Disaster Tweet Classification with RNN (LSTM/GRU)

This project is **Natural Language Processing with Disaster Tweets** using recurrent neural networks. The objective is to predict whether a tweet refers to an actual disaster. The official training set contains **7613** tweets, each with an identifier (`id`), optional `keyword` and `location` , the `text` of the tweet, and a binary `target` indicating whether it describes a disaster (1) or not (0). The test set contains **3263** tweets without `target` labels. The dataset is drawn from Kaggle's *NLP with Disaster Tweets* competition.

Kaggle reference (APA format):

> Kaggle. (2020). *Natural Language Processing with Disaster Tweets* [Data set]. Kaggle. https://www.kaggle.com/competitions/nlp-getting-started/data

In [1]:
```
import os
import pandas as pd
import numpy as np
from wordcloud import WordCloud
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, GRU, Bidirectional, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
import keras_tuner as kt
```

```
2025-10-31 18:51:39.265993: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due t
o floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-10-31 18:51:40.738620: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions
in performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2025-10-31 18:51:44.989700: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due t
o floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
```

Data Summary

- In the Kaggle dataset, there are **10,876** tweets with five columns: `id` , `keyword` , `location` , `text` , and `target` .
- The dataset is already split into a train and test set. The training data has 7,613 rows and the testing data has 3,263 rows.
- The `target` column is `1` for tweets describing disasters and `0` for non-disaster tweets.
- The dataset includes a balanced mix of disaster and non-disaster tweets (3,271 disaster tweets vs 4,342 non-disaster tweets).
- Missing values occur in the `keyword` and `location` columns.

In [2]:
```
# Load train and test datasets
train_df = pd.read_csv('train.csv')
test_df = pd.read_csv('test.csv')

# Concatenate to create full dataset
df = pd.concat([train_df, test_df])
df
```

Out[2]:

	id	keyword	location	text	target
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1.0
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1.0
2	5	NaN	NaN	All residents asked to 'shelter in place' are ...	1.0
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1.0
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1.0
...	...	...	...	...	...
3258	10861	NaN	NaN	EARTHQUAKE SAFETY LOS ANGELES ÛÒ SAFETY FASTE...	NaN
3259	10865	NaN	NaN	Storm in RI worse than last hurricane. My city...	NaN
3260	10868	NaN	NaN	Green Line derailment in Chicago http://t.co/U...	NaN
3261	10874	NaN	NaN	MEG issues Hazardous Weather Outlook (HWO) htt...	NaN
3262	10875	NaN	NaN	#CityofCalgary has activated its Municipal Eme...	NaN

10876 rows × 5 columns

In [3]:
```
# Basic information about the dataset
print('Number of rows:', df.shape[0])
print('Number of columns:', df.shape[1])
print('\nData types:')
print(df.dtypes)
```

```
Number of rows: 10876
Number of columns: 5

Data types:
id            int64
keyword      object
location     object
text         object
target      float64
dtype: object
```

Exploratory Data Analysis (EDA)

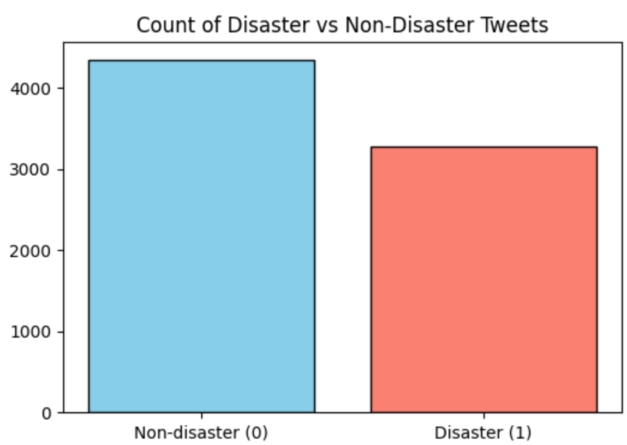
I first explore the training dataset to understand its size, structure, and class distribution. Understanding class imbalance is important when evaluating model performance. I also check for missing values in the `keyword` and `location` fields and examine tweet lengths.

Target Class Distribution

In [4]:
```
# Target class distribution
print('Target class distribution (0=non-disaster, 1=disaster):')
print(df['target'].value_counts(), '\n')

# Plot distribution
counts = df['target'].value_counts().sort_index()  # ensure order 0 then 1
plt.figure(figsize=(6, 4))
bars = plt.bar(
    ['Non-disaster (0)', 'Disaster (1)'],
    counts.values,
    color=['skyblue', 'salmon'],
    edgecolor='black'
)
plt.title('Count of Disaster vs Non-Disaster Tweets')
plt.xticks(rotation=0)
plt.show()
```

```
Target class distribution (0=non-disaster, 1=disaster):
target
0.0    4342
1.0    3271
Name: count, dtype: int64
```



> The target shows some class imbalance:
>
> - Non-Disaster = 57% of target values
> - Disaster = 43% of target values

MIssing Values

In [5]:
```
print('Missing values per column:')
print(df.isnull().sum(), '\n')
```

```
Missing values per column:
id             0
keyword       87
location    3638
text           0
target      3263
dtype: int64
```

> Note: the `target` column has 3263 missing values because `test_df` doesn't include target values.

Word Count Distributions

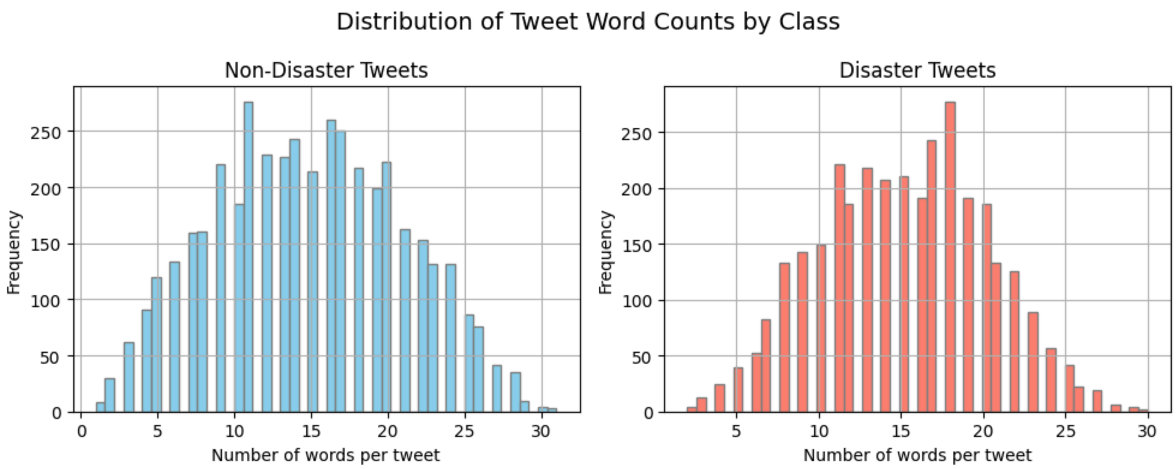
In [6]:
```python
# Add tweet length column
train_df['length'] = train_df['text'].apply(lambda x: len(x.split()))

# Create side-by-side histograms
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
train_df[train_df['target'] == 0]['length'].hist(bins=50, color='skyblue', edgecolor='grey')
plt.title('Non-Disaster Tweets')
plt.xlabel('Number of words per tweet')
plt.ylabel('Frequency')

plt.subplot(1, 2, 2)
train_df[train_df['target'] == 1]['length'].hist(bins=50, color='salmon', edgecolor='grey')
plt.title('Disaster Tweets')
plt.xlabel('Number of words per tweet')
plt.ylabel('Frequency')

plt.suptitle('Distribution of Tweet Word Counts by Class', fontsize=14)
plt.tight_layout()
plt.show()
```



> Both the Non-Disaster and Disaster tweets appear to be somewhat normally distributed.

Frequent Keywords

In [7]:
```python
# Replace '%20' with spaces
df['keyword'] = df['keyword'].str.replace('%20', ' ')

# Drop missing keywords and combine into a single string
keywords_text = ' '.join(df['keyword'].dropna())

# Generate word cloud
wordcloud = WordCloud(
    width=1200,
    height=400,
    background_color='white',
    colormap='inferno',
    max_words=1000,
    collocations=False
).generate(keywords_text)

# Display the word cloud
plt.figure(figsize=(15, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

RNN Model Architecture

I'm testing two neural network architectures:

1. **LSTM** model: Embedding → LSTM → Dense
2. **Bidirectional GRU** model: Embedding → Bidirectional(GRU) → Dense

Both models use an Embedding layer to learn word vectors during training. The models are compiled with the binary cross-entropy loss and evaluated using accuracy and F1-score. A validation split from the training data is used to monitor performance and prevent overfitting via early stopping.

Text Preprocessing

I preprocess the text by cleaning and tokenizing. Using `tf.keras.preprocessing.text.Tokenizer`, I convert each tweet into a sequence of integer indices, keeping only the most frequent words. Sequences are padded to a uniform length using `tf.keras.preprocessing.sequence.pad_sequences`.

In [8]:

```
# Convert text column
train_df['text'] = train_df['text'].astype(str)
test_df['text'] = test_df['text'].astype(str)

# Tokenization
max_words = 20000  # vocabulary size
max_len = 30       # maximum sequence length

# Fit tokenizer on training text
tokenizer = Tokenizer(num_words=max_words, oov_token='<OOV>')
tokenizer.fit_on_texts(train_df['text'])

# Convert text to sequences
X = tokenizer.texts_to_sequences(train_df['text'])
X_test_seq = tokenizer.texts_to_sequences(test_df['text'])

# Pad sequences
X = pad_sequences(X, maxlen=max_len, padding='post', truncating='post')
X_test_seq = pad_sequences(X_test_seq, maxlen=max_len, padding='post', truncating='post')

y = train_df['target'].values

# Split into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# Define model architectures
embedding_dim = 64

# Set random seed
tf.random.set_seed(666)
```

LSTM Model

In [9]:

```
# LSTM model
lstm_model = Sequential([
    Embedding(max_words, embedding_dim),
    LSTM(64, dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])
lstm_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
lstm_model.build((None, max_len))
lstm_model.summary()
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1761958352.472577    2332 gpu_device.cc:2020] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 1756 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 3050 Ti Laptop GPU, pci bus id: 0000:01:00.0, compute capability: 8.6
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 30, 64)	1,280,000
lstm (LSTM)	(None, 64)	33,024
dense (Dense)	(None, 1)	65

Total params: 1,313,089 (5.01 MB)

Trainable params: 1,313,089 (5.01 MB)

Non-trainable params: 0 (0.00 B)

In [10]:
```
# Fit LSTM model
lstm_history = lstm_model.fit(
    X_train,
    y_train,
    epochs=5,
    batch_size=64,
    validation_data=(X_val, y_val),
    callbacks=EarlyStopping(monitor='val_loss', patience=2, restore_best_weights=True)
)

# Evaluate on validation set
lstm_val_pred = (lstm_model.predict(X_val) > 0.5).astype(int)
lstm_accuracy = accuracy_score(y_val, lstm_val_pred)
lstm_f1 = f1_score(y_val, lstm_val_pred)
print(f'LSTM Validation Accuracy: {lstm_accuracy:.4f}')
print(f'LSTM Validation F1: {lstm_f1:.4f}')
```

```
Epoch 1/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 43s 363ms/step - accuracy: 0.6759 - loss: 0.5968 - val_accuracy: 0.7800 - val_loss: 0.4854
Epoch 2/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 34s 348ms/step - accuracy: 0.8504 - loss: 0.3779 - val_accuracy: 0.8011 - val_loss: 0.4564
Epoch 3/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 35s 359ms/step - accuracy: 0.9123 - loss: 0.2558 - val_accuracy: 0.7807 - val_loss: 0.5363
Epoch 4/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 34s 352ms/step - accuracy: 0.9396 - loss: 0.1952 - val_accuracy: 0.7840 - val_loss: 0.5215
48/48 ━━━━━━━━━━━━━━━━━━━━ 4s 69ms/step
LSTM Validation Accuracy: 0.8011
LSTM Validation F1: 0.7434
```

Bidirectional GRU Model

In [11]:
```
# Bidirectional GRU model
gru_model = Sequential([
    Embedding(max_words, embedding_dim),
    Bidirectional(GRU(64, dropout=0.2, recurrent_dropout=0.2)),
    Dense(1, activation='sigmoid')
])
gru_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
gru_model.build((None, max_len))
gru_model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 30, 64)	1,280,000
bidirectional (Bidirectional)	(None, 128)	49,920
dense_1 (Dense)	(None, 1)	129

Total params: 1,330,049 (5.07 MB)

Trainable params: 1,330,049 (5.07 MB)

Non-trainable params: 0 (0.00 B)

In [12]:
```
# Fit GRU model
gru_history = gru_model.fit(
    X_train,
    y_train,
    epochs=5,
    batch_size=64,
    validation_data=(X_val, y_val),
    callbacks=EarlyStopping(monitor='val_loss', patience=2, restore_best_weights=True)
)

# Evaluate GRU
gru_val_pred = (gru_model.predict(X_val) > 0.5).astype(int)
gru_accuracy = accuracy_score(y_val, gru_val_pred)
gru_f1 = f1_score(y_val, gru_val_pred)
print(f'Bidirectional GRU Validation Accuracy: {gru_accuracy:.4f}')
print(f'Bidirectional GRU Validation F1: {gru_f1:.4f}')
```

```
Epoch 1/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 92s 888ms/step - accuracy: 0.6765 - loss: 0.5957 - val_accuracy: 0.7781 - val_loss: 0.4769
Epoch 2/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 87s 907ms/step - accuracy: 0.8502 - loss: 0.3489 - val_accuracy: 0.7827 - val_loss: 0.4801
Epoch 3/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 85s 882ms/step - accuracy: 0.9159 - loss: 0.2198 - val_accuracy: 0.7978 - val_loss: 0.5123
48/48 ━━━━━━━━━━━━━━━━━━━━ 10s 180ms/step
Bidirectional GRU Validation Accuracy: 0.7781
Bidirectional GRU Validation F1: 0.7455
```

LSTM Model (Hyperparameter Tuned)

In [13]:
```
def build_lstm_model(hp):
    model = Sequential()
    model.add(Embedding(max_words, hp.Int('embedding_dim', min_value=32, max_value=128, step=32)))
    model.add(LSTM(
        hp.Int('lstm_units', min_value=32, max_value=128, step=32),
        dropout=hp.Float('dropout', min_value=0.1, max_value=0.5, step=0.1),
        recurrent_dropout=hp.Float('recurrent_dropout',min_value=0.1, max_value=0.5, step=0.1)
    ))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

# Instantiate the tuner for LSTM
lstm_tuner = kt.Hyperband(
    build_lstm_model,
    objective='val_accuracy',
    max_epochs=5,
    factor=3,
    directory='my_dir',
    project_name='lstm_tuning'
)

# Run the search for LSTM
print("Running KerasTuner search for LSTM model...")
lstm_tuner.search(X_train, y_train, validation_data=(X_val, y_val), epochs=5, callbacks=[EarlyStopping(monitor='val_loss', patience=2)])

# Get the best LSTM model
best_lstm_model = lstm_tuner.get_best_models(num_models=1)[0]

# Evaluate the best LSTM model
lstm_val_pred = (best_lstm_model.predict(X_val) > 0.5).astype(int)
lstm_accuracy = accuracy_score(y_val, lstm_val_pred)
lstm_f1 = f1_score(y_val, lstm_val_pred)
print(f'Best Hyperparameter Tuned LSTM Validation Accuracy: {lstm_accuracy:.4f}')
print(f'Best Hyperparameter Tuned LSTM Validation F1: {lstm_f1:.4f}')
```

```
Trial 10 Complete [00h 04m 40s]
val_accuracy: 0.8076165318489075

Best val_accuracy So Far: 0.810899555683136
Total elapsed time: 00h 35m 10s
48/48 ━━━━━━━━━━━━━━━━━━━━ 4s 70ms/step
Best Hyperparameter Tuned LSTM Validation Accuracy: 0.8109
Best Hyperparameter Tuned LSTM Validation F1: 0.7559
```

Bidirectional GRU Model (Hyperparameter Tuned)

In [14]:
```
def build_gru_model(hp):
    model = Sequential()
    model.add(Embedding(max_words, hp.Int('embedding_dim', min_value=32, max_value=128, step=32)))
    model.add(Bidirectional(GRU(
        hp.Int('gru_units', min_value=32, max_value=128, step=32),
        dropout=hp.Float('dropout', min_value=0.1, max_value=0.5, step=0.1),
        recurrent_dropout=hp.Float('recurrent_dropout', min_value=0.1, max_value=0.5, step=0.1))
    ))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

# Instantiate the tuner for GRU
gru_tuner = kt.Hyperband(
    build_gru_model,
    objective='val_accuracy',
    max_epochs=5,
    factor=3,
    directory='my_dir',
    project_name='gru_tuning'
)

# Run the search for GRU
print("Running KerasTuner search for Bidirectional GRU model...")
gru_tuner.search(X_train, y_train, validation_data=(X_val, y_val), epochs=5, callbacks=[EarlyStopping(monitor='val_loss', patience=2)])

# Get the best GRU model
best_gru_model = gru_tuner.get_best_models(num_models=1)[0]

# Evaluate the best GRU model
gru_val_pred = (best_gru_model.predict(X_val) > 0.5).astype(int)
gru_accuracy = accuracy_score(y_val, gru_val_pred)
gru_f1 = f1_score(y_val, gru_val_pred)
```

```python
print(f'Best Hyperparameter Tuned Bidirectional GRU Validation Accuracy: {gru_accuracy:.4f}')
print(f'Best Hyperparameter Tuned Bidirectional GRU Validation F1: {gru_f1:.4f}')
```

```
Trial 10 Complete [00h 04m 06s]
val_accuracy: 0.7925148010253906

Best val_accuracy So Far: 0.813525915145874
Total elapsed time: 00h 58m 39s
48/48 ━━━━━━━━━━━━━━━━━━━━ 4s 81ms/step
Best Hyperparameter Tuned Bidirectional GRU Validation Accuracy: 0.8135
Best Hyperparameter Tuned Bidirectional GRU Validation F1: 0.7676
```

LSTM Validation Accuracy: 0.8011 LSTM Validation F1: 0.7434 Bidirectional GRU Validation Accuracy: 0.7781 Bidirectional GRU Validation F1: 0.7455 Best Hyperparameter Tuned LSTM Validation Accuracy: 0.8109 Best Hyperparameter Tuned LSTM Validation F1: 0.7559 Best Hyperparameter Tuned Bidirectional GRU Validation Accuracy: 0.8135 Best Hyperparameter Tuned Bidirectional GRU Validation F1: 0.7676

Model Result Summary

Model	Accuracy	F1-score
LSTM	0.8011	0.7434
Bidirectional GRU	0.7781	0.7455
Best Hyperparameter Tuned LSTM	0.8109	0.7559
Best Hyperparameter Tuned Bidirectional GRU	0.8135	0.7676

The hyperparameter-tuned GRU model achieved the highest validation accuracy (0.8135) and validation F1-score (0.7676). Overall, hyperparameter tuning improved the accuracy of both models.

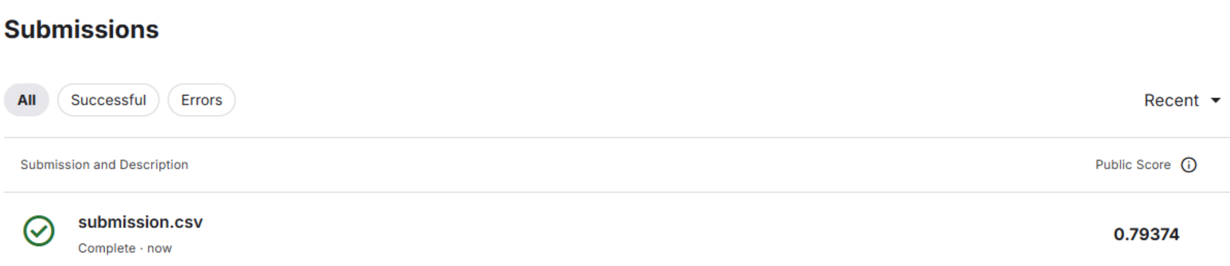
In [ ]:
```python
# Define best model
best_model = best_gru_model

# Predict best model
pred_test = (best_model.predict(X_test_seq) > 0.5).astype(int).flatten()

# Create Kaggle submission dataframe
submission = pd.DataFrame({'id': test_df['id'], 'target': pred_test})

# Save to CSV (path can be changed for local use)
submission_path = 'submission.csv'
submission.to_csv(submission_path, index=False)
print('Submission file saved to', submission_path)
```

Kaggle results:



Conclusion

Learnings:
The hyperparameter-tuned GRU model showed slightly better performance in both accuracy and F1-score compared to the tuned LSTM model. Importantly, tuning with KerasTuner helped find better model configurations, improving performance over the initial models. The dataset has some class imbalance, making the F1-score a key metric for evaluating the model's ability to identify disaster tweets.

What helped / did not help:
Hyperparameter tuning was clearly beneficial for improving performance. Using the F1-score provided a better evaluation for the imbalanced data. The initial basic LSTM and Bidirectional GRU models had similar performance, and the commented-out keyword grouping was not utilized in the final models, so their impact is unknown.

Future Improvements:
To further improve the model, consider addressing class imbalance using techniques like oversampling, undersampling, or class weights. Exploring more advanced models such as stacked RNNs or Transformer models like BERT could also be beneficial. Refining text preprocessing and exploring better ways to incorporate keyword and location data through feature engineering are also potential avenues. Finally, conducting a more extensive hyperparameter search, considering ensembling multiple models, and performing error analysis on misclassified tweets can help identify and address weaknesses.