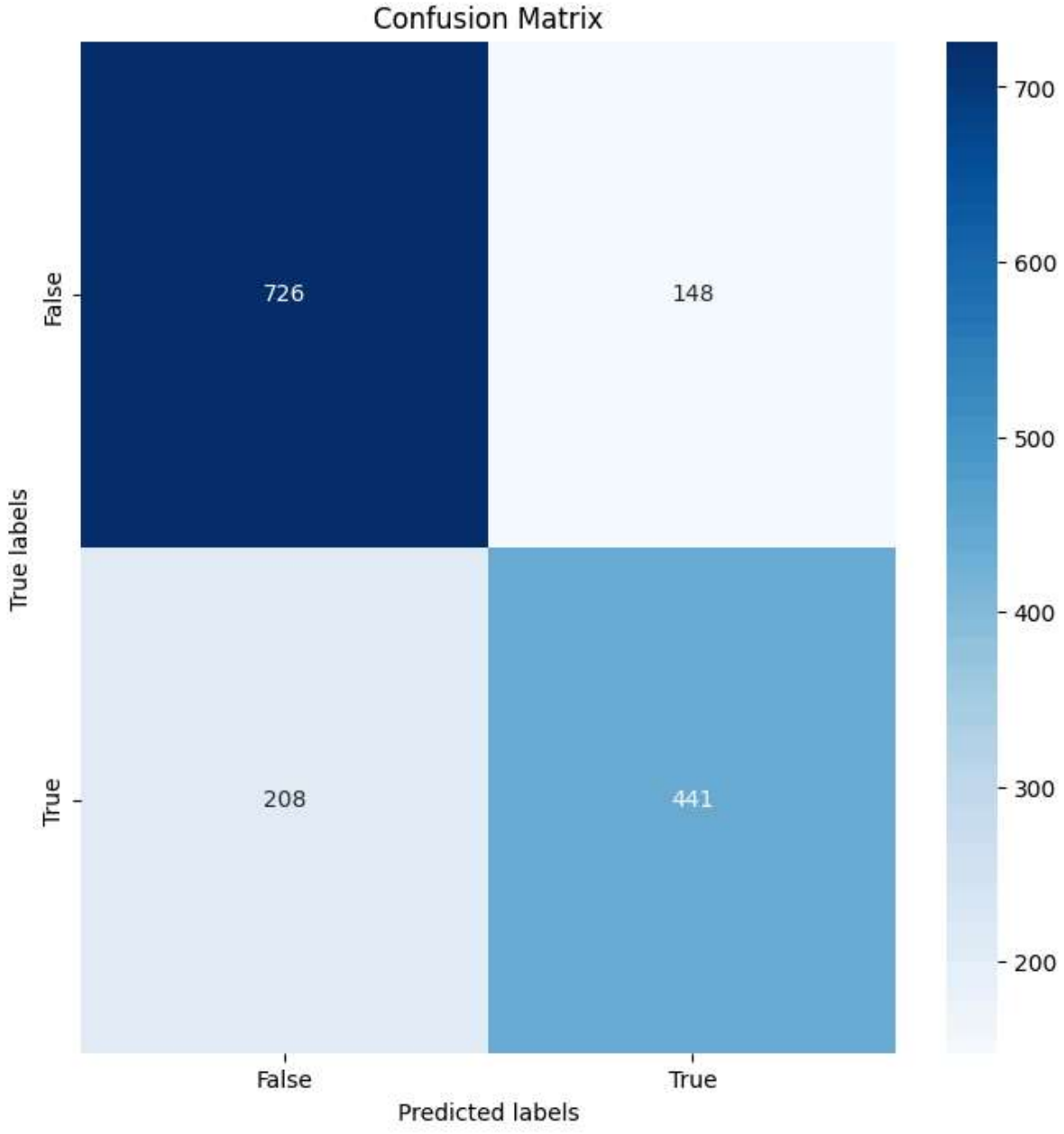
```
Reloading Tuner from my_dir/keras_tuning_advanced/tuner0.json
Epoch 1/10
191/191 [==============================] - 26s 85ms/step - loss: 0.7042 - accurac
y: 0.6425 - val_loss: 0.5557 - val_accuracy: 0.7827
Epoch 2/10
191/191 [==============================] - 7s 36ms/step - loss: 0.4274 - accuracy:
0.8493 - val_loss: 0.5192 - val_accuracy: 0.8030
Epoch 3/10
191/191 [==============================] - 4s 22ms/step - loss: 0.2765 - accuracy:
0.9192 - val_loss: 0.5561 - val_accuracy: 0.7774
Epoch 4/10
191/191 [==============================] - 4s 21ms/step - loss: 0.1891 - accuracy:
0.9532 - val_loss: 0.6107 - val_accuracy: 0.7794
Epoch 5/10
191/191 [==============================] - 4s 22ms/step - loss: 0.1422 - accuracy:
0.9683 - val_loss: 0.6388 - val_accuracy: 0.7774
Epoch 6/10
191/191 [==============================] - 4s 22ms/step - loss: 0.1107 - accuracy:
0.9800 - val_loss: 0.8467 - val_accuracy: 0.7754
Epoch 7/10
191/191 [==============================] - 4s 22ms/step - loss: 0.0924 - accuracy:
0.9856 - val_loss: 0.8216 - val_accuracy: 0.7551
Epoch 8/10
191/191 [==============================] - 4s 22ms/step - loss: 0.0901 - accuracy:
0.9844 - val_loss: 0.7507 - val_accuracy: 0.7525
Epoch 9/10
191/191 [==============================] - 4s 22ms/step - loss: 0.0753 - accuracy:
0.9888 - val_loss: 0.9180 - val_accuracy: 0.7649
Epoch 10/10
191/191 [==============================] - 4s 20ms/step - loss: 0.0730 - accuracy:
0.9883 - val_loss: 0.8930 - val_accuracy: 0.7715
Best epoch: 2
Epoch 1/2
191/191 [==============================] - 3s 17ms/step - loss: 0.0603 - accuracy:
0.9905
Epoch 2/2
191/191 [==============================] - 3s 17ms/step - loss: 0.0608 - accuracy:
0.9906
48/48 [==============================] - 0s 8ms/step - loss: 0.8793 - accuracy: 0.
7663
Validation accuracy: 0.7662508487701416
```

In [23]:

```python
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming you have your validation set as X_validate_padded and y_validate
# and that you have a trained model named 'model'

# Predict the classes for validation data
y_pred = model.predict(X_validate_padded)
y_pred = (y_pred > 0.5).astype(int)  # Convert probabilities to binary predictions

# Generate the confusion matrix
cm = confusion_matrix(y_validate, y_pred)

# Plot the confusion matrix using Seaborn
fig, ax = plt.subplots(figsize=(8, 8))
sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap="Blues")  # fmt='d' for integer fo
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')
ax.xaxis.set_ticklabels(['False', 'True'])
ax.yaxis.set_ticklabels(['False', 'True'])
```

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

```
48/48 [==============================] - 2s 7ms/step
```




In [27]:
```
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score,

# Assuming you have your predictions (y_pred) and true labels (y_validate)
y_pred_prob = model.predict(X_validate_padded)
precision = precision_score(y_validate, y_pred)
recall = recall_score(y_validate, y_pred)
f1 = f1_score(y_validate, y_pred)
roc_auc = roc_auc_score(y_validate, y_pred_prob)  # Assuming you have the probabili
mcc = matthews_corrcoef(y_validate, y_pred)
kappa = cohen_kappa_score(y_validate, y_pred)

print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
print(f'ROC-AUC: {roc_auc}')
print(f'MCC: {mcc}')
print(f'Cohen\'s Kappa: {kappa}')
```

```
48/48 [==============================] - 0s 9ms/step
Precision: 0.7487266553480475
Recall: 0.6795069337442219
F1 Score: 0.7124394184168014
ROC-AUC: 0.8226218473765307
MCC: 0.5180388893760524
Cohen's Kappa: 0.5163147039302307
```

In [25]:
```
df_test = pd.read_csv('test.csv')
df_test.head()
```

Out[25]:

| | 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 [26]:
```
# Preprocess df_test['text'] in the same way as the training data
# Tokenize the text
X_test = df_test['text']
X_test_sequences = tokenizer.texts_to_sequences(X_test)
# Pad the sequences
X_test_padded = pad_sequences(X_test_sequences, maxlen=max_length, padding='post')

# Predict using the model
test_predictions = model.predict(X_test_padded)
test_predictions = (test_predictions > 0.5).astype(int)  # Convert probabilities to

# Combine the original IDs with the predicted target
results_df = pd.DataFrame({'id': df_test['id'], 'target': test_predictions.flatten(

# Output the dataframe in the desired format
results_df.to_csv('submission_key_loc.csv', index=False)
```

```
102/102 [==============================] - 1s 7ms/step
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [1]:
```
!pip install keras-preprocessing
!pip install tensorflow.keras.layers
!pip install keras-tuner
```

```
Collecting keras-preprocessing
  Downloading Keras_Preprocessing-1.1.2-py2.py3-none-any.whl (42 kB)
     ---------------------------------------- 42.6/42.6 kB 2.0 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.9.1 in /usr/local/lib/python3.10/dist-pack
ages (from keras-preprocessing) (1.25.2)
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.10/dist-packag
es (from keras-preprocessing) (1.16.0)
Installing collected packages: keras-preprocessing
Successfully installed keras-preprocessing-1.1.2
ERROR: Could not find a version that satisfies the requirement tensorflow.keras.la
yers (from versions: none)
ERROR: No matching distribution found for tensorflow.keras.layers
Collecting keras-tuner
  Downloading keras_tuner-1.4.7-py3-none-any.whl (129 kB)
     ---------------------------------------- 129.1/129.1 kB 3.8 MB/s eta 0:00:00
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (f
rom keras-tuner) (2.15.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-package
s (from keras-tuner) (23.2)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages
(from keras-tuner) (2.31.0)
Collecting kt-legacy (from keras-tuner)
  Downloading kt_legacy-1.0.5-py3-none-any.whl (9.6 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.
10/dist-packages (from requests->keras-tuner) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pack
ages (from requests->keras-tuner) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dis
t-packages (from requests->keras-tuner) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dis
t-packages (from requests->keras-tuner) (2024.2.2)
Installing collected packages: kt-legacy, keras-tuner
Successfully installed keras-tuner-1.4.7 kt-legacy-1.0.5
```

# 1. Imports

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import regularizers
from kerastuner import HyperParameters
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
import keras
import sklearn
from tensorflow.keras.layers import Input, Bidirectional, Dropout, LSTM, Embedding,
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import tqdm
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from keras.models import Sequential
```

```
from keras_tuner.tuners import RandomSearch
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM
from sklearn.model_selection import GridSearchCV
from keras.optimizers import Adam
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
import io
import re
import os
import random
import string
import tqdm
from keras.utils import to_categorical
```

```
<ipython-input-2-519cd75d6544>:9: DeprecationWarning: `import kerastuner` is depre
cated, please use `import keras_tuner`.
  from kerastuner import HyperParameters
```

In [3]:
```
from google.colab import drive

drive.mount('/content/drive')

# Read datafile
%cd /content/drive/My Drive/Colab Notebooks/
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks
```

In [4]:
```
df = pd.read_csv('train.csv')
```

In [5]:
```
df.shape
```

Out[5]:
```
(7613, 5)
```

In [6]:
```
df.dtypes
```

Out[6]:
```
id           int64
keyword     object
location    object
text        object
target       int64
dtype: object
```

# 2. Data Exploration

In [7]:
```
df.head()
```

Out[7]:

| | 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 [8]:
```
df.isna().sum()
```

Out[8]:
```
id             0
keyword       61
location    2533
text           0
target         0
dtype: int64
```

In [9]:
```
df.duplicated().sum()
```

Out[9]:
```
0
```

In [10]:
```
df.describe()
```

Out[10]:

| | id | target |
|---|---|---|
| **count** | 7613.000000 | 7613.00000 |
| **mean** | 5441.934848 | 0.42966 |
| **std** | 3137.116090 | 0.49506 |
| **min** | 1.000000 | 0.00000 |
| **25%** | 2734.000000 | 0.00000 |
| **50%** | 5408.000000 | 0.00000 |
| **75%** | 8146.000000 | 1.00000 |
| **max** | 10873.000000 | 1.00000 |

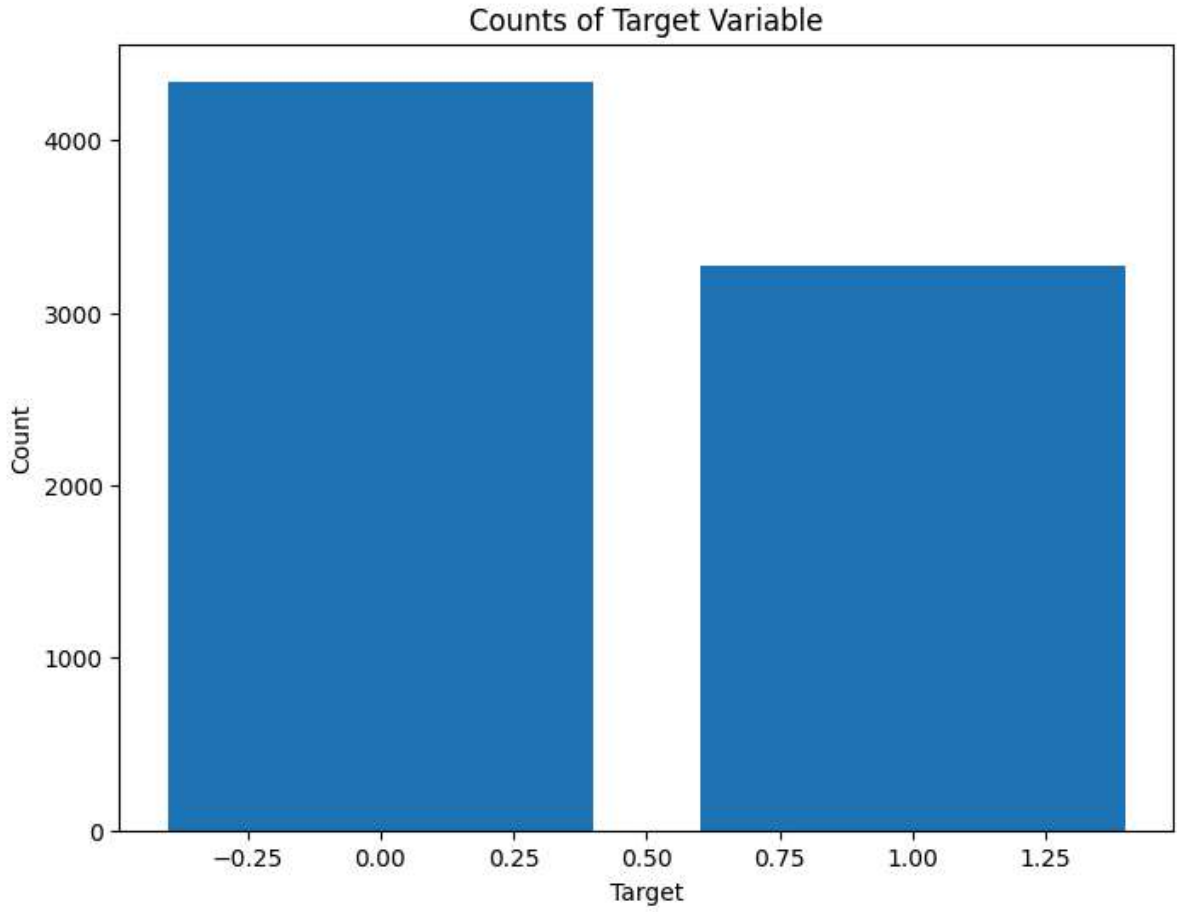
In [11]:
```
df.keyword.nunique()
```

Out[11]:
```
221
```

In [12]:
```
df.target.value_counts().sort_values(ascending=False)
```

Out[12]:
```
0    4342
1    3271
Name: target, dtype: int64
```

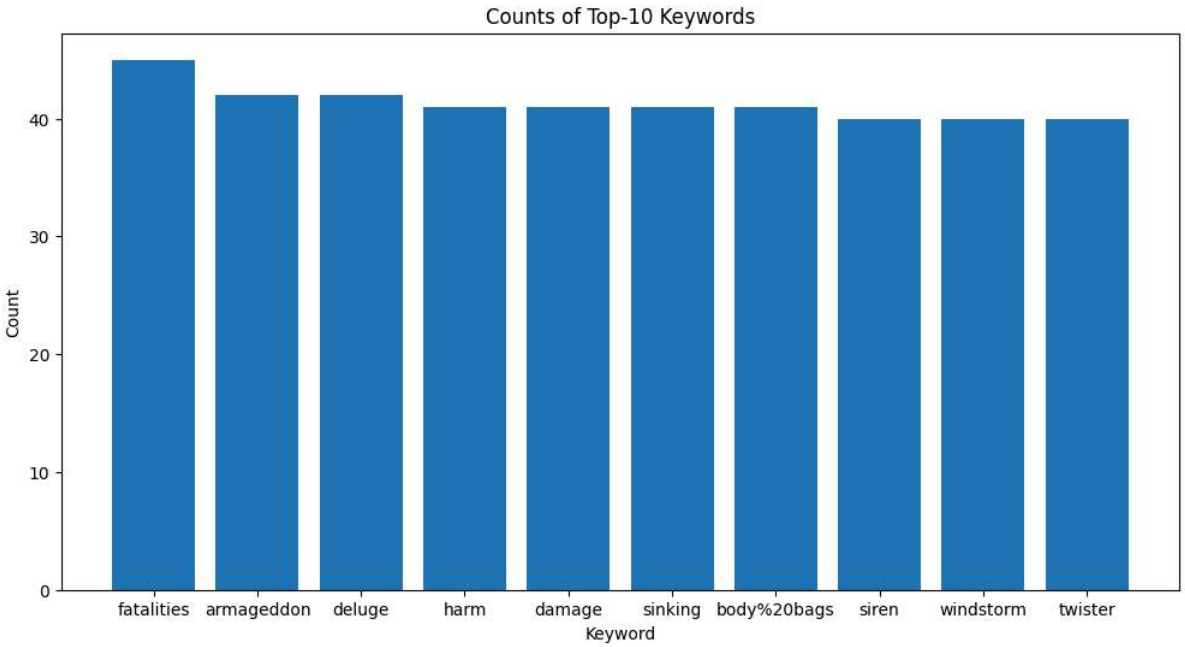
In [13]:
```
# Create bar plot
plt.figure(figsize=(8, 6))
plt.bar(df.target.value_counts().index, df.target.value_counts())

# Set plot title and labels
plt.title('Counts of Target Variable')
plt.xlabel('Target')
plt.ylabel('Count')

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




In [14]:
```
df.keyword.value_counts().sort_values(ascending=False)
```

Out[14]:
```
fatalities              45
armageddon              42
deluge                  42
harm                    41
damage                  41
                        ..
forest%20fire           19
epicentre               12
threat                  11
inundation              10
radiation%20emergency    9
Name: keyword, Length: 221, dtype: int64
```

In [15]:
```
# Create bar plot
plt.figure(figsize=(12, 6))
plt.bar(df.keyword.value_counts().index[:10], df.keyword.value_counts()[:10])

# Set plot title and labels
plt.title('Counts of Top-10 Keywords')
plt.xlabel('Keyword')
plt.ylabel('Count')

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




The code has calculated the text length statistics for training and test datasets, showing the average text length is around 101 characters for both. It also tokenizes texts from both datasets, identifying 22,700 unique tokens in the training set and 12,818 in the test set, indicating a rich vocabulary used across the texts.

# 3. Preprocessing

In [16]:
```
# To ensure reproducibility in the experiments, we fix the random seed for all comp
seed_value=1

# 1. Set the `PYTHONHASHSEED` environment variable at a fixed value
os.environ['PYTHONHASHSEED']=str(seed_value)

# 2. Set Python built-in pseudorandom generator at a fixed value
random.seed(seed_value)

# 3. Set NumPy pseudorandom generator at a fixed value
np.random.seed(seed_value)

# 4. Set TensorFlow pseudorandom generator at a fixed value
tf.random.set_seed(seed_value)
```

In [17]:
```
# Function to lowercase the text and remove punctuation
def custom_standardization(input_data):
    lowercase = tf.strings.lower(input_data)
    return tf.strings.regex_replace(lowercase, '[{}]'.format(re.escape(string.punct
```

In [18]:
```
# Define the vocabulary size and the number of words in a sequence.
vocab_size = 4096
sequence_length = 10

# Vectorize layer
vectorize_layer = tf.keras.layers.TextVectorization(
    standardize=custom_standardization,
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length)
```

In [19]:
```
def generate_training_data(sequences, window_size, num_ns, vocab_size, seed):
    targets, contexts, labels = [], [], []

    sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(vocab_size

    for sequence in tf.data.Dataset.from_tensor_slices(sequences).as_numpy_iterator
        positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
            sequence,
            vocabulary_size=vocab_size,
            sampling_table=sampling_table,
            window_size=window_size,
            negative_samples=0)

        for target_word, context_word in positive_skip_grams:
            context_class = tf.expand_dims(tf.constant([context_word], dtype="int64
            negative_sampling_candidates, _, _ = tf.random.log_uniform_candidate_sa
                true_classes=context_class,
                num_true=1,
                num_sampled=num_ns,
                unique=True,
                range_max=vocab_size,
                seed=seed,
                name="negative_sampling")
            context = tf.concat([tf.squeeze(context_class, 1), negative_sampling_ca
            label = tf.constant([1] + [0] * num_ns, dtype="int64")

            targets.append(target_word)
            contexts.append(context)
            labels.append(label)

    return targets, contexts, labels
```

In [21]:
```
from sklearn.preprocessing import OneHotEncoder

# Assuming df['keyword'] contains categorical data that you want to include

max_length = 60  # Maximum length of the text sequences
num_ns = 4  # Number of negative samples per positive sample

# One-hot encode the 'keyword' feature
onehot_encoder = OneHotEncoder(sparse=False)
keywords_encoded = onehot_encoder.fit_transform(df[['keyword']])

# Prepare the text data as before
X_text = df['text']
y = df['target']

# Tokenize the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_text)
X_text_sequences = tokenizer.texts_to_sequences(X_text)
X_text_padded = pad_sequences(X_text_sequences, maxlen=max_length, padding='post')

# Split data into train and validate sets for both text and keyword
X_train_padded, X_validate_padded, y_train, y_validate, keywords_train, keywords_va
    X_text_padded, y, keywords_encoded, test_size=0.2, random_state=42)

# Ensure the y data is in the right shape
if isinstance(y_train, pd.Series):
    y_train = y_train.values.reshape(-1, 1)
if isinstance(y_validate, pd.Series):
    y_validate = y_validate.values.reshape(-1, 1)
```

```
# Concatenate the text data with the encoded keyword features
X_train = np.concatenate([X_train_padded, keywords_train], axis=1)
X_validate = np.concatenate([X_validate_padded, keywords_validate], axis=1)

print("Train dataset size: ", X_train.shape[0])
print("Validate dataset size: ", X_validate.shape[0])
print('Max length (text sequences + keyword): ', X_train.shape[1])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: Fu
tureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be re
moved in 1.4. `sparse_output` is ignored unless you leave `sparse` to its default
value.
  warnings.warn(
Train dataset size:  6090
Validate dataset size:  1523
Max length (text sequences + keyword):  282
```

In [22]:

```
def build_model(hp: HyperParameters):
    model = Sequential()
    model.add(Input(shape=(60,)))
    model.add(Embedding(input_dim=20000, output_dim=hp.Int('embedding_output_dim',
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_1', min_value=32, max_val
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_2', min_value=32, max_val
    model.add(Bidirectional(LSTM(units=hp.Int('lstm_units_3', min_value=32, max_val
    model.add(Dropout(hp.Float('dropout_rate', min_value=0.0, max_value=0.5, step=0
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer=Adam(hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4, 1e-5
    return model

# Assuming you have defined and prepared X_train_padded, y_train, X_validate_padded

tuner = RandomSearch(build_model, objective='val_accuracy', max_trials=10, executio
tuner.search(x=X_train_padded, y=y_train, epochs=10, validation_data=(X_validate_pa

best_hps = tuner.get_best_hyperparameters()[0]

# Now, you should only instantiate, train, and evaluate your model once with the op
model = tuner.hypermodel.build(best_hps)
history = model.fit(X_train_padded, y_train, epochs=10, validation_data=(X_validate

val_acc_per_epoch = history.history['val_accuracy']
best_epoch = val_acc_per_epoch.index(max(val_acc_per_epoch)) + 1
print(f'Best epoch: {best_epoch}')

model.fit(X_train_padded, y_train, epochs=best_epoch)
val_loss, val_accuracy = model.evaluate(X_validate_padded, y_validate)

print(f"Validation accuracy: {val_accuracy}")
```

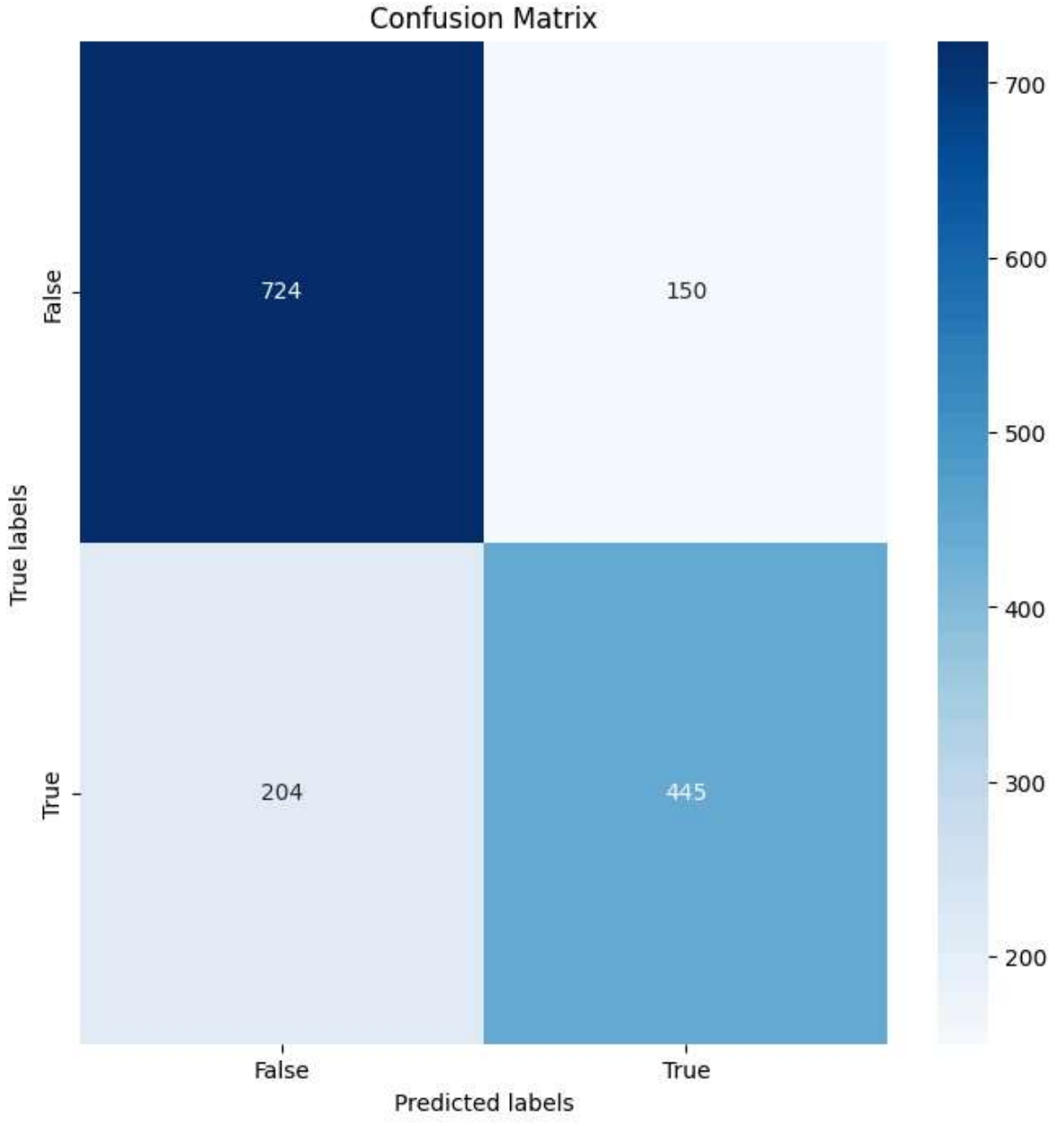
```
Reloading Tuner from my_dir/keras_tuning_advanced/tuner0.json
Epoch 1/10
191/191 [==============================] - 30s 95ms/step - loss: 0.7042 - accurac
y: 0.6425 - val_loss: 0.5557 - val_accuracy: 0.7827
Epoch 2/10
191/191 [==============================] - 8s 41ms/step - loss: 0.4274 - accuracy:
0.8493 - val_loss: 0.5192 - val_accuracy: 0.8030
Epoch 3/10
191/191 [==============================] - 5s 26ms/step - loss: 0.2765 - accuracy:
0.9192 - val_loss: 0.5561 - val_accuracy: 0.7774
Epoch 4/10
191/191 [==============================] - 5s 25ms/step - loss: 0.1891 - accuracy:
0.9532 - val_loss: 0.6107 - val_accuracy: 0.7794
Epoch 5/10
191/191 [==============================] - 4s 23ms/step - loss: 0.1422 - accuracy:
0.9683 - val_loss: 0.6388 - val_accuracy: 0.7774
Epoch 6/10
191/191 [==============================] - 5s 25ms/step - loss: 0.1107 - accuracy:
0.9800 - val_loss: 0.8467 - val_accuracy: 0.7754
Epoch 7/10
191/191 [==============================] - 5s 24ms/step - loss: 0.0924 - accuracy:
0.9856 - val_loss: 0.8216 - val_accuracy: 0.7551
Epoch 8/10
191/191 [==============================] - 5s 24ms/step - loss: 0.0901 - accuracy:
0.9844 - val_loss: 0.7502 - val_accuracy: 0.7525
Epoch 9/10
191/191 [==============================] - 5s 26ms/step - loss: 0.0752 - accuracy:
0.9888 - val_loss: 0.9181 - val_accuracy: 0.7649
Epoch 10/10
191/191 [==============================] - 4s 23ms/step - loss: 0.0730 - accuracy:
0.9883 - val_loss: 0.8904 - val_accuracy: 0.7715
Best epoch: 2
Epoch 1/2
191/191 [==============================] - 4s 19ms/step - loss: 0.0602 - accuracy:
0.9905
Epoch 2/2
191/191 [==============================] - 4s 20ms/step - loss: 0.0607 - accuracy:
0.9908
48/48 [==============================] - 0s 9ms/step - loss: 0.9009 - accuracy: 0.
7676
Validation accuracy: 0.7675639986991882
```

In [23]:

```python
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming you have your validation set as X_validate_padded and y_validate
# and that you have a trained model named 'model'

# Predict the classes for validation data
y_pred = model.predict(X_validate_padded)
y_pred = (y_pred > 0.5).astype(int)  # Convert probabilities to binary predictions

# Generate the confusion matrix
cm = confusion_matrix(y_validate, y_pred)

# Plot the confusion matrix using Seaborn
fig, ax = plt.subplots(figsize=(8, 8))
sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap="Blues")  # fmt='d' for integer fo
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')
ax.xaxis.set_ticklabels(['False', 'True'])
ax.yaxis.set_ticklabels(['False', 'True'])
```

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

```
48/48 [==============================] - 2s 7ms/step
```




In [26]:
```
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score,

# Assuming you have your predictions (y_pred) and true labels (y_validate)
y_pred_prob = model.predict(X_validate_padded)
precision = precision_score(y_validate, y_pred)
recall = recall_score(y_validate, y_pred)
f1 = f1_score(y_validate, y_pred)
roc_auc = roc_auc_score(y_validate, y_pred_prob)  # Assuming you have the probabili
mcc = matthews_corrcoef(y_validate, y_pred)
kappa = cohen_kappa_score(y_validate, y_pred)

print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
print(f'ROC-AUC: {roc_auc}')
print(f'MCC: {mcc}')
print(f'Cohen\'s Kappa: {kappa}')
```

```
48/48 [==============================] - 0s 8ms/step
Precision: 0.7478991596638656
Recall: 0.6856702619414484
F1 Score: 0.7154340836012861
ROC-AUC: 0.8227055882487756
MCC: 0.5210113639372386
Cohen's Kappa: 0.5196105860989289
```

In [24]:
```
df_test = pd.read_csv('test.csv')
df_test.head()
```

Out[24]:

| | 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 [25]:
```
# Preprocess df_test['text'] in the same way as the training data
# Tokenize the text
X_test = df_test['text']
X_test_sequences = tokenizer.texts_to_sequences(X_test)
# Pad the sequences
X_test_padded = pad_sequences(X_test_sequences, maxlen=max_length, padding='post')

# Predict using the model
test_predictions = model.predict(X_test_padded)
test_predictions = (test_predictions > 0.5).astype(int)  # Convert probabilities to

# Combine the original IDs with the predicted target
results_df = pd.DataFrame({'id': df_test['id'], 'target': test_predictions.flatten(

# Output the dataframe in the desired format
results_df.to_csv('submission_key.csv', index=False)
```

```
102/102 [==============================] - 1s 8ms/step
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: