## Introduction

Our final project of NLP was disaster tweet classification. The primary objective is to develop and evaluate an NLP model capable of distinguishing tweets related to disaster and those that are not. Our report will describe our dataset, models, and purpose of using those models, the setup used for the models, the results and conclusions that we've derived from our models, and our future direction in this project.

## Description of the deep learning network and training algorithm

**Naive Bayes:**

The naive Bayes classifier is a popular supervised machine-learning algorithm used for classification tasks such as text classification. It is categorized as a generative learning algorithm, implying that it models the distribution of inputs associated with a specific class or category. This methodology relies on the assumption that the features of the input data are conditionally independent given the class, enabling the algorithm to make prompt and accurate predictions. One of the strengths of naive Bayes is its simplicity. It's easy to implement and computationally efficient, making it particularly suitable for large datasets. The algorithm performs well even with limited computational resources. Below is the equation which is used to calculate the posterior probability using naive Bayes.

$$\underbrace{P(c \mid x)}_{\text{Posterior Probability}} = \frac{\overbrace{P(x \mid c)}^{\text{Likelihood}}\,\overbrace{P(c)}^{\text{Class Prior Probability}}}{\underbrace{P(x)}_{\text{Predictor Prior Probability}}}$$

$$P(c \mid \mathrm{X}) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

Figure 1: Equation to calculate the posterior probability using naive Bayes

In Figure 3, P(c|x) is the posterior probability of class (c, target) given predictor (x, attributes). P(c) is the prior probability of class. P(x|c) is the likelihood which is the probability of the predictor given class. P(x) is the prior probability of the predictor.

After training my naive Bayes mode I created the confusion matrix, you can see that for the Non-disaster tweet, 756 tweets are being correctly classified as Non-diaster., whereas 109 are being misclassified. For the disaster tweet, 469 are being correctly classified as disaster and 170 are being misclassified.

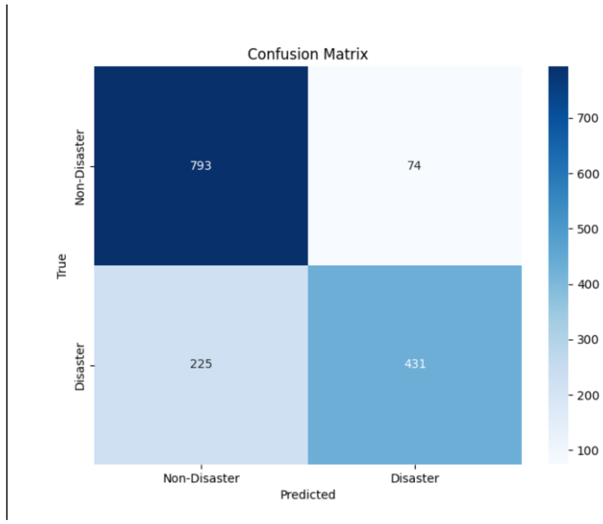


Figure 2: Confusion matrix of naive bayes

**Bidirectional LSTM:**
Bidirectional Lstm is a recurrent neural network primarily used for natural language processing. Unlike the standard LSTM, where the input flows in one direction, in bidirectional LSTM, the input flows in both directions. It is capable of utilizing information from both sides. So the BiLSTM adds one more LSTM layer which reverses the direction of information flow. Which implied that the input sequence flows backward in the additional LSTM layer. Then we combine the outputs from both LSTM layers in several ways, such as average, sum, multiplication, or concatenation. The advantage of this mode is that every component of the input sequence has information that is both past and present. For this reason, BiLstm can produce a more meaningful output as it has both forward and backward layers.

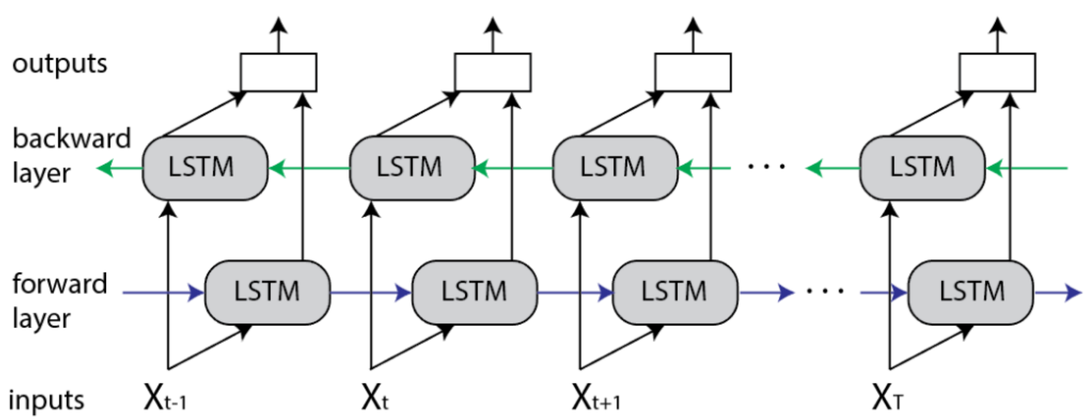


Figure 4: Bidirectional LSTM architecture

So training my bilstm for 20 epochs, For each epoch I am printing F1_score, validation loss, and training loss. whichever epoch is giving me the f1_ best score I am saving the model and I am printing the confusion matrix for it as seen in the below figure. So In the confusion matrix, you can see that for the Non-disaster tweet, 756 tweets are being correctly classified as Non-diaster., whereas 109 are being misclassified. For the disaster tweet 469 are being correctly classified as disaster and 170 are being misclassified.

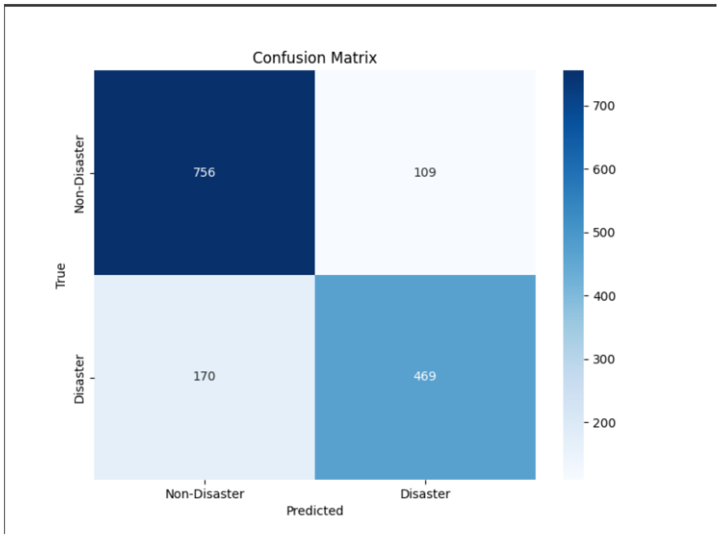


Figure 2: Confusion matrix of Bilstm

# My written code

a) Preprocessing function:

Used for preprocessing all the text

```
from collections import Counter
import torch
import numpy as np
from nltk import WordNetLemmatizer
import pandas as pd
import re
import spacy
from sklearn.model_selection import train_test_split
from spacy.lang.en.stop_words import STOP_WORDS
from torch import nn
from torch.utils.data import TensorDataset, DataLoader

nlp = spacy.load("en_core_web_sm")
def preprocess_data(X):
  contractions = {
      "ain't": "am not",
      "aren't": "are not",
```

```
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how does",
"i'd": "i would",
"i'd've": "i would have",
"i'll": "i will",
"i'll've": "i will have",
"i'm": "i am",
"i've": "i have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
```

```
    "needn't've": "need not have",
    "o'clock": "of the clock",
    "oughtn't": "ought not",
    "oughtn't've": "ought not have",
    "shan't": "shall not",
    "sha'n't": "shall not",
    "shan't've": "shall not have",
    "she'd": "she would",
    "she'd've": "she would have",
    "she'll": "she will",
    "she'll've": "she will have",
    "she's": "she is",
    "should've": "should have",
    "shouldn't": "should not",
    "shouldn't've": "should not have",
    "so've": "so have",
    "so's": "so is",
    "that'd": "that would",
    "that'd've": "that would have",
    "that's": "that is",
    "there'd": "there would",
    "there'd've": "there would have",
    "there's": "there is",
    "they'd": "they would",
    "they'd've": "they would have",
    "they'll": "they will",
    "they'll've": "they will have",
    "they're": "they are",
    "they've": "they have",
    "to've": "to have",
    "wasn't": "was not",
    " u ": " you ",
    " ur ": " your ",
    " n ": " and "}

def cont_to_exp(x):
    if type(x) is str:
        for key in contractions:
            value = contractions[key]
            x = x.replace(key, value)
        return x
    else:
        return x
```

```
  # Function to lemmatize and remove stop words
  # def lemmatize_and_remove_stop_words(text):
  #     doc = nlp(text)
  #     lemmatized_text = " ".join([token.lemma_ for token in doc if token.text.lower() not in
STOP_WORDS])
  #     return lemmatized_text

  # Function to lemmatize words
  def lemmatize_words(text):
      lemmatizer = WordNetLemmatizer()
      return " ".join([lemmatizer.lemmatize(word) for word in text.split()])

  # Preprocessing steps
  X = X.str.lower()
  X = X.apply(lambda x: cont_to_exp(x))
  X = X.apply(lambda x: re.sub('[^A-Z a-z 0-9-]+', ' ', x))
  X = X.apply(lambda x: re.sub(r'\b\d+\b', '', x))
  X = X.apply(lambda x: re.sub(r'<.*?>', '', x))
  X = X.apply(lambda x: re.sub(r'https?://\S+|www\.\S+', '', x))
  X = X.apply(lambda x: " ".join(x.split()))
  X = X.apply(lambda x: " ".join([t for t in x.split() if t not in STOP_WORDS]))
  X = X.apply(lambda text: lemmatize_words(text))
  X= X.str.strip()

  return X
```

b) Model.py:

I am defining bilstm neural network which will be used in streamlit.

```
import torch
from torch import nn

embedding_dim =300
output_dim =1
is_cuda = torch.cuda.is_available()
if is_cuda:
  device = torch.device("cuda")
  print("GPU is available")
else:
  device = torch.device("cpu")
  print("GPU not available, CPU used")
class TwitterClassification(nn.Module):
  def __init__(self, no_layers, vocab_size, hidden_dim, embedding_matrix):
      super(TwitterClassification, self).__init__()
```

```
        self.output_dim = output_dim
        self.hidden_dim = hidden_dim

        self.no_layers = no_layers
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding.from_pretrained(torch.FloatTensor(embedding_matrix),
freeze=True)
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=self.hidden_dim,
num_layers=no_layers,
                            batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(0.5)
        self.fc1 = nn.Linear(self.hidden_dim*2,128)
        self.relu = nn.ReLU()
        self.dropout_fc1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128,output_dim)
        self.sig = nn.Sigmoid()

    def forward(self, x, hidden):
        batch_size = x.size(0)
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)
        lstm_out = lstm_out.view(batch_size, -1, self.hidden_dim*2)
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim*2)
        out = self.dropout(lstm_out)
        out = self.fc1(out)
        out = self.relu(out)
        out = self.dropout_fc1(out)
        out = self.fc2(out)
        sig_out = self.sig(out)
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1]
        return sig_out, hidden

    def init_hidden(self, batch_size):
        h0 = torch.zeros((self.no_layers*2, batch_size, self.hidden_dim)).to(device)
        c0 = torch.zeros((self.no_layers*2, batch_size, self.hidden_dim)).to(device)
        hidden = (h0, c0)
        return hidden
```

c) Bilstm.py code
   The training of bilstm model

```
from collections import import Counter
import nltk
```

```
import sns as sns
import torch
import numpy as np
from nltk import WordNetLemmatizer
import pandas as pd
import re
import spacy
from sklearn.model_selection import train_test_split
from spacy.lang.en.stop_words import STOP_WORDS
from torch import nn
import pickle
from torch.utils.data import TensorDataset, DataLoader
from tqdm import tqdm
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
nltk.download('wordnet')
nlp = spacy.load("en_core_web_sm")
def preprocess_data(X):
  contractions = {
     "ain't": "am not",
     "aren't": "are not",
     "can't": "cannot",
     "can't've": "cannot have",
     "'cause": "because",
     "could've": "could have",
     "couldn't": "could not",
     "couldn't've": "could not have",
     "didn't": "did not",
     "doesn't": "does not",
     "don't": "do not",
     "hadn't": "had not",
     "hadn't've": "had not have",
     "hasn't": "has not",
     "haven't": "have not",
     "he'd": "he would",
     "he'd've": "he would have",
     "he'll": "he will",
     "he'll've": "he will have",
     "he's": "he is",
     "how'd": "how did",
     "how'd'y": "how do you",
     "how'll": "how will",
```

```
"how's": "how does",
"i'd": "i would",
"i'd've": "i would have",
"i'll": "i will",
"i'll've": "i will have",
"i'm": "i am",
"i've": "i have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that would",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there would",
```

```
    "there'd've": "there would have",
    "there's": "there is",
    "they'd": "they would",
    "they'd've": "they would have",
    "they'll": "they will",
    "they'll've": "they will have",
    "they're": "they are",
    "they've": "they have",
    "to've": "to have",
    "wasn't": "was not",
    " u ": " you ",
    " ur ": " your ",
    " n ": " and "}

def cont_to_exp(x):
    if type(x) is str:
        for key in contractions:
            value = contractions[key]
            x = x.replace(key, value)
        return x
    else:
        return x



# Function to lemmatize words
def lemmatize_words(text):
    lemmatizer = WordNetLemmatizer()
    return " ".join([lemmatizer.lemmatize(word) for word in text.split()])

# Preprocessing steps
X = X.str.lower()
X = X.apply(lambda x: cont_to_exp(x))
X = X.apply(lambda x: re.sub('[^A-Z a-z 0-9-]+', ' ', x))
X = X.apply(lambda x: re.sub(r'\b\d+\b', '', x))
X = X.apply(lambda x: re.sub(r'<.*?>', '', x))
X = X.apply(lambda x: re.sub(r'https?://\S+|www\.\S+', '', x))
X = X.apply(lambda x: " ".join(x.split()))
X = X.apply(lambda x: " ".join([t for t in x.split() if t not in STOP_WORDS]))
X = X.apply(lambda text: lemmatize_words(text))
X= X.str.strip()

return X
```

```
df= pd.read_csv('nlpproject/train.csv')
glove_path = 'glove.6B.300d.txt'

df['text'] = preprocess_data(df['text'])
print(df['text'])
X = df['text']
Y = df['target']
def tokenize(x_train, x_val):
   word_list = []
   for sent in x_train:
       for word in sent.split():
           if word != '':
               word_list.append(word)

   corpus = Counter(word_list)
   corpus_ = sorted(corpus, key=corpus.get, reverse=True)[:10000]
   onehot_dict = {w: i + 1 for i, w in enumerate(corpus_)}

   final_list_train, final_list_test = [], []
   for sent in x_train:
       final_list_train.append([onehot_dict[word] for word in sent.split() if word in
onehot_dict.keys()])

   for sent in x_val:
       final_list_test.append([onehot_dict[word] for word in sent.split() if word in
onehot_dict.keys()])

   return final_list_train, final_list_test,onehot_dict
# load glove embedding
def load_glove_embeddings(file_path):
   print("Loading GloVe embeddings...")
   embeddings_index = {}
   with open(file_path, 'r', encoding='utf-8') as file:
       for line in tqdm(file, total=400000):
           values = line.split()
           word = values[0]
           coefs = np.asarray(values[1:], dtype='float32')
           embeddings_index[word] = coefs
   return embeddings_index


glove_embeddings = load_glove_embeddings(glove_path)
# creating embedding matrix
def create_embedding_matrix(onehot_dict, glove_embeddings, embedding_dim):
```

```
    vocab_size = len(onehot_dict) + 1
    embedding_matrix = np.zeros((vocab_size, embedding_dim))

    for word, i in onehot_dict.items():
        embedding_vector = glove_embeddings.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector

    return embedding_matrix

def padding(X_token):
    max_len = max(len(seq) for seq in X_token)
    padded_sequences = [seq + [0] * (max_len - len(seq)) for seq in X_token]
    return padded_sequences

x_train, x_val, y_train, y_val = train_test_split(X, Y, test_size=0.2, random_state=42)

# Tokenize the data
x_train_token, x_val_tokens, vocab = tokenize(x_train,x_val)
with open('onehot_dicts.pkl', 'wb') as file:
    pickle.dump(vocab, file)
print("Dumping done")
x_train_pad = padding(x_train_token)
x_val_pad = padding(x_val_tokens)
embedding_dim = 300
embedding_matrix = create_embedding_matrix(vocab, glove_embeddings, embedding_dim)
with open('embedding_matrix.pkl', 'wb') as file:
    pickle.dump(embedding_matrix, file)

train_data = TensorDataset(torch.from_numpy(np.array(x_train_pad)),
torch.from_numpy(np.array(y_train)))
valid_data = TensorDataset(torch.from_numpy(np.array(x_val_pad)),
torch.from_numpy(np.array(y_val)))

batch_size = 32

train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size, drop_last=True)
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=batch_size, drop_last=True)

is_cuda = torch.cuda.is_available()
if is_cuda:
    device = torch.device("cuda")
    print("GPU is available")
else:
```

```
  device = torch.device("cpu")
  print("GPU not available, CPU used")

# defining Twitter classification module
class TwitterClassification(nn.Module):
  def __init__(self, no_layers, vocab_size, hidden_dim, embedding_matrix):
     super(TwitterClassification, self).__init__()

     self.output_dim = output_dim
     self.hidden_dim = hidden_dim

     self.no_layers = no_layers
     self.vocab_size = vocab_size
     self.embedding = nn.Embedding.from_pretrained(torch.FloatTensor(embedding_matrix),
freeze=True)
     self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=self.hidden_dim,
num_layers=no_layers,
                    batch_first=True, bidirectional=True)
     self.dropout = nn.Dropout(0.5)
     self.fc1 = nn.Linear(self.hidden_dim*2,128)
     self.relu = nn.ReLU()
     self.dropout_fc1 = nn.Dropout(0.5)
     self.fc2 = nn.Linear(128,output_dim)
     self.sig = nn.Sigmoid()

  def forward(self, x, hidden):
     batch_size = x.size(0)
     embeds = self.embedding(x)
     lstm_out, hidden = self.lstm(embeds, hidden)
     lstm_out = lstm_out.view(batch_size, -1, self.hidden_dim*2)
     lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim*2)
     out = self.dropout(lstm_out)
     out = self.fc1(out)
     out = self.relu(out)
     out = self.dropout_fc1(out)
     out = self.fc2(out)
     sig_out = self.sig(out)
     sig_out = sig_out.view(batch_size, -1)
     sig_out = sig_out[:, -1]
     return sig_out, hidden

  def init_hidden(self, batch_size):
     h0 = torch.zeros((self.no_layers*2, batch_size, self.hidden_dim)).to(device)
     c0 = torch.zeros((self.no_layers*2, batch_size, self.hidden_dim)).to(device)
```

```
      hidden = (h0, c0)
      return hidden

no_layers = 1
vocab_size = len(vocab) +1

output_dim = 1
hidden_dim = 256

# Instantiate the model with the embedding matrix
model = TwitterClassification(no_layers, vocab_size, hidden_dim, embedding_matrix)
model.to(device)


lr=3e-4
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=lr)


def acc(pred, label):
  pred = torch.round(pred.squeeze())
  return torch.sum(pred == label.squeeze()).item()


clip = 5
epochs = 20
valid_loss_min = np.Inf
epoch_tr_loss, epoch_vl_loss = [], []
epoch_tr_acc, epoch_vl_acc = [], []
best_model_path = 'best_modellatests.pth'
best_f1 = 0.0
for epoch in range(epochs):
  train_losses = []
  train_acc = 0.0
  all_labels = []
  all_pred = []
  model.train()
  h = model.init_hidden(batch_size)
  for inputs, labels in train_loader:
      inputs, labels = inputs.to(device), labels.to(device)
      h = tuple([each.data for each in h])
      model.zero_grad()
      output, h = model(inputs, h)
      loss = criterion(output.squeeze(), labels.float())
```

```
    loss.backward()
    train_losses.append(loss.item())
    accuracy= acc(output, labels)
    train_acc += accuracy
    nn.utils.clip_grad_norm_(model.parameters(), clip)
    optimizer.step()

val_h = model.init_hidden(batch_size)
val_losses = []
val_acc = 0.0
model.eval()
for inputs, labels in valid_loader:
    val_h = tuple([each.data for each in val_h])
    inputs, labels = inputs.to(device), labels.to(device)
    output, val_h = model(inputs, val_h)
    val_loss = criterion(output.squeeze(), labels.float())
    val_losses.append(val_loss.item())
    accuracy = acc(output, labels)
    pred = torch.round(output.squeeze()).detach().cpu().numpy()
    val_acc += accuracy
    all_pred.extend(pred)
    all_labels.extend(labels.cpu().numpy())

epoch_train_loss = np.mean(train_losses)
epoch_val_loss = np.mean(val_losses)
epoch_train_acc = train_acc / len(train_loader.dataset)
epoch_val_acc = val_acc / len(valid_loader.dataset)
epoch_tr_loss.append(epoch_train_loss)
epoch_vl_loss.append(epoch_val_loss)
epoch_tr_acc.append(epoch_train_acc)
epoch_vl_acc.append(epoch_val_acc)
f1 = f1_score(all_labels, all_pred)

print(f'Epoch {epoch + 1}')
print(f'train_loss : {epoch_train_loss} val_loss : {epoch_val_loss}')
print(f'train_accuracy : {epoch_train_acc * 100} val_accuracy : {epoch_val_acc * 100}')
print(f"F1 Score: {f1}")

if f1 > best_f1:
    best_f1 = f1
    conf_matrix = confusion_matrix(all_labels, all_pred)

    # Plot the confusion matrix
    plt.figure(figsize=(8, 6))
```

```
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Non-Disaster',
'Disaster'],
                yticklabels=['Non-Disaster', 'Disaster'])
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()

    # save the best f1_score model
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'train_loss': epoch_train_loss,
        'val_loss': epoch_val_loss,
        'train_accuracy': epoch_train_acc,
        'val_accuracy': epoch_val_acc,
        'f1_score': f1
    }, best_model_path)

  print(f'Best F1 Score: {best_f1}')
```

d) Naive bayes:
   Create naives bayes model

```
import joblib
import pandas as pd

import re
import spacy
from nltk import WordNetLemmatizer
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from spacy.lang.en.stop_words import STOP_WORDS
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
import matplotlib.pyplot as plt
import seaborn as sns
nlp = spacy.load("en_core_web_sm")
lemmatizer = WordNetLemmatizer()
def preprocess_data(X):
  contractions = {
    "ain't": "am not",
    "aren't": "are not",
    "can't": "cannot",
```

```
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how does",
"i'd": "i would",
"i'd've": "i would have",
"i'll": "i will",
"i'll've": "i will have",
"i'm": "i am",
"i've": "i have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
```

```
    "o'clock": "of the clock",
    "oughtn't": "ought not",
    "oughtn't've": "ought not have",
    "shan't": "shall not",
    "sha'n't": "shall not",
    "shan't've": "shall not have",
    "she'd": "she would",
    "she'd've": "she would have",
    "she'll": "she will",
    "she'll've": "she will have",
    "she's": "she is",
    "should've": "should have",
    "shouldn't": "should not",
    "shouldn't've": "should not have",
    "so've": "so have",
    "so's": "so is",
    "that'd": "that would",
    "that'd've": "that would have",
    "that's": "that is",
    "there'd": "there would",
    "there'd've": "there would have",
    "there's": "there is",
    "they'd": "they would",
    "they'd've": "they would have",
    "they'll": "they will",
    "they'll've": "they will have",
    "they're": "they are",
    "they've": "they have",
    "to've": "to have",
    "wasn't": "was not",
    " u ": " you ",
    " ur ": " your ",
    " n ": " and "}

def cont_to_exp(x):
    if type(x) is str:
        for key in contractions:
            value = contractions[key]
            x = x.replace(key, value)
        return x
    else:
        return x

# Function to lemmatize and remove stop words
```

```
    def lemmatize_and_remove_stop_words(text):
        doc = nlp(text)
        lemmatized_text = " ".join([token.lemma_ for token in doc if token.text.lower() not in
STOP_WORDS])
        return lemmatized_text

    # Function to lemmatize words
    def lemmatize_words(text):
        lemmatizer = WordNetLemmatizer()
        return " ".join([lemmatizer.lemmatize(word) for word in text.split()])

    # Preprocessing steps
    X = X.str.lower()
    X = X.apply(lambda x: cont_to_exp(x))
    X = X.apply(lambda x: re.sub('[^A-Z a-z 0-9-]+', ' ', x))
    X = X.apply(lambda x: re.sub(r'\b\d+\b', '', x))
    X = X.apply(lambda x: re.sub(r'<.*?>', '', x))
    X = X.apply(lambda x: re.sub(r'https?://\S+|www\.\S+', '', x))
    X = X.apply(lambda x: " ".join(x.split()))
    X = X.apply(lambda x: " ".join([t for t in x.split() if t not in STOP_WORDS]))
    X = X.apply(lambda text: lemmatize_words(text))
    X= X.str.strip()

    return X
def metrics(prediction, actual):
    print('Confusion_matrix \n', confusion_matrix(actual, prediction))
    print('\nAccuracy:', accuracy_score(actual, prediction))
    print('\nclassification_report\n')
    print(classification_report(actual, prediction))

# Loading Dataset
df_train = pd.read_csv('nlpproject/train.csv')
print(df_train.head(10))
columns_drop = ['id','keyword','location']

df_train = df_train.drop(columns=columns_drop)



# Assuming 'target' is the name of your target variable
target_counts = df_train['target'].value_counts()
print(target_counts)
# Plotting the distribution
plt.figure(figsize=(8, 6))
```

```
sns.barplot(x=target_counts.index, y=target_counts.values, palette="viridis")
plt.title('Distribution of Target Variable')
plt.xlabel('Target')
plt.ylabel('Count')
plt.show()
X_train, X_test,Y_train , Y_test = train_test_split(df_train['text'],df_train['target'], test_size=0.2)
X_train = preprocess_data(X_train)
X_test = preprocess_data(X_test)

print(X_train.head(10))



tf_vectorizer = TfidfVectorizer(max_features=10000,ngram_range=(1,2))

# convert into tfidf
xtrain_tf = tf_vectorizer.fit_transform(X_train)
xtest_tf = tf_vectorizer.transform(X_test)
vectorizer_filename = 'tfidf_vectorizer.joblib'
joblib.dump(tf_vectorizer, vectorizer_filename)

# Naive bayes model prediction
clb_tf = MultinomialNB().fit(xtrain_tf, Y_train)
predicted = clb_tf.predict(xtest_tf)
metrics(predicted, Y_test)
model_filename = 'multinomial_nb_model.joblib'
joblib.dump(clb_tf, model_filename)

print(f"Model saved to {model_filename}")

import seaborn as sns
import matplotlib.pyplot as plt

# Calculate confusion matrix
conf_matrix = confusion_matrix(Y_test, predicted)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Non-Disaster',
'Disaster'],
        yticklabels=['Non-Disaster', 'Disaster'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
```

```
plt.show()
```

e) Preprocess function:
   This function created to handle streamlit text and apply necessary step after that.

```
import joblib
from spacy.lang.en.stop_words import STOP_WORDS
from nltk import WordNetLemmatizer
import nltk
import spacy
import re
from Model import TwitterClassification
nltk.download('wordnet')
nlp = spacy.load("en_core_web_sm")
import torch

import numpy as np
import pickle
is_cuda = torch.cuda.is_available()
if is_cuda:
  device = torch.device("cuda")
  print("GPU is available")
else:
  device = torch.device("cpu")
  print("GPU not available, CPU used")
def preprocess_data(text):
  contractions = {
     "ain't": "am not",
     "aren't": "are not",
     "can't": "cannot",
     "can't've": "cannot have",
     "'cause": "because",
     "could've": "could have",
     "couldn't": "could not",
     "couldn't've": "could not have",
     "didn't": "did not",
     "doesn't": "does not",
     "don't": "do not",
     "hadn't": "had not",
     "hadn't've": "had not have",
     "hasn't": "has not",
     "haven't": "have not",
     "he'd": "he would",
     "he'd've": "he would have",
```

```
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how does",
"i'd": "i would",
"i'd've": "i would have",
"i'll": "i will",
"i'll've": "i will have",
"i'm": "i am",
"i've": "i have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
```

```
    "so've": "so have",
    "so's": "so is",
    "that'd": "that would",
    "that'd've": "that would have",
    "that's": "that is",
    "there'd": "there would",
    "there'd've": "there would have",
    "there's": "there is",
    "they'd": "they would",
    "they'd've": "they would have",
    "they'll": "they will",
    "they'll've": "they will have",
    "they're": "they are",
    "they've": "they have",
    "to've": "to have",
    "wasn't": "was not",
    " u ": " you ",
    " ur ": " your ",
    " n ": " and "}

def cont_to_exp(x):
    if type(x) is str:
        for key in contractions:
            value = contractions[key]
            x = x.replace(key, value)
        return x
    else:
        return x



# Function to lemmatize words
def lemmatize_words(text):
    lemmatizer = WordNetLemmatizer()
    return " ".join([lemmatizer.lemmatize(word) for word in text.split()])

# Preprocessing steps
text = text.lower()
text = cont_to_exp(text)
text = re.sub('[^A-Z a-z 0-9-]+', ' ', text)
text = re.sub(r'\b\d+\b', '', text)
text = re.sub(r'<.*?>', '', text)
text = re.sub(r'https?://\S+|www\.\S+', '', text)
text = " ".join(text.split())
```

```
    text = " ".join([t for t in text.split() if t not in STOP_WORDS])
    text = lemmatize_words(text)
    text = text.strip()
    return text

def tokenize_text(text):

    final_list = []
    for word in text.split():
        if word in loaded_onehot_dict.keys():
            print(word)
            final_list.append(loaded_onehot_dict[word])

    return final_list


with open('onehot_dicts.pkl', 'rb') as file:
    loaded_onehot_dict = pickle.load(file)


# # loading tfidf vectorizer
# loaded_vectorizer = joblib.load("tfidf_vectorizer.joblib")
# # loading naive bayes
# loaded_model = joblib.load("multinomial_nb_model.joblib")
#
#
#
#
# this are required when create model
with open('embedding_matrix.pkl', 'rb') as file:
    print("going into")
    loaded_embedding_matrix = pickle.load(file)
print("exited")
hidden_dim = 256
no_layers = 1
vocab_size = len(loaded_onehot_dict) +1
embedding_matrix = loaded_embedding_matrix

# loading Bilstm
checkpoint = torch.load('best_modellatests.pth')
model = TwitterClassification(no_layers, vocab_size, hidden_dim, embedding_matrix)
model = model.to(device)
model.load_state_dict(checkpoint['model_state_dict'])
model.eval()
```

```
# streamlit example when user puts the tweet in
user_input = "Just happened a terrible car crash"
processed_input = preprocess_data(user_input)
#
# # for naive bayes
# text_transformed = loaded_vectorizer.transform([processed_input])
# prediction_NB = loaded_model.predict(text_transformed)[0]
# print(prediction_NB)
#
#
# # for Bi lstm output
tokenized_input = tokenize_text(processed_input)
input_tensor = torch.from_numpy(np.array([tokenized_input]))
input_tensor = input_tensor.to(device)
with torch.no_grad():
  output, _ = model(input_tensor, model.init_hidden(1))
  prediction = torch.round(output).item()

# Print the prediction
if prediction == 1:
  print("The text is related to a disaster.")
else:
  print("The text is not related to a disaster.")
```

## Result

From the performance of our models, we saw that, while the neural networks from both constructed BiLSTM and pre-trained DistilBERT could outperform the Naive Bayes model, we also found that the models all performed relatively closely. This could be in part, due to the dataset structure of how some words were clear indications of one target or the other.

Below are some sample outputs from our app.



## Summary and Conclusions

From our project, we can show that an affordable and light gpu model can be constructed to tackle this fake and real disaster tweet problem. We did run into a few problems with preprocessing, for example, due to filtering out stopwords using Spacy, past and present tense were ambiguous to the model in some situations, so past disasters and ongoing disasters were treated the same (I was in a car accident vs. I am in a car accident).
In the future, we plan to pay closer attention to the specific words/show embeddings and associations of the disaster and do additional data analysis on the focus/robustness of our models.
For future exploration, we could automate disaster classification on hashtagged tweets to filter out false or mistakenly tagged tweets from interfering with the information provided by the true disaster tweets and updates. We could also further classify positively predicted tweets between which type of disaster it is (fire, hurricane, man-made disasters, etc.), and give referral links/phone numbers as to which resource is best suited to the disaster.

Codes
I have used 85 percent of the code I found in the in the internet

## References

https://www.kaggle.com/code/harrycheng5/nlp-fake-tweets-classification
https://www.baeldung.com/cs/bidirectional-vs-unidirectional-lstm
https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/
https://github.com/amir-jafari/NLP/blob/master/Lecture_08/Lecture%20Code/3-LSTM_Sentiment_Analysis_Custom.py