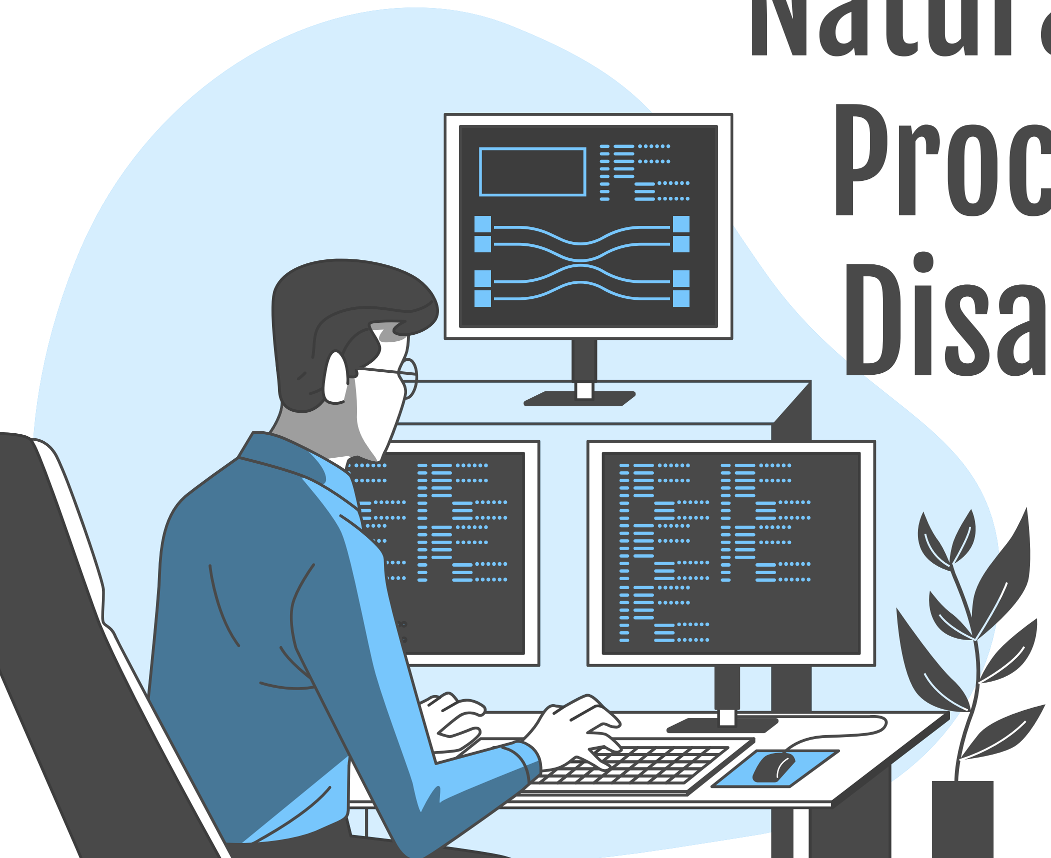
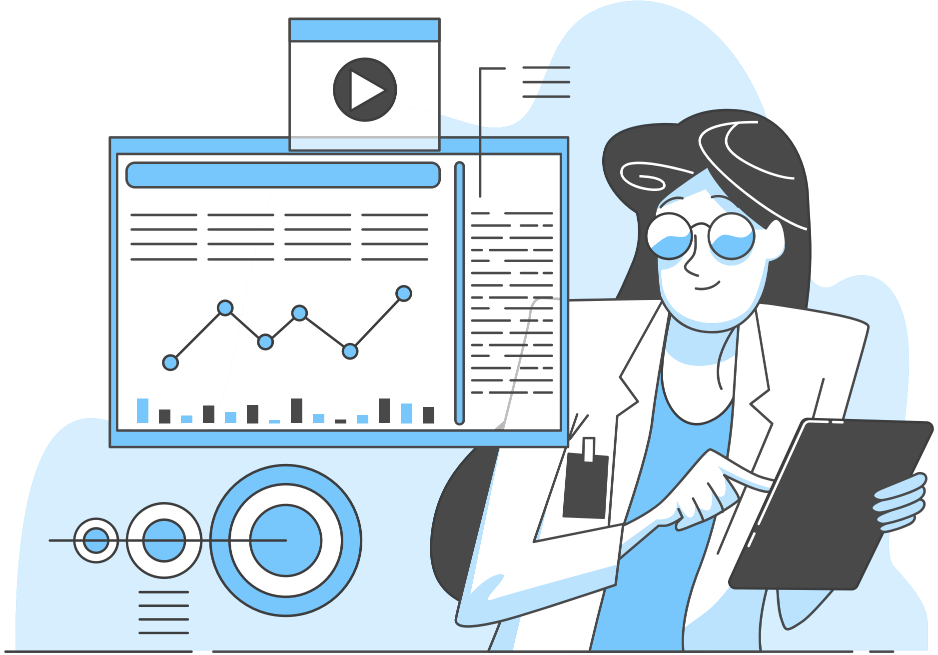


Natural Language Processing with Disaster Tweets

Jajapuram Giridhar Reddy (20BRS1004)
S Abenav Ram (20BRS1137)
Rohan Jacob John (20BRS1159)
Penmetsa Gajendra Varma (20BRS1218)

Table of Contents



01
PROBLEM STATEMENT

PROBLEM STATEMENT

Twitter is a well-liked site for sharing current information and is a useful tool for first responders. People frequently use Twitter during a crisis to report circumstances, ask for assistance, and share updates about the situation. Social media's quick information dissemination makes disaster location, timing, and impact more predictable. It is required to create machine learning algorithms that can efficiently evaluate and comprehend the large quantity of information accessible in order to use Twitter data to predict disasters. Despite the difficulties, predicting disasters using Twitter data is a crucial field of study with the potential to save lives and lessen the effects of natural disasters.

02
ABSTRACT

ABSTRACT

The Natural Language Processing with Disaster Tweets study is a research effort that leverages the power of NLP to analyze tweets that are related to disaster events. The objective of the study is to use NLP techniques such as text classification and sentiment analysis to gain a better understanding of the information contained in these tweets. The insights gathered from this NLP analysis can be valuable for disaster response organizations, as they can help to prioritize resources, identify areas of need, and address public concerns. Additionally, by understanding the public's perception of the disaster, response organizations can better tailor their communications and actions to effectively meet the needs of those affected.

03

Proposed Technology

PROPOSED TECHNOLOGY

The first step in solving this problem is to perform exploratory data analysis (EDA) to understand the structure and content of your tweet data. This includes visualizing the distribution of disaster types, tweet lengths, word frequencies, and more. Next, we perform data cleansing to pre-process the tweets and prepare them for analysis. This usually involves removing irrelevant information such as URLs, hashtags, emojis, as well as correcting misspelled words and removing stop words. After preprocessing there are two ways to proceed with the problem.

One approach is to use a pre-trained BERT model to generate representations of tweets and use a CNN to classify tweets into relevant disaster categories based on the generated representations. A CNN acts as a classifier and a BERT model provides a high-level representation of tweets.

04
JUSTIFICATION

JUSTIFICATION

The justification for the Natural Language Processing with Disaster Tweets study stems from the increasing importance of social media as a source of information during disaster events. In today's digital age, people often turn to social media platforms like Twitter to share information and updates about disasters as they unfold. This has created a vast and constantly updating repository of information related to disasters, which can be analyzed to gain valuable insights. However, manual analysis of this large amount of data can be time-consuming and resource-intensive. This is where NLP comes into play. By using NLP techniques, the study aims to automatically process and analyze these tweets, reducing the time and effort required to gain insights.

05

EXPECTED RESULT

EXPECTED RESULT

The expected result of the disaster tweet NLP problem is a model that accurately classifies tweets into relevant disaster categories. Model accuracy depends on many factors, including: Quality of training data, size of data set, choice of model architecture, and specific implementation of the model. This will help in providing disaster relief to the affected people and provide quick relief to the affected people.

06
IMPLEMENTATION

BERT & LSTM

BERT

```python
import pandas as pd
import numpy as np
df = pd.read_csv("/content/train (1).csv",encoding='ISO-8859-1')
df.head(5)
```

	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

Importing some commonly used libraries and Reading the dataset

```
df = df.drop(['id','keyword','location'],axis=1)
df.head()
```

	text	target
0	Our Deeds are the Reason of this #earthquake M...	1
1	Forest fire near La Ronge Sask. Canada	1
2	All residents asked to 'shelter in place' are ...	1
3	13,000 people receive #wildfires evacuation or...	1
4	Just got sent this photo from Ruby #Alaska as ...	1

```
df['target'].value_counts()
```

```
0    4342
1    3271
Name: target, dtype: int64
```

```
df_0_class = df[df['target']==0]
df_1_class = df[df['target']==1]
df_0_class_undersampled = df_0_class.sample(df_1_class.shape[0])
df = pd.concat([df_0_class_undersampled, df_1_class], axis=0)
```

- The 1st block clears the unnecessary features from the dataset (id, keyword, and location)
- The 2nd block identifies number of target types and their respective counts are viewed
- Then next block performs undersampling on the dataset based on target column
    - The 1st two lines create 2 dataframes by filtering the dataset to only include rows based on the value of target
    - 3rd line performs undersampling by randomly selecting a subset of rows from df_0_class that has same number of rows as df_1_class

- The last line in this block concatenates the undersampled df_0_class_undersampled dataframe and the original df_1_class DataFrame along the rows (axis=0) and assigns the result back to the original df DataFrame. This results in a new DataFrame that has the same number of rows for both classes.

Overall, this block of code balances the number of samples for each class in the target column by randomly undersampling the majority class (target=0) to have the same number of samples as the minority class (target=1).

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df['text'],df['target'], stratify=df['target'])
```

```python
!pip install tensorflow-text
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_text as text
```

Here we are splitting the data into test and train based on target column of df (stratify will split the data with equal proportions of target column values) Then we import necessary tensorflow libraries where hub is a repository for pretrained NLP models and text provides a collection of text-specific operations and layers for building NLP models

```python
preprocess = hub.KerasLayer("https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3")
encoder = hub.KerasLayer("https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4")
```

```python
text_input = tf.keras.layers.Input(shape=(), dtype=tf.string, name='text-layer')
preprocessed_text = preprocess(text_input)
outputs = encoder(preprocessed_text)
d_layer = tf.keras.layers.Dropout(0.1, name="dropout-layer")(outputs['pooled_output'])
d_layer = tf.keras.layers.Dense(1, activation='sigmoid', name="output")(d_layer)
model = tf.keras.Model(inputs=[text_input], outputs = [d_layer])
```

- The 1st block loads a preprocessing layer for BERT and BERT encoder which are responsible for tokenizing input text and preparing it for encoding by BERT encoder and the encoder is responsible for generating high quality representations of input text.
- The 2nd block define a tensorflow keras model for classifying text using BERT.
  - 1st line defines input layer for the text data
  - 2nd line applies the pre-processing layer that was loaded earlier to the text input layer.

- 3rd line applies BERT encoder.
- 4th line applies a dropout layer to the output of the BERT encoder. The ‘0.1’ argument in the command specifies the dropout rate. The outputs['pooled_output'] selects the output of the BERT model's pooler layer
- 5th line applies dense layer with single output unit and a sigmoid activation function to the output of the dropout layer
- The last line defines the final Keras model, which takes the text input layer as input and produces ‘d_layer’ as output

```
1 model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 text-layer (InputLayer)        [(None,)]            0           []

 keras_layer (KerasLayer)       {'input_type_ids':   0           ['text-layer[0][0]']
                                (None, 128),
                                 'input_mask': (Non
                                e, 128),
                                 'input_word_ids':
                                (None, 128)}

 keras_layer_1 (KerasLayer)     {'pooled_output': (  109482241   ['keras_layer[0][0]',
                                None, 768),                       'keras_layer[0][1]',
                                 'encoder_outputs':               'keras_layer[0][2]']
                                 [(None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768),
                                 (None, 128, 768)],
                                 'sequence_output':
                                 (None, 128, 768),
                                 'default': (None,
                                768)}

 dropout-layer (Dropout)        (None, 768)          0           ['keras_layer_1[0][13]']

 output (Dense)                 (None, 1)            769         ['dropout-layer[0][0]']

==================================================================================================
Total params: 109,483,010
Trainable params: 769
Non-trainable params: 109,482,241
__________________________________________________________________________________________________
```

MODEL SUMMARY

```python
m= [
      tf.keras.metrics.BinaryAccuracy(name='accuracy'),
      tf.keras.metrics.Precision(name='precision'),
      tf.keras.metrics.Recall(name='recall')
]
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=m)
```

```python
model.fit(X_train, y_train, epochs=10)
model.evaluate(X_test, y_test)
```

```
Epoch 1/10
154/154 [==============================] - 1946s 13s/step - loss: 0.6424 - accuracy: 0.6298 - precision: 0.6236 - recall: 0.6551
Epoch 2/10
154/154 [==============================] - 1907s 12s/step - loss: 0.6085 - accuracy: 0.6863 - precision: 0.6847 - recall: 0.6906
Epoch 3/10
154/154 [==============================] - 1939s 13s/step - loss: 0.5958 - accuracy: 0.6910 - precision: 0.6888 - recall: 0.6967
Epoch 4/10
154/154 [==============================] - 1933s 13s/step - loss: 0.5796 - accuracy: 0.7051 - precision: 0.7051 - recall: 0.7049
Epoch 5/10
154/154 [==============================] - 1973s 13s/step - loss: 0.5759 - accuracy: 0.7077 - precision: 0.7098 - recall: 0.7028
Epoch 6/10
154/154 [==============================] - 1946s 13s/step - loss: 0.5663 - accuracy: 0.7159 - precision: 0.7166 - recall: 0.7142
Epoch 7/10
154/154 [==============================] - 1891s 12s/step - loss: 0.5687 - accuracy: 0.7144 - precision: 0.7186 - recall: 0.7049
Epoch 8/10
154/154 [==============================] - 1871s 12s/step - loss: 0.5527 - accuracy: 0.7240 - precision: 0.7292 - recall: 0.7126
Epoch 9/10
154/154 [==============================] - 1880s 12s/step - loss: 0.5522 - accuracy: 0.7275 - precision: 0.7325 - recall: 0.7167
Epoch 10/10
154/154 [==============================] - 1911s 12s/step - loss: 0.5538 - accuracy: 0.7218 - precision: 0.7252 - recall: 0.7142
52/52 [==============================] - 644s 12s/step - loss: 0.5400 - accuracy: 0.7323 - precision: 0.7568 - recall: 0.6846
[0.5400314331054688, 0.7322738170623779, 0.7567567825317383, 0.684596598148346]
```

The 2 blocks compiles the Keras model that was defined earlier

- The 1st line creates a list of Keras metrics to be used for model evaluation during training. (Here there are 3: BinaryAccuracy, Precision, Recall)
- 2nd line compiles Keras model using the compile function with the optimizer argument the algorithm is provided (adam) for training.
- The next lines are used for training and evaluating the Keras model using fit and evaluate functions respectively.

```python
import numpy as np
y_predicted = model.predict(X_test)
y_predicted = y_predicted.flatten()
y_predicted = np.where(y_predicted > 0.5, 1, 0)
from sklearn.metrics import confusion_matrix, classification_report
matrix = confusion_matrix(y_test, y_predicted)
matrix
```

```
52/52 [==============================] - 650s 12s/step
array([[638, 180],
       [258, 560]])
```

```python
print(classification_report(y_test, y_predicted))
```

```
              precision    recall  f1-score   support

           0       0.71      0.78      0.74       818
           1       0.76      0.68      0.72       818

    accuracy                           0.73      1636
   macro avg       0.73      0.73      0.73      1636
weighted avg       0.73      0.73      0.73      1636
```

First block uses trained Keras model to make predictions on the test data and compute the confusion matrix.

- 2nd line uses trained Keras model to make predictions.
- 3rd line flattens the predicted probabilities into a 1D array
- 4th line converts probabilities to binary predictions by thresholding at 0.5
- Then the next lines imports and uses the necessary libraries for creating a confusion matrix and a classification report which provides a more detailed evaluation of model's performance on test data

A Brief Summary - BERT

1. Importing necessary libraries and reading the dataset.
2. In the preprocessing step, removing irrelevant columns from the data is done.
3. Then code performs under-sampling to balance the target classes and splits the data into training and testing sets.
4. Then the code uses a pre-trained BERT model for text classification, compiles and trains the model.
5. Then the evaluation of its performance is displayed
6. A confusion matrix is constructed and a classification report is printed. This outputs the report containing metrics such as accuracy, precision, and recall.

LSTM

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re

import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import SnowballStemmer

from sklearn import model_selection, metrics, preprocessing, ensemble, model_selection, metrics
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer


import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Conv1D, Bidirectional, LSTM, Dense, Dropout, Input, SpatialDropout1D
from tensorflow.keras.callbacks import ModelCheckpoint, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
```

1)importing libraries commonly used in data analysis and visualization they can be used to perform various data analysis tasks such as reading and cleaning data, creating isualizations, and building machine learning models.

2)importing and downloading the Natural Language Toolkit (NLTK) library and its resources. NLTK is a popular library for natural language processing (NLP) tasks in Python

3)importing various modules and classes from the TensorFlow library, which is a popular library for building and training machine learning models

```python
# Rreading train dataset
file_path = "./train.csv"
raw_data = pd.read_csv(file_path)
print("Data points count: ", raw_data['id'].count())
raw_data.head()
```

```
Data points count:  7613
```

	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

Reading Dataset

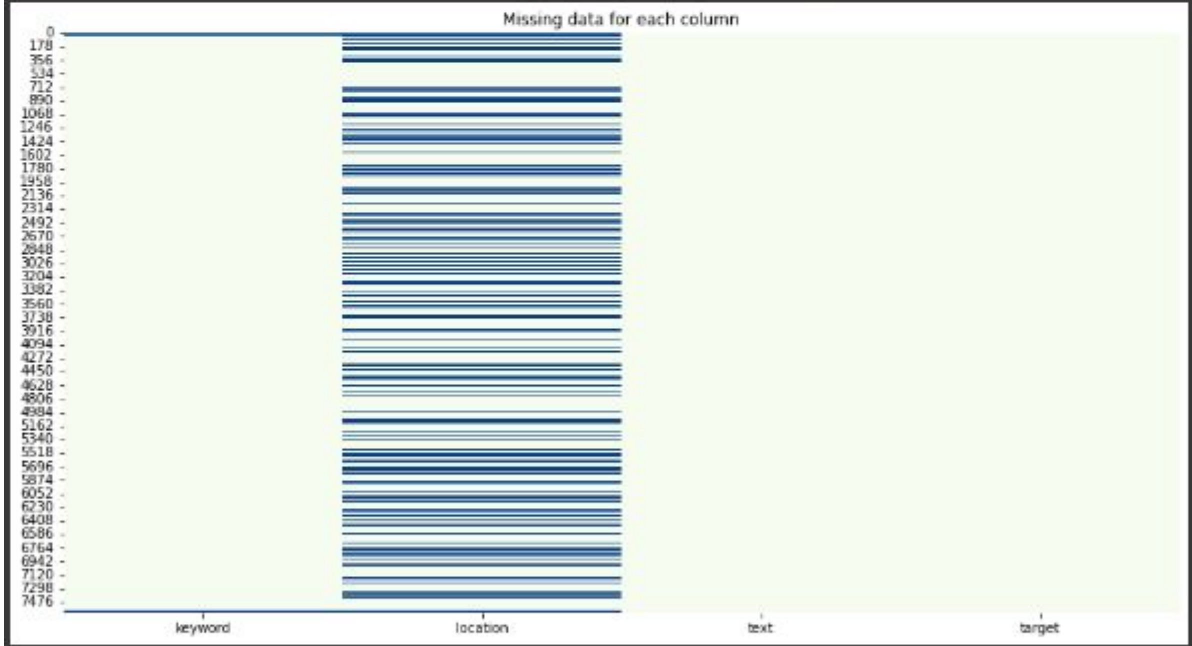
Preprocessing

```
# Plotting target value counts
plt.figure(figsize=(10,8))
ax = raw_data['target'].value_counts().sort_values().plot(kind="bar")
ax.grid(axis="y")
plt.suptitle("Target Value Counts", fontsize=20)
plt.show()
```

```python
print("Number of missing data for column keyword: ", raw_data['keyword'].isna().sum())
print("Number of missing data for column location: ", raw_data['location'].isna().sum())
print("Number of missing data for column text: ", raw_data['text'].isna().sum())
print("Number of missing data for column target: ", raw_data['target'].isna().sum())
```

```
Number of missing data for column keyword:  61
Number of missing data for column location:  2533
Number of missing data for column text:  0
Number of missing data for column target:  0
```

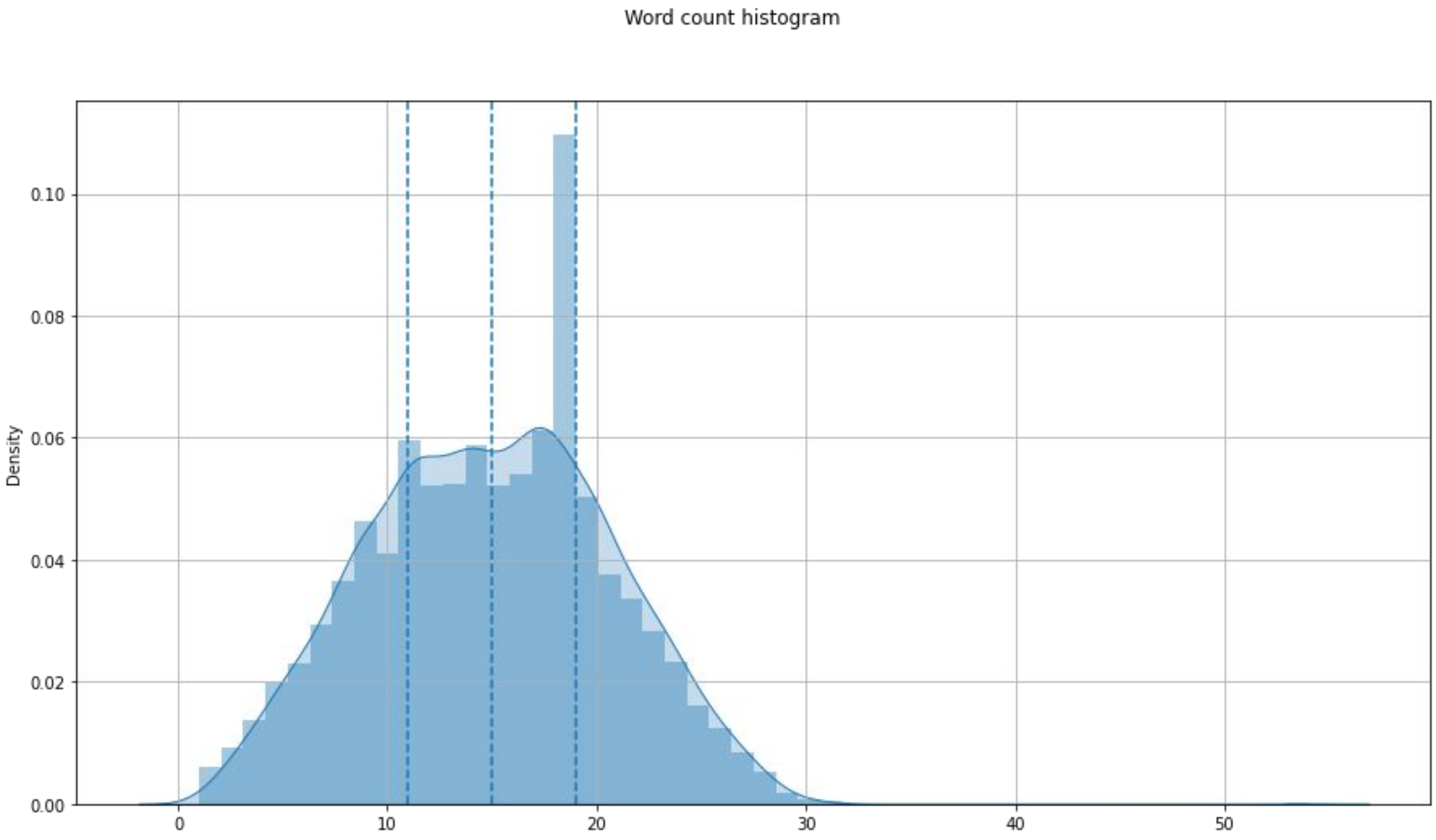
```python
plt.figure(figsize=(15,8))
sns.heatmap(raw_data.drop('id', axis=1).isnull(), cbar=False, cmap="GnBu").set_title("Missing data for each column")
plt.show()
```



```python
plt.figure(figsize=(15,8))
raw_data['word_count'] = raw_data['text'].apply(lambda x: len(x.split(" ")) )
sns.distplot(raw_data['word_count'].values, hist=True, kde=True, kde_kws={"shade": True})
plt.axvline(raw_data['word_count'].describe()['25%'], ls="--")
plt.axvline(raw_data['word_count'].describe()['50%'], ls="--")
plt.axvline(raw_data['word_count'].describe()['75%'], ls="--")

plt.grid()
plt.suptitle("Word count histogram")
plt.show()

# remove rows with under 3 words
raw_data = raw_data[raw_data['word_count']>2]
raw_data = raw_data.reset_index()
```


Word count histogram
Density
0.00
0.02
0.04
0.06
0.08
0.10
0
10
20
30
40
50

- The first plot shows the target value counts of the data using a bar chart.
- The next four print statements display the number of missing data for each column in the DataFrame.
- The second plot shows the distribution of word counts in the text column using a histogram.
- Rows with less than 3 words in the text column are removed from the DataFrame.
- The final three print statements display the summary statistics of the word count column

```python
import nltk
nltk.download('punkt')
# Clean text columns
stop_words = set(stopwords.words('english'))
stemmer = SnowballStemmer('english')


def clean_text(each_text):

    # remove URL from text
    each_text_no_url = re.sub(r"http\S+", "", each_text)

    # remove numbers from text
    text_no_num = re.sub(r'\d+', '', each_text_no_url)

    # tokenize each text
    word_tokens = word_tokenize(text_no_num)

    # remove sptial character
    clean_text = []
    for word in word_tokens:
        clean_text.append("".join([e for e in word if e.isalnum()]))

    # remove stop words and lower
    text_with_no_stop_word = [w.lower() for w in clean_text if not w in stop_words]

    # do stemming
    stemmed_text = [stemmer.stem(w) for w in text_with_no_stop_word]

    return " ".join(" ".join(stemmed_text).split())


raw_data['clean_text'] = raw_data['text'].apply(lambda x: clean_text(x) )
raw_data['keyword'] = raw_data['keyword'].fillna("none")
raw_data['clean_keyword'] = raw_data['keyword'].apply(lambda x: clean_text(x) )
# Combine column 'clean_keyword' and 'clean_text' into one
raw_data['keyword_text'] = raw_data['clean_keyword'] + " " + raw_data["clean_text"]
```

text preprocessing by cleaning the text columns in the dataset. It starts by downloading the punkt tokenizer from the nltk library.

The `clean_text` function is defined to remove URLs, numbers, special characters, and stop words, lowercases the text, and performs stemming.

The function is applied to the `text` column in the dataset using the `apply` method, and the cleaned text is stored in a new column named `clean_text`. The same function is also applied to the `keyword` column after filling missing values with the string "none", and the cleaned keywords are stored in a new column named `clean_keyword`.

Finally, a new column named `keyword_text` is created by combining the `clean_keyword` and `clean_text` columns.

```
feature = 'keyword_text'
label = "target"

# split train and test
X_train, X_test,y_train, y_test = model_selection.train_test_split(raw_data[feature],
                                                                    raw_data[label],
                                                                    test_size=0.3,
                                                                    random_state=0,
                                                                    shuffle=True)
```

the `feature` and `label` variables are defined, with `feature` being the column of text data that will be used as the predictor, and `label` being the target variable. The `train test split` function from scikit-learn is then used to split the data into training and testing sets.

The training set consists of 70% of the data, while the testing set consists of the remaining 30%. The split is randomized using the `random state` parameter to ensure reproducibility, and the data is shuffled before the split using the `shuffle` parameter.

The predictor variable is taken from the `feature` column of the raw data, while the target variable is taken from the `label` column. The resulting data is stored in the `X train`, `X_test`, `y_train`, and `y_test` variables.

```
X_train_GBC = X_train.values.reshape(-1)
x_test_GBC = X_test.values.reshape(-1)
# Vectorize text
vectorizer = CountVectorizer()
X_train_GBC = vectorizer.fit_transform(X_train_GBC)
x_test_GBC = vectorizer.transform(x_test_GBC)
# Train the model
model = ensemble.GradientBoostingClassifier(learning_rate=0.1,
                                            n_estimators=2000,
                                            max_depth=9,
                                            min_samples_split=6,
                                            min_samples_leaf=2,
                                            max_features=8,
                                            subsample=0.9)
model.fit(X_train_GBC, y_train)
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=9,
                           max_features=8, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=2, min_samples_split=6,
                           min_weight_fraction_leaf=0.0, n_estimators=2000,
                           n_iter_no_change=None, presort='auto',
                           random_state=None, subsample=0.9, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
# Evaluate the model
predicted_prob = model.predict_proba(x_test_GBC)[:,1]
predicted = model.predict(x_test_GBC)

accuracy = metrics.accuracy_score(predicted, y_test)
print("Test accuracy: ", accuracy)
print(metrics.classification_report(y_test, predicted, target_names=["0", "1"]))
print("Test F-scoare: ", metrics.f1_score(y_test, predicted))
```

Used Gradient Boosting Classifier (GBC) model to classify tweets as either related to a real disaster (`target=1`) or not (`target=0`).

First, the text data is cleaned and preprocessed using functions defined earlier, including removing URLs, numbers, special characters, and stopwords, and stemming words.

Then the data is split into training and test sets using `train_test_split` from `sklearn.model_selection`.

The `CountVectorizer` is used to vectorize the text data into numerical features that can be used for training the GBC model.

```python
path_to_glove_file = './glove.6B.300d.txt' # download link: http://nlp.stanford.edu/data/glove.6B.zip
embedding_dim = 300
learning_rate = 1e-3
batch_size = 1024
epochs = 20
sequence_len = 100
```

```python
# Tokenize train data
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)

word_index = tokenizer.word_index
vocab_size = len(word_index) + 1
print("Vocabulary Size: ", vocab_size)
```

```
Vocabulary Size:  11148
```

defines several variables for the LSTM model:

- `path_to_glove_file` is the path to the pre-trained word embeddings file.
- `embedding_dim` is the dimension of the embedding space.
- `learning_rate` is the learning rate used by the optimizer during training.
- `batch_size` is the number of samples processed before the model is updated.
- `epochs` is the number of times the entire training dataset is iterated over during training.
- `sequence_len` is the maximum length of a sequence, which will be used to pad or truncate sequences to a fixed length.

Here, we are initializing a `Tokenizer` object and fitting it on the training data `X_train` using `fit_on_texts` method, which tokenizes each word in the text and assigns a unique integer index to each word.

We then retrieve the vocabulary size by adding 1 to the number of unique words in the `word_index`, as `Tokenizer` index starts from 1 instead of 0. The `vocab_size` variable will be used to define the input shape of the neural network.

```python
# Read word embeddings
embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print("Found %s word vectors." % len(embeddings_index))
```

```python
# Define embedding layer in Keras
embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

embedding_layer = tf.keras.layers.Embedding(vocab_size,
                                            embedding_dim,
                                            weights=[embedding_matrix],
                                            input_length=sequence_len,
                                            trainable=False)
```

Loads pre-trained word embeddings from the GloVe file and creates an embedding matrix to be used as weights for the embedding layer of the LSTM model.

The embeddings_index dictionary is populated with word embeddings from the GloVe file, where each word is mapped to its corresponding embedding vector. Then, an embedding matrix of shape (vocab_size, embedding_dim) is initialized with zeros, where each row represents a word in the vocabulary and each column represents a dimension of the embedding space.

The embedding matrix is then filled with the pre-trained embeddings from the embeddings_index dictionary. If a word in the vocabulary is not found in the embeddings_index dictionary, then its corresponding row in the embedding matrix will remain all zeros.

```python
# Define model architecture
sequence_input = Input(shape=(sequence_len, ), dtype='int32')
embedding_sequences = embedding_layer(sequence_input)

x = Conv1D(128, 5, activation='relu')(embedding_sequences)
x = Bidirectional(LSTM(128, dropout=0.5, recurrent_dropout=0.2))(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(512, activation='relu')(x)
outputs = Dense(1, activation='sigmoid')(x)
model = Model(sequence_input, outputs)
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 100)]             0

 embedding (Embedding)       (None, 100, 300)          3344400

 conv1d (Conv1D)             (None, 96, 128)           192128

 bidirectional (Bidirectiona  (None, 256)              263168
 l)

 dense (Dense)               (None, 512)               131584

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 512)               262656

 dense_2 (Dense)             (None, 1)                 513

=================================================================
Total params: 4,194,449
Trainable params: 850,049
Non-trainable params: 3,344,400
_________________________________________________________________
```

the architecture of the LSTM model using Keras API. The input to the model is a sequence of integer tokens of length `sequence_len`. The integer tokens are fed into an embedding layer that maps each integer token to a dense vector of `embedding_dim`. The embedding layer is initialized with pre-trained word embeddings from the GloVe model. The embedded sequences are then fed into a 1-dimensional convolutional layer with 128 filters and kernel size 5, followed by a bidirectional LSTM layer with 128 units, a fully connected layer with 512 units and ReLU activation, a dropout layer with rate 0.5, another fully connected layer with 512 units and ReLU activation, and a final output layer with sigmoid activation that predicts the sentiment of the input text. The `model.summary()` function provides a summary of the model architecture.

```
# Optimize the model
model.compile(optimizer=Adam(learning_rate=learning_rate), loss='binary_crossentropy', metrics=['accuracy'])
```

```
# Train the LSTM Model
history = model.fit(X_train,
                    y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_data=(X_test, y_test))
```

```
Epoch 1/20
6/6 [==============================] - 62s 9s/step - loss: 0.6930 - accuracy: 0.5654 - val_loss: 0.6927 - val_accuracy: 0.5767
Epoch 2/20
6/6 [==============================] - 50s 8s/step - loss: 0.6926 - accuracy: 0.5654 - val_loss: 0.6923 - val_accuracy: 0.5767
Epoch 3/20
6/6 [==============================] - 50s 8s/step - loss: 0.6923 - accuracy: 0.5654 - val_loss: 0.6918 - val_accuracy: 0.5767
Epoch 4/20
6/6 [==============================] - 56s 9s/step - loss: 0.6919 - accuracy: 0.5654 - val_loss: 0.6914 - val_accuracy: 0.5767
Epoch 5/20
6/6 [==============================] - 50s 8s/step - loss: 0.6916 - accuracy: 0.5654 - val_loss: 0.6910 - val_accuracy: 0.5767
Epoch 6/20
6/6 [==============================] - 59s 10s/step - loss: 0.6912 - accuracy: 0.5654 - val_loss: 0.6906 - val_accuracy: 0.5767
Epoch 7/20
```

`model.compile()` compiles the model using the Adam optimizer with a specified learning rate, binary crossentropy loss function, and accuracy as the evaluation metric.

`model.fit()` trains the model using the compiled model, with the training data, batch size, number of epochs, and validation data as inputs. It stores the training history in the `history` object.

```python
# Plot train accuracy and loss
accuraties = history.history['acc']
losses = history.history['loss']
accuraties_losses = list(zip(accuraties,losses))

accuraties_losses_df = pd.DataFrame(accuraties_losses, columns={"accuraties", "losses"})

plt.figure(figsize=(10,4))
plt.suptitle("Train Accuracy vs Train Loss")
sns.lineplot(data=accuraties_losses_df)
plt.show()
```

```python
# Evaluate the model
predicted = model.predict(X_test, verbose=1, batch_size=10000)

y_predicted = [1 if each > 0.5 else 0 for each in predicted]

score, test_accuracy = model.evaluate(X_test, y_test, batch_size=10000)

print("Test Accuracy: ", test_accuracy)
print(metrics.classification_report(list(y_test), y_predicted))
```

```python
# Plot confusion matrix
conf_matrix = metrics.confusion_matrix(y_test, y_predicted)

fig, ax = plt.subplots()
sns.heatmap(conf_matrix, cbar=False, cmap='Reds', annot=True, fmt='d')
ax.set(xlabel="Predicted Value", ylabel="True Value", title="Confusion Matrix")
ax.set_yticklabels(labels=['0', '1'], rotation=0)
plt.show()
```

A Brief Summary - LSTM

1. Importing the necessary libraries and loading the dataset.
2. Next, the dataset is preprocessed by removing unwanted characters, URLs, and stop words, and by performing stemming and tokenization.
3. The preprocessed dataset is split into train and test sets.
4. The code loads pre-trained GloVe embeddings, which are used to represent the words in the tweets.
5. A LSTM model is defined and compiled.
6. The model is trained on the train set and evaluated on the test set.
7. The accuracy and loss of the model are plotted over the training epochs.
8. The model is evaluated on the test set and the accuracy, classification report, and confusion matrix are displayed.

Summary

LSTM Model showed better accuracy compared to BERT Model also the time taken by the BERT model is comparatively more than LSTM Model.

LSTM model is the better choice in this case.

BERT is likely to perform better than LSTM on sentiment analysis tasks if a large amount of high-quality labeled data is available for fine-tuning. However, with the data is limited or noisy, LSTM may be a better choice as it is less complex and easier to train.

GITHUB Links

https://github.com/RohanJJ/NLP-with-Disaster-Tweets