Disaster Tweets Classification with RNN – GRU/ BiLSTM/ BERT/ USE

MARCH 31, 2023MARCH 31, 2023 / SANDIPAN DEY

This problem appeared in a project in the coursera course Deep Learning (by the University of Colorado Boulder) and also as a past Kaggle competition.

Brief description of the problem and data

In this project, we shall build a machine learning model that predicts which Tweets are about real disasters and which one's aren't. We shall have access to a dataset of 10,000 tweets that were hand-classified apriori, hence contain groud-truth labels. We shall use a **binary text classification model** which will be trained on these tweets and then later will be used to predict the class labels for an unseen test data.

Given a train and a test csv file, where each sample in the train and test set has the following information:

- The text of a tweet
- A keyword from that tweet (although this may be blank!)
- The location the tweet was sent from (may also be blank)

We shall predict whether a given tweet is about a real disaster or not. If so, predict a 1. If not, predict a 0.

Twitter has become an important communication channel in times of emergency. The ubiquitousness of smartphones enables people to announce an emergency they're observing in real-time. Because of this, more agencies are interested in programatically monitoring Twitter (i.e. disaster relief organizations and news agencies). But, it's not always clear whether a person's words are actually announcing a disaster. That's where the classifier will be useful.

Exploratory Data Analysis (EDA) — Inspect, Visualize and Clean the Data

First we need to import all `python` packages / functions (need to install with `pip` if some of them are not already installed) that are required to the clean the texts (from the tweets), for building the RNN models and for visualization. We shall use `tensorflow / keras` to to train the deep learning models.

```
import numpy as np
import pandas as pd
import os, math

#for visualization
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud, STOPWORDS

#for text cleaning
import string, re
import nltk
nltk.download('punkt')
nltk.download('stopwords')
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

#for data analysis and modeling
import tensorflow as tf
import tensorflow_hub as hub
# !pip install tensorflow_text
import tensorflow_text
from tensorflow.keras.preprocessing import tex
from tensorflow.keras.layers import Dropout
from tensorflow.keras.metrics import Recall
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, GRU,
from tensorflow.keras.layers import TextVector
tf.__version__
# 2.12.0

from sklearn.model_selection import train_test
from sklearn.utils.class_weight import compute
```

Read the train and test dataframe, the only columns that we shall use are *text* (to extract input features) and *target* (output to predict).

```
df_train = pd.read_csv('nlp-getting-started/tr
df_test = pd.read_csv('nlp-getting-started/tes
df_train.head()
```

	keyword	location	text	target
id				
1	NaN	NaN	Our Deeds are the Reason of this #earthquake M…	1
4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
5	NaN	NaN	All residents asked to ‘shelter in place’ are …	1
6	NaN	NaN	13,000 people receive #wildfires evacuation or…	1
7	NaN	NaN	Just got sent this photo from Ruby #Alaska as …	1

There around 7.6k tweets in the training and 3.2k tweets in the test dataset, respectively.

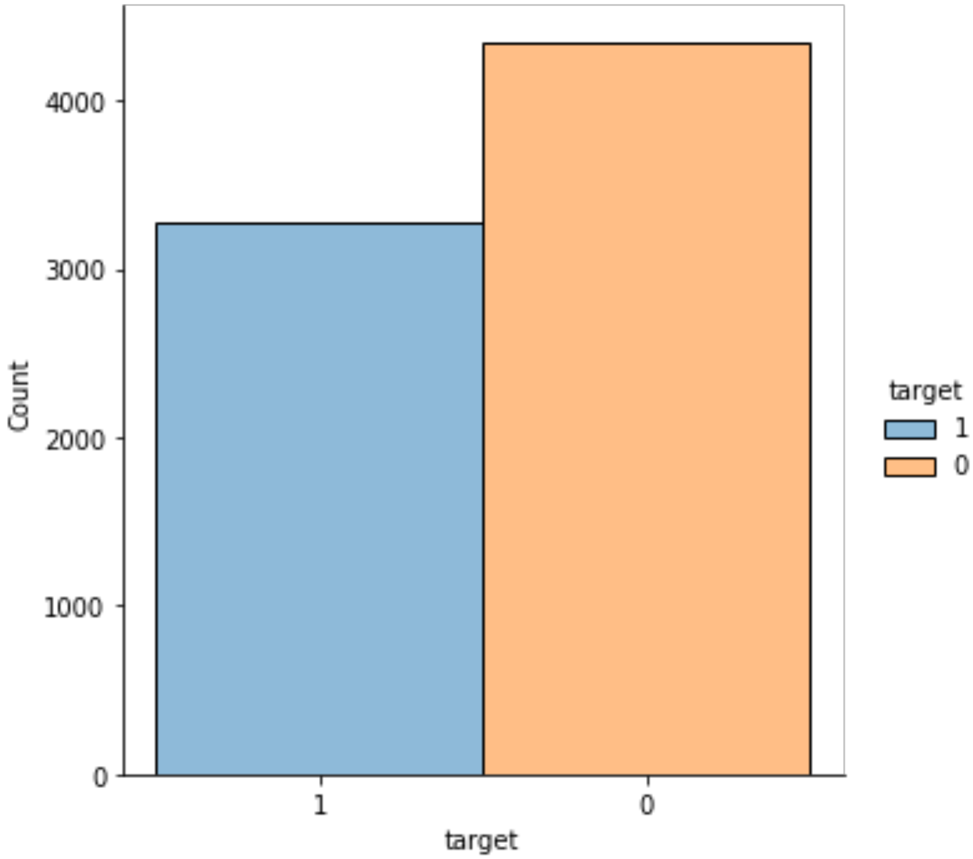
```
df_train.shape, df_test.shape
# ((7613, 4), (3263, 3))
```

Maximum number of words present in a tweet is 31, for both training and test dataset

```
max_len_train = max(df_train['text'].apply(lam
max_len_test = max(df_train['text'].apply(lamb
max_len_train, max_len_test
# (31, 31)
```

The following plot shows histogram of class labels, the number of positive (disaster) and negative (no distaster) classes in the training dataset. As can be seen, the dataset is slightly imbalanced.

```
#train_df['target'] = train_df['target'].astyp
sns.displot(data=train_df, x='target', hue='ta
train_df['target'].value_counts()

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



Now, let's use the wordcloud library to find the most frequent words in disaster tweets and normal tweets. As we can see,

- the top 10 most frequent words in disaster tweets (with class label 1) are: 'fire', 'New', 'via', 'disaster', 'California', 'suicide', 'U', 'police', 'amp', 'people'
- the top 10 most frequent words in the normal tweets (with class label 0) are: 'new', 'amp', 'u', 'one', 'body', 'time', 'video', 'via', 'day', 'love'

```
def plot_wordcloud(text, title, k=10):
  # Create and Generate a Word Cloud Image
  wordcloud = WordCloud(width = 3000, height =
  # top k words
  plt.figure(figsize=(10,5))
  print(f'top {k} words: {list(wordcloud.words
  ax = sns.barplot(x=0, y=1, data=pd.DataFrame
  ax.set(xlabel = 'words', ylabel='count', tit
  plt.show()
  #Display the generated image
  plt.figure(figsize=(15,15))
  plt.imshow(wordcloud, interpolation="bilinea
  plt.show()

plot_wordcloud(' '.join(df_train[df_train['tar
plot_wordcloud(' '.join(df_train[df_train['tar
```

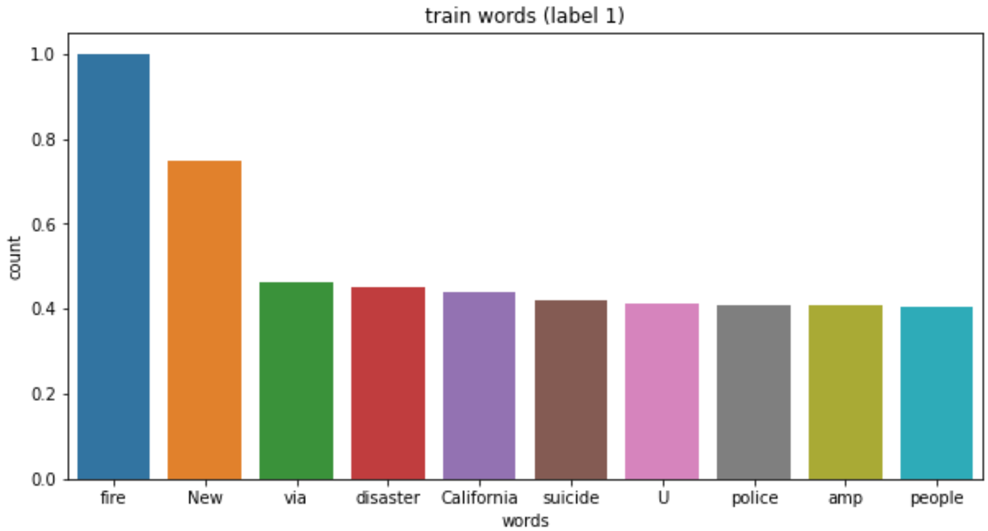

Now, let's use the wordcloud library to find the most frequent words in disaster tweets and normal tweets. As we can see,

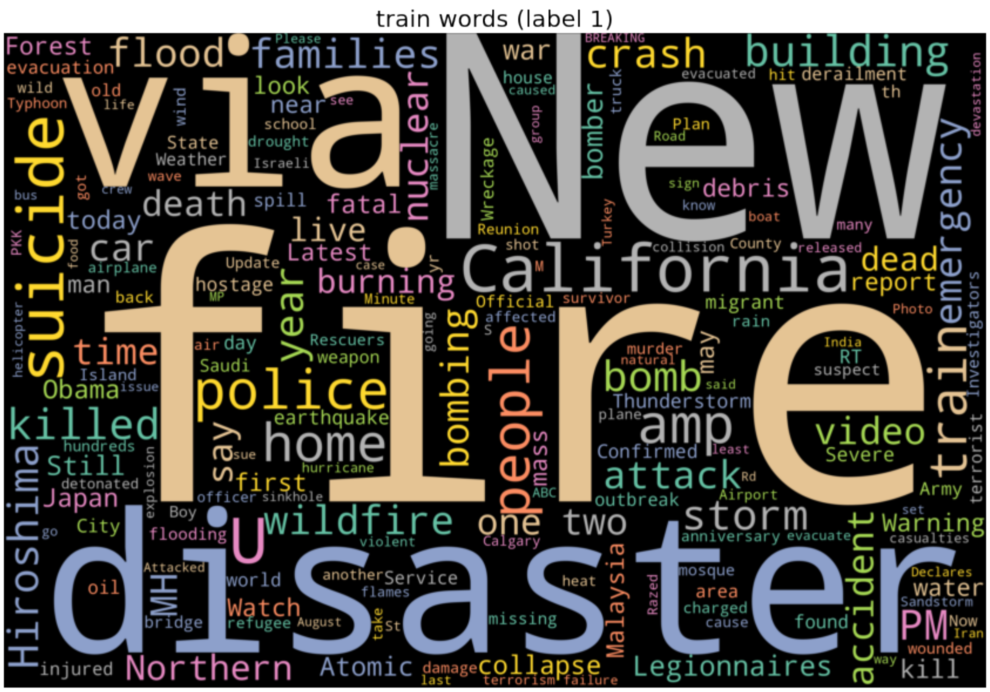
- the top 10 most frequent words in disaster tweets (with class label 1) are: 'fire', 'New', 'via', 'disaster', 'California', 'suicide', 'U', 'police', 'amp', 'people'

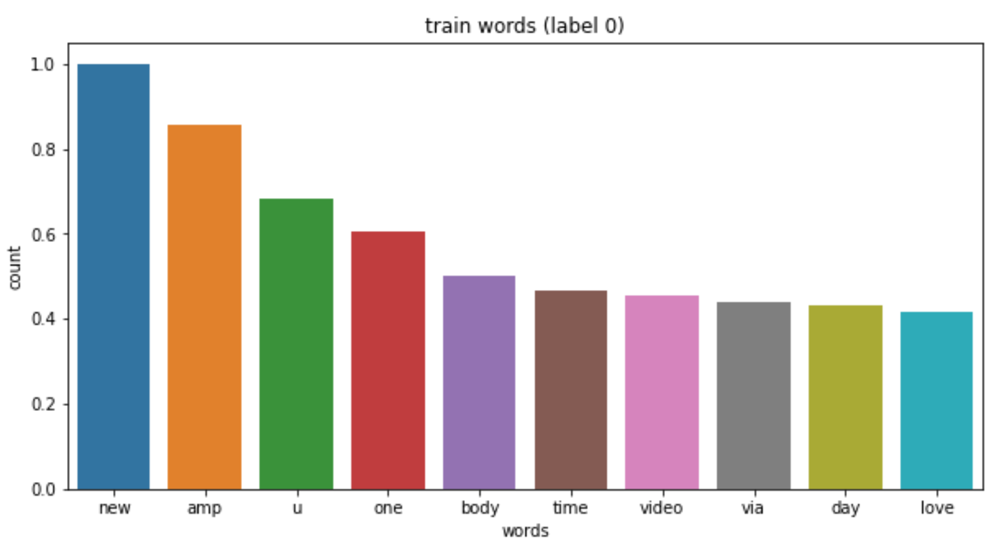
- the top 10 most frequent words in the normal tweets (with class label 0) are: 'new', 'amp', 'u', 'one', 'body', 'time', 'video', 'via', 'day', 'love'

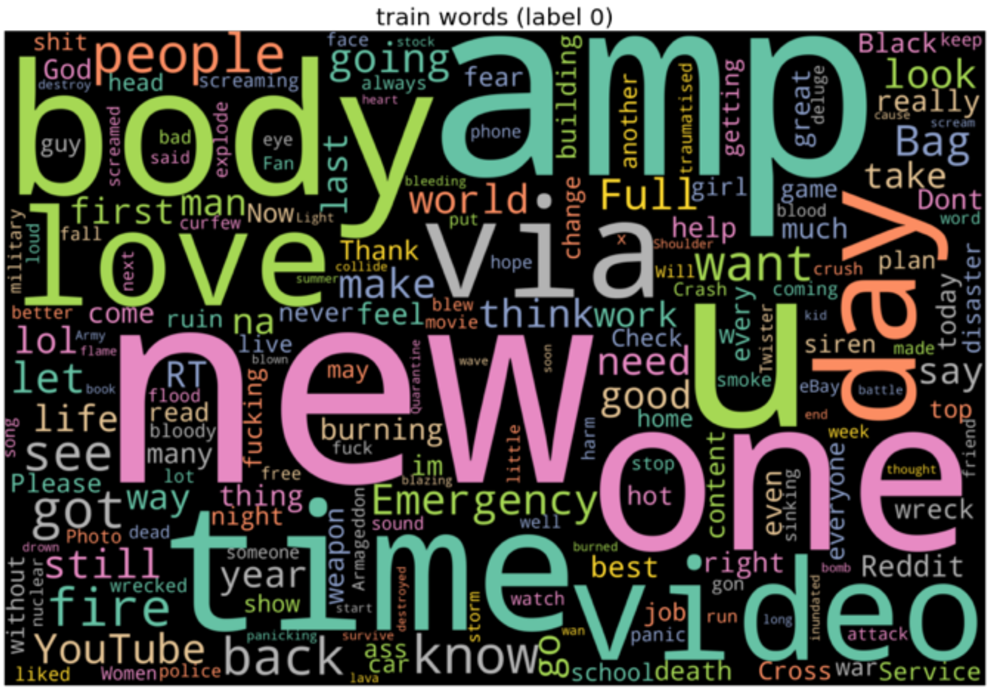
```
def plot_wordcloud(text, title, k=10):
  # Create and Generate a Word Cloud Image
  wordcloud = WordCloud(width = 3000, height =
  # top k words
  plt.figure(figsize=(10,5))
  print(f'top {k} words: {list(wordcloud.words
  ax = sns.barplot(x=0, y=1, data=pd.DataFrame
  ax.set(xlabel = 'words', ylabel='count', tit
  plt.show()
  #Display the generated image
  plt.figure(figsize=(15,15))
  plt.imshow(wordcloud, interpolation="bilinea
  plt.show()

plot_wordcloud(' '.join(df_train[df_train['tar
plot_wordcloud(' '.join(df_train[df_train['tar
```

```
top 10 words: ['fire', 'New', 'via',
'disaster', 'California', 'suicide', 'U',
'police', 'amp', 'people']
```


train words (label 1)
count
1.0
0.8
0.6
0.4
0.2
0.0
fire
New
via
disaster
California
suicide
U
police
amp
people
words


train words (label 1)
fire
New
via
disaster
California
suicide
police
people
home
amp
wildfire
attack
storm
bomb
video
killed
Hiroshima
emergency
train
accident
building
crash
families
flood
nuclear
bombing
burning
death
car
time
year
live
dead
report
Northern
Atomic
collapse
Legionnaires
Japan
Obama
Malaysia
PM
Warning
water
kill


train words (label 0)
count
1.0
0.8
0.6
0.4
0.2
0.0
new
amp
u
one
body
time
video
via
day
love
words



Preprocessing / Cleaning

Since the tweet texts are likely to contain many junk characters, very common non-informative words (*stopwords*, e.g., 'the'), it is a good idea to clean the text (with the function `clean_text()` as shown below) and remove unnecessary stuffs before building the models, otherwise they can affect the performance. It's important that we apply the same preprocessing on both the training and test tweets.

```
def clean_text(txt):
    """"""
    cleans the input text by following the ste
    * replace contractions
    * remove punctuation
    * split into words
    * remove stopwords
    * remove leftover punctuations
    """"""
    contraction_dict = {"ain't": "is not", "ar
                        "could've": "could hav
```

```
                        "doesn't": "does not",
                        "haven't": "have not",
                        "you'd've": "you would
    def _get_contractions(contraction_dict):
        contraction_re = re.compile('(%s)' % '
        return contraction_dict, contraction_r

    def replace_contractions(text):
        contractions, contractions_re = _get_c
        def replace(match):
            return contractions[match.group(0)
        return contractions_re.sub(replace, te

    # replace contractions
    txt = replace_contractions(txt)

    #remove punctuations
    txt  = "".join([char for char in txt if ch
    #remove numbers
    txt = re.sub('[0-9]+', '', txt)
    #txt = txt.str.replace(r"[^A-Za-z0-9()!?\'
    txt = txt.str.lower() # lowercase
    txt = txt.str.replace(r"\#","", regex = Tr
    txt = txt.str.replace(r"http\S+","URL", re
    txt = txt.str.replace(r"@","", regex = Tru
    txt = txt.str.replace("\s{2,}", " ", regex

    # split into words
    words = word_tokenize(txt)

    # remove stopwords
    stop_words = set(stopwords.words('english'
    words = [w for w in words if not w in stop

    # removing leftover punctuations
    words = [word for word in words if word.is

    cleaned_text = ' '.join(words)
    return cleaned_text

# clean train and test tweets
df_train['text'] = df_train['text'].apply(lamb
```

```
df_test['text'] = df_test['text'].apply(lambda

df_train.head()

# CPU times: user 2.05 s, sys: 101 ms, total:
# Wall time: 2.16 s
```

	keyword	location	text	target
id				
1	NaN	NaN	Our Deeds Reason earthquake May ALLAH Forgive us	1
4	NaN	NaN	Forest fire near La Ronge Sask Canada	1
5	NaN	NaN	All residents asked shelter place notified off…	1
6	NaN	NaN	people receive wildfires evacuation orders Cal…	1
7	NaN	NaN	Just got sent photo Ruby Alaska smoke wildfire…	1

Model Architecture

We shall use multiple models, starting from LSTM/GRU/BiLSTM to BERT and USE.

LSTM / GRU

Let's start with vanilla LSTM / GRU model. We need to start by tokenizing the texts followed adding appropriate pads to the token sequence (to have the seuqence length fixed, e.g. equal to `max_len`)

```
xtrain, xtest, ytrain, ytest = train_test_spli

max_len = max(df_train['text'].apply(lambda x:
max_words = 20000
tokenizer = text.Tokenizer(num_words = max_wor
# create the vocabulary by fitting on x_train
tokenizer.fit_on_texts(xtrain)
# generate the sequence of tokens
xtrain_seq = tokenizer.texts_to_sequences(xtra
xtest_seq = tokenizer.texts_to_sequences(xtest

# pad the sequences
xtrain_pad = sequence.pad_sequences(xtrain_seq
xtest_pad = sequence.pad_sequences(xtest_seq,
word_index = tokenizer.word_index

print('text example:', xtrain[0])
print('sequence of indices(before padding):',
print('sequence of indices(after padding):', x

# text example: Witness video shows car explod
# MikeCroninWMUR
# sequence of indices(before padding): [17, 29
# sequence of indices(after padding): [ 0  0
```


We shall first use a pretrained (semantic) embedding from *Global Vectors for Word Representation* (**GloVe**) model (dowload the pretrained weights) and create a word-level embedding matrix as shown below. Later we shall use `LSTM` to train the embedding on our own.

```
#https://nlp.stanford.edu/projects/glove/
!wget https://nlp.stanford.edu/data/glove.6B.z
!unzip g*zip
```

```
%%time
embedding_vectors = {}
with open('glove.6B.300d.txt','r',encoding='ut
    for row in file:
        values = row.split(' ')
        word = values[0]
        weights = np.asarray([float(val) for v
        embedding_vectors[word] = weights
print(f"Size of vocabulary in GloVe: {len(embe

# Size of vocabulary in GloVe: 400000
# CPU times: user 33.1 s, sys: 1.55 s, total:
# Wall time: 33.4 s
```

```
#initialize the embedding_matrix with zeros
emb_dim = 300
vocab_len = max_words if max_words is not None
embedding_matrix = np.zeros((vocab_len, emb_di
oov_count = 0
oov_words = []
for word, idx in word_index.items():
    if idx < vocab_len:
        embedding_vector = embedding_vectors.g
        if embedding_vector is not None:
            embedding_matrix[idx] = embedding_
        else:
            oov_count += 1
            oov_words.append(word)
#print some of the out of vocabulary words
print(f'Some out of valubulary words: {oov_wor
print(f'{oov_count} out of {vocab_len} words w

# Some out of valubulary words: []
# 0 out of 50 words were OOV.
```

Let's create the model with and `Embedding` layer followed by the `LSTM` layer and add a bunch of `Dense` layers on top. We shall first use pretrained GloVe embeddings and then later build another model to train the embeddings from the data provided.

```
model_lstm = Sequential(name='model_lstm')
model_lstm.add(Embedding(vocab_len, emb_dim, t
#model_lstm.add(Embedding(vocab_len, emb_dim,
model_lstm.add(LSTM(64, activation='tanh', ret
model_lstm.add(Dense(128, activation='relu'))
#model_lstm.add(tf.keras.layers.BatchNormaliza
model_lstm.add(Dropout(0.2)) # Adding Dropout
model_lstm.add(Dense(256, activation='relu'))
model_lstm.add(Dense(128, activation='relu'))
model_lstm.add(Dense(64, activation='relu'))
model_lstm.add(Dense(1, activation='sigmoid'))
model_lstm.compile(loss='binary_crossentropy',
model_lstm.summary()
```

```
Model: "model_lstm"
_______________________________________________
 Layer (type)                Output Shape
===============================================
 embedding_3 (Embedding)     (None, None, 300)

 lstm_2 (LSTM)               (None, 64)

 dense_7 (Dense)             (None, 128)

 dropout_3 (Dropout)         (None, 128)

 dense_8 (Dense)             (None, 256)

 dense_9 (Dense)             (None, 128)

 dense_10 (Dense)            (None, 64)

 dense_11 (Dense)            (None, 1)

===============================================
Total params: 6,176,001
```

```
Trainable params: 6,176,001
Non-trainable params: 0
_________________________________________________________________
None
```

Now, let's create the model using `GRU` layer instead of LSTM, as shown in the following code snippet.

```
emb_dim = embedding_matrix.shape[1]
model_gru = Sequential(name='model_gru')
model_gru.add(Embedding(vocab_len, emb_dim, tr
model_gru.add(GRU(128, return_sequences=False)
model_gru.add(Dropout(0.5))
model_gru.add(Dense(1, activation = 'sigmoid')
model_gru.compile(loss='binary_crossentropy',
model_gru.summary()

Model: "model_gru"
_________________________________________________________________
 Layer (type)                Output Shape
=================================================================
 embedding_4 (Embedding)     (None, None, 300)

 gru_1 (GRU)                 (None, 128)

 dropout_4 (Dropout)         (None, 128)

 dense_12 (Dense)            (None, 1)

=================================================================
Total params: 6,165,249
Trainable params: 165,249
Non-trainable params: 6,000,000
_________________________________________________________________
None
```

BiLSTM

Now, let's create a Bidirection LSTM model instead, this time using `TextVectorization`: a preprocessing layer which maps text features to integer sequences. Let's create training and validation datasets for model evaluation, by applying the vectorizer on the text tweets.

```
# Define Embedding layer as pre-processing lay
max_features = 20000 #  20000 most frequent wo

vectorizer = TextVectorization(max_tokens=max_
vectorizer.adapt(np.hstack((X_train, X_test)))
vectorizerd_text = vectorizer(X_train)

dataset = tf.data.Dataset.from_tensor_slices((
dataset = dataset.cache()
dataset = dataset.shuffle(160000)
dataset = dataset.batch(32)
dataset = dataset.prefetch(8)
batch_X, batch_y = dataset.as_numpy_iterator()

train = dataset.take(int(len(dataset)*.8))
val = dataset.skip(int(len(dataset)*.8)).take(

model_bilstm = Sequential(name='model_bilstm')
model_bilstm.add(Embedding(max_features + 1, 6
model_bilstm.add(Bidirectional(LSTM(64, activa
model_bilstm.add(Dense(128, activation='relu')
model_bilstm.add(Dropout(0.2)) # Adding Dropou
model_bilstm.add(Dense(256, activation='relu')
model_bilstm.add(Dense(128, activation='relu')
model_bilstm.add(Dense(64, activation='relu'))
model_bilstm.add(Dense(1, activation='sigmoid'
model_bilstm.compile(loss='BinaryCrossentropy'
model_bilstm.summary()
```

```
Model: "model_bilstm"
```

```
_________________________________________________________________
 Layer (type)                Output Shape
=================================================================
 embedding_1 (Embedding)     (None, None, 64)

 bidirectional_1 (Bidirectio  (None, 128)
 nal)

 dense_5 (Dense)             (None, 128)

 dropout_1 (Dropout)         (None, 128)

 dense_6 (Dense)             (None, 256)

 dense_7 (Dense)             (None, 128)

 dense_8 (Dense)             (None, 64)

 dense_9 (Dense)             (None, 1)

=================================================================
Total params: 1,436,865
Trainable params: 1,436,865
Non-trainable params: 0
_________________________________________________________________
```

BERT

Next, let's use the *Bidirectional Encoder Representations from Transformers* (BERT) model for the text classification. The function `get_BERT_model()` uses the BERT model as backbone, extracts the *pooled_output* layer and adds a couple of `Dense` layers (with `Dropout` regularizer) on top of it, as shown in thee next code snippet.

```
def get_BERT_model():
    # Preprocessing
    tfhub_handle_preprocess = 'https://tfhub.d
    # Bert encoder
    tfhub_handle_encoder = 'https://tfhub.dev/
    bert_preprocess_model = hub.KerasLayer(tfh
    bert_model = hub.KerasLayer(tfhub_handle_e
    input_layer = tf.keras.layers.Input(shape=
    x = bert_preprocess_model(input_layer)
    x = bert_model(x)['pooled_output']
    x = tf.keras.layers.Dropout(0.5)(x) #Optio
    x = tf.keras.layers.Dense(256, activation=
    classification_out = tf.keras.layers.Dense
    bert_preprocess_model._name = "preprocess"
    bert_model._name = "bert_encoder"
    model_bert = tf.keras.Model(input_layer, c
    model_bert._name = "model_bert"
    return model_bert

model_bert = get_BERT_model()
model_bert.summary()
```

```
Model: "model_bert"
______________________________________________
 Layer (type)                   Output Shape
==============================================
 tweets (InputLayer)            [(None,)]

 preprocess (KerasLayer)        {'input_type_i
                                (None, 128),
                                 'input_mask':
                                e, 128),
                                 'input_word_i
                                (None, 128)}

 bert_encoder (KerasLayer)      {'pooled_outpu
                                None, 128),
                                 'sequence_out
                                 (None, 128, 1
                                 'encoder_outp
                                 [(None, 128,
```

```
                                      (None, 128, 1
                                      'default': (N
                                     128)}

 dropout_1 (Dropout)                 (None, 128)

 dense_1 (Dense)                     (None, 256)

 classifier (Dense)                  (None, 1)

==============================================
Total params: 4,419,202
Trainable params: 33,281
Non-trainable params: 4,385,921
______________________________________________
```

Universal Sequence Encoder Model (USE)

Finally, we shall use the *Universal Sentence Encoder* to obtain sentence level embedding, along with our regular `Dense` layers to create a binary text classification model.

```
transfer_model_url = 'https://tfhub.dev/google
sentence_encoder_layer = hub.KerasLayer("https
                                        input_
                                        dtype=
                                        traina
                                        # keep
                                        name="

model_use = tf.keras.Sequential([
    sentence_encoder_layer,
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(16, activation="relu
    tf.keras.layers.Dense(16, activation="relu
    tf.keras.layers.Dense(1, activation="sigmo
], name = 'transfer_mode')
model_use.summary()
```

```
Model: "transfer_mode"
_________________________________________________
 Layer (type)                Output Shape
=================================================
 USE (KerasLayer)            (None, 512)

 dropout_9 (Dropout)         (None, 512)

 dense_13 (Dense)            (None, 16)

 dense_14 (Dense)            (None, 16)

 dense_15 (Dense)            (None, 1)

=================================================
Total params: 256,806,321
Trainable params: 8,497
Non-trainable params: 256,797,824
_________________________________________________
```

Results and Analysis

Let's now fit the models on the training dataset and compare the performance of the model (in terms of accuracy, recall and ROC AUC) on the held-out validation daatset. The metric *Recall* is more important than *precision / accuracy* here becuase we shall like our model to capture as many of the true disaster tweets as possibile.

LSTM / GRU

The LSTM model was trained for 50 epochs (10 epochs are shown below) and the accuracy did not seem to improve over time (obtained ~66% accuracy on validation).

Hyperparameter Tuning

- Number of LSTM units and batch size were varied to see the impact on performance, but the model did almost the same.
- First the model was trained with pe-trained **GloVe** `Embedding` layers and then later the `Embedding` layer was trained from the data, but the accuracies did not improve much.

```
# model_lstm.add(Embedding(vocab_len, emb_dim,
# with pretrained GloVe weights
%%time
batch_size = 32
epochs  = 50
history = model_lstm.fit(xtrain_pad, np.asarra

Epoch 1/10
24/24 [==============================] - 9s 31
Epoch 2/10
24/24 [==============================] - 0s 9m
Epoch 3/10
24/24 [==============================] - 0s 8m
Epoch 4/10
24/24 [==============================] - 0s 8m
Epoch 5/10
24/24 [==============================] - 0s 10
Epoch 6/10
24/24 [==============================] - 0s 10
Epoch 7/10
24/24 [==============================] - 0s 12
Epoch 8/10
24/24 [==============================] - 0s 10
Epoch 9/10
24/24 [==============================] - 0s 11
Epoch 10/10
24/24 [==============================] - 0s 10
CPU times: user 5.78 s, sys: 719 ms, total: 6.
Wall time: 12.4 s
```

```
# model_lstm.add(Embedding(vocab_len, emb_dim,
# learning the embedding layer weights
%%time
batch_size = 32
epochs  = 50
history = model_lstm.fit(xtrain_pad, np.asarra
```

```
Epoch 1/50
191/191 [==============================] - 8s
Epoch 2/50
191/191 [==============================] - 2s
Epoch 3/50
191/191 [==============================] - 2s
Epoch 4/50
191/191 [==============================] - 2s
Epoch 5/50
191/191 [==============================] - 2s
Epoch 6/50
191/191 [==============================] - 2s
Epoch 7/50
191/191 [==============================] - 2s
Epoch 8/50
191/191 [==============================] - 2s
Epoch 9/50
191/191 [==============================] - 2s
Epoch 10/50
191/191 [==============================] - 2s
Epoch 11/50
191/191 [==============================] - 2s
Epoch 12/50
191/191 [==============================] - 2s
```

The GRU model was trained for 50 epochs (12 epochs are shown below) and the accuracy did not seem to improve over time (obtained ~67% accuracy on validation).

```
batch_size = 32
epochs  = 50
history = model_gru.fit(xtrain_pad, np.asarray

Epoch 1/10
24/24 [==============================] - 4s 27
Epoch 2/10
24/24 [==============================] - 0s 11
Epoch 3/10
24/24 [==============================] - 0s 10
Epoch 4/10
24/24 [==============================] - 0s 8m
Epoch 5/10
24/24 [==============================] - 0s 8m
Epoch 6/10
24/24 [==============================] - 0s 8m
Epoch 7/10
24/24 [==============================] - 0s 7m
Epoch 8/10
24/24 [==============================] - 0s 7m
Epoch 9/10
24/24 [==============================] - 0s 10
Epoch 10/10
24/24 [==============================] - 0s 10
```

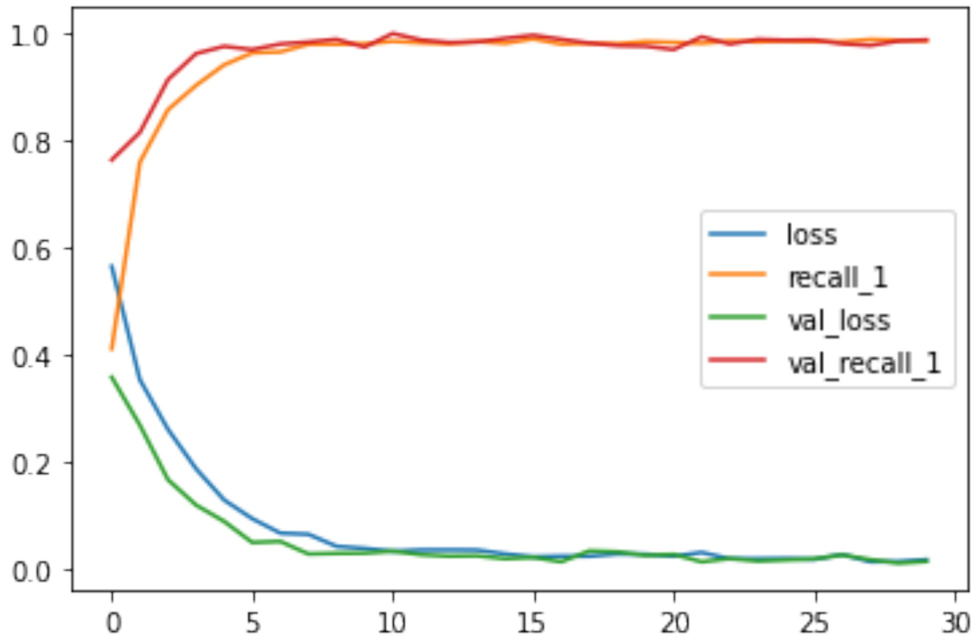
BiLSTM

This model was trained with `TextVectorization` as preprocessing layer. This time recall was used as evaluation metric. This model performed quite well an achived over 98% validation recall, as shown in the next figure too. This model is the second best performing model (in terms of bulic score) on the unseen test dataset.

```
hist= model_bilstm.fit(train, epochs=30, batch

Epoch 1/30
166/166 [==============================] - 29s
Epoch 2/30
166/166 [==============================] - 6s
Epoch 3/30
166/166 [==============================] - 5s
Epoch 4/30
166/166 [==============================] - 4s
Epoch 5/30
166/166 [==============================] - 4s
Epoch 6/30
166/166 [==============================] - 3s
Epoch 7/30
166/166 [==============================] - 4s
Epoch 8/30
166/166 [==============================] - 4s
Epoch 9/30
166/166 [==============================] - 6s
Epoch 10/30
166/166 [==============================] - 5s
Epoch 11/30
166/166 [==============================] - 3s
Epoch 12/30
166/166 [==============================] - 4s
Epoch 13/30
166/166 [==============================] - 3s
Epoch 14/30
166/166 [==============================] - 3s
Epoch 15/30
166/166 [==============================] - 4s
Epoch 16/30
166/166 [==============================] - 4s
Epoch 17/30
166/166 [==============================] - 3s
Epoch 18/30
166/166 [==============================] - 4s
Epoch 19/30
166/166 [==============================] - 3s
Epoch 20/30
```

```
166/166 [==============================] - 3s
Epoch 21/30
166/166 [==============================] - 4s
Epoch 22/30
166/166 [==============================] - 4s
Epoch 23/30
166/166 [==============================] - 5s
Epoch 24/30
166/166 [==============================] - 5s
Epoch 25/30
166/166 [==============================] - 6s
Epoch 26/30
166/166 [==============================] - 4s
Epoch 27/30
166/166 [==============================] - 3s
Epoch 28/30
166/166 [==============================] - 3s
Epoch 29/30
166/166 [==============================] - 3s
Epoch 30/30
166/166 [==============================] - 3s
```

```
plt.figure(figsize=(10, 6))
pd.DataFrame(hist.history).plot()
plt.show()
```



Predictions

Before computing the prediction, we need to preprocess the test tweets by applying `TextVectorization`.

```
vectorizerd_test_text = vectorizer(X_test)
preds = []
for input_text in vectorizerd_test_text:
    pred = model.predict(np.expand_dims(input_
    preds.append(pred)

preds = np.round(np.array(preds))
sub_sample = pd.read_csv('sample_submission.cs
sub_sample['target'] = preds.flatten()
sub_sample['target'] = sub_sample['target'].as
sub_sample.to_csv('submission.csv', index=Fals
```

BERT

Since the training data is a little imbalanced, we shall compute the class weights and use them in the loss function to compensate the imbalance.

```
class_weights = compute_class_weight(class_wei
                                     classes =
                                     y= df_tra
class_weights = {k:class_weights[k] for k in n
class_weights
# {0: 0.8766697374481806, 1: 1.163711403240599
```

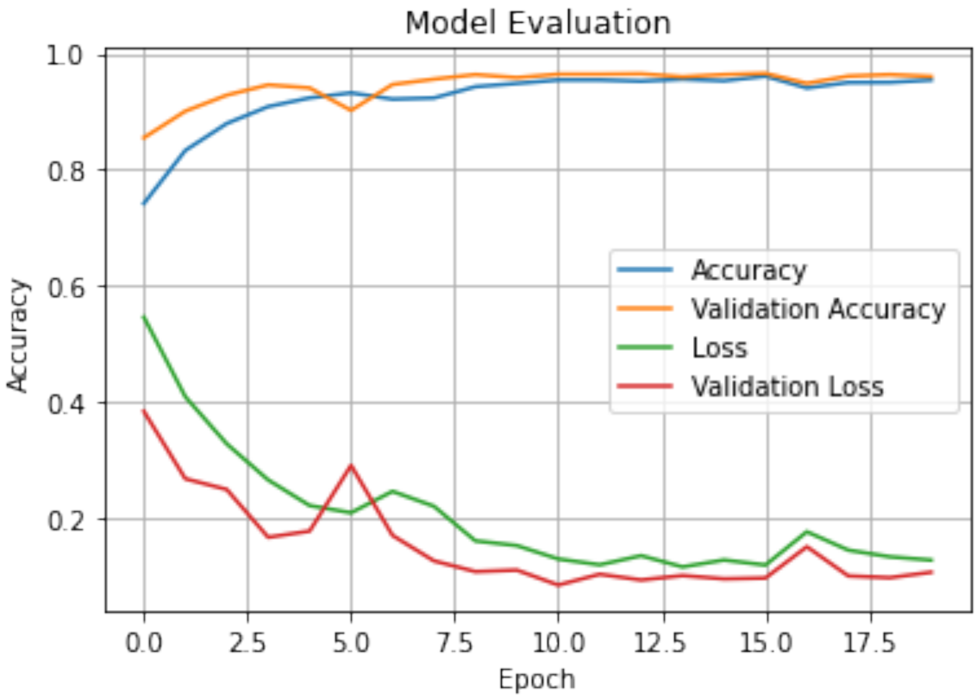
The model was trained for 20 epochs with `Adam` optimizer and weighted `BCE` loss function. We can change the optimizer and use `AdamW` or `SGD` instead and observe the result on hyperparameter tuning. This model happened to be a competitor of the BiLSTM model above, in terms of performance score obtained on the unseen test data.

```
epochs = 20
batch_size = 32

optimizer = tf.keras.optimizers.Adam(learning_

loss = tf.keras.losses.BinaryCrossentropy(from
metrics = [tf.keras.metrics.BinaryAccuracy(),

model_bert.get_layer('bert_encoder').trainable

model_bert.compile(optimizer=optimizer, loss=l

train_data = df_train.sample(frac=0.8,random_s
```

```
valid_data = df_train.drop(train_data.index)

history = model_bert.fit(x=df_train.text.value
                                          y=df
                                          clas
                                          epoc
                                          batc
                                          vali

Epoch 1/20
238/238 [==============================] - 58s
Epoch 2/20
238/238 [==============================] - 31s
Epoch 3/20
238/238 [==============================] - 28s
Epoch 4/20
238/238 [==============================] - 27s
Epoch 5/20
238/238 [==============================] - 27s
Epoch 6/20
238/238 [==============================] - 28s
Epoch 7/20
238/238 [==============================] - 28s
Epoch 8/20
238/238 [==============================] - 28s
Epoch 9/20
238/238 [==============================] - 27s
Epoch 10/20
238/238 [==============================] - 26s
Epoch 11/20
238/238 [==============================] - 26s
Epoch 12/20
238/238 [==============================] - 27s
Epoch 13/20
238/238 [==============================] - 28s
Epoch 14/20
238/238 [==============================] - 28s
Epoch 15/20
238/238 [==============================] - 26s
Epoch 16/20
238/238 [==============================] - 26s
Epoch 17/20
```

```
238/238 [==============================] - 27s
Epoch 18/20
238/238 [==============================] - 27s
Epoch 19/20
238/238 [==============================] - 25s
Epoch 20/20
238/238 [==============================] - 26s
```

```
def plot_hist(hist):
    '''
    Plots the training / validation loss and a
    '''
    plt.plot(hist.history["binary_accuracy"])
    plt.plot(hist.history['val_binary_accuracy
    plt.plot(hist.history['loss'])
    plt.plot(hist.history['val_loss'])
    plt.title("Model Evaluation")
    plt.ylabel("Accuracy")
    plt.xlabel("Epoch")
    plt.legend(["Accuracy","Validation Accurac
    plt.grid()
    plt.show()

plot_hist(history)
```



Prediction on the test dataset

```
X_test = df_test["text"].values
predictions_prob = model_bert.predict(X_test)
predictions = tf.round(predictions_prob)
submission = pd.read_csv('nlp-getting-started/
submission['target'] = predictions
submission['target'] =submission['target'].ast
submission['id'] = df_test.index
submission.to_csv('submission2.csv', index=Fal
submission.head()

102/102 [==============================] - 7s
```

	id	target
0	0	0

1	2	1
2	3	1
3	9	1
4	11	1

Model USE

Finally, the *Universal Sentence Embedding* model was trained, it outperformed all the models and obtained more than 80% public score on *Kaggle* on the test dataset.

```
X, y = df_train['text'].values, df_train['targ
X_train, X_val, y_train, y_val = train_test_sp
X.shape, y.shape
# ((7613,), (7613,))

model_use.compile(loss = tf.keras.losses.Binar
              optimizer=tf.keras.optimizers.Ada
              metrics=['accuracy',tf.keras.metr
```

```
%%time
history = model_use.fit(X_train, y_train, epoc

Epoch 1/10
179/179 [==============================] - 8s
Epoch 2/10
179/179 [==============================] - 3s
Epoch 3/10
179/179 [==============================] - 3s
Epoch 4/10
179/179 [==============================] - 3s
Epoch 5/10
179/179 [==============================] - 4s
Epoch 6/10
179/179 [==============================] - 4s
Epoch 7/10
179/179 [==============================] - 3s
Epoch 8/10
179/179 [==============================] - 3s
Epoch 9/10
179/179 [==============================] - 3s
Epoch 10/10
179/179 [==============================] - 4s
CPU times: user 41.2 s, sys: 3.48 s, total: 44
Wall time: 36.5 s
```

```
def plot_hist(hist):
    '''
    Plots the training / validation loss and a
    '''
    plt.plot(hist.history["accuracy"])
    plt.plot(hist.history['val_accuracy'])
    plt.plot(hist.history['loss'])
    plt.plot(hist.history['val_loss'])
    plt.title("Model Evaluation")
    plt.ylabel("Accuracy")
    plt.xlabel("Epoch")
    plt.legend(["Accuracy","Validation Accurac
    plt.grid()
    plt.show()

plot_hist(history)
```

Prediction and Submission to Kaggle

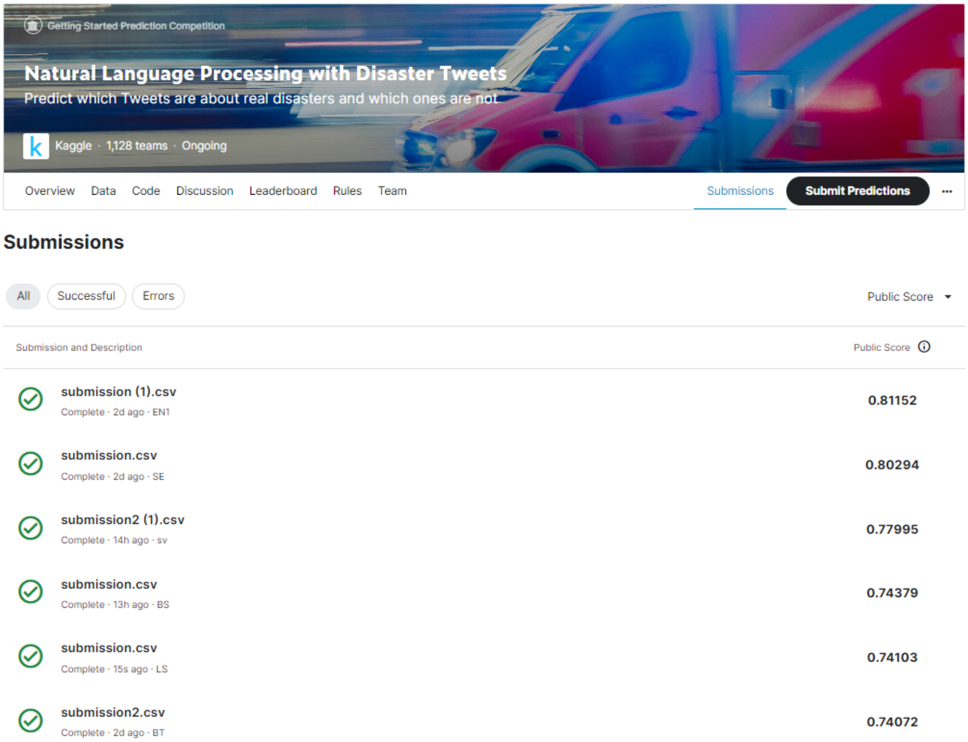
```
X_test = df_test['text'].values
predictions_prob = model_use.predict(X_test)
predictions = tf.round(predictions_prob)

102/102 [==============================] - 1s
```

```
submission = pd.read_csv('nlp-getting-started/
submission['target'] = predictions
submission['target'] =submission['target'].ast
submission['id'] = df_test.index
submission.to_csv('submission.csv', index=Fals
submission.head()
```

	id	target
0	0	1
1	2	1
2	3	1
3	9	1
4	11	1

Conclusion

The Sentence-level Embedding (USE) model performed the best on the test data (*Kaggle* public score ~81.1%), whereas *BiLSTM* and *BERT* models did decent jobs. Surprisingly, the USE model performed pretty well without any preprocessing. Training *BERT* for longer time may improve the accuracy of the transfomer on the test dataset. The next screenshots show the *Kaggle* public scores obtained for different submissions and the **leaderboard** position for the best sumission is **265**, as of now.



Getting Started Prediction Competition

Natural Language Processing with Disaster Tweets

Predict which Tweets are about real disasters and which ones are not

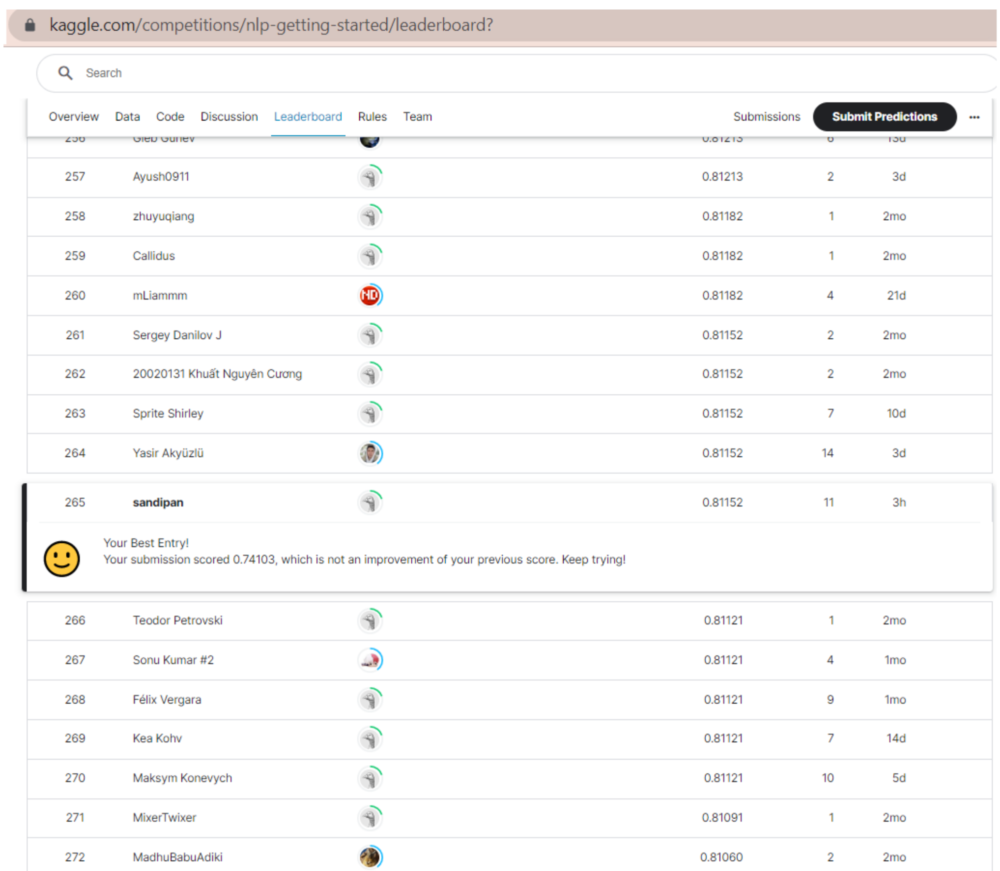
Kaggle · 1,128 teams · Ongoing

Overview Data Code Discussion Leaderboard Rules Team Submissions Submit Predictions

Submissions

All Successful Errors Public Score

Submission and Description	Public Score
submission (1).csv Complete · 2d ago · EN1	0.81152
submission.csv Complete · 2d ago · SE	0.80294
submission2 (1).csv Complete · 14h ago · sv	0.77995
submission.csv Complete · 13h ago · BS	0.74379
submission.csv Complete · 15s ago · LS	0.74103
submission2.csv Complete · 2d ago · BT	0.74072



kaggle.com/competitions/nlp-getting-started/leaderboard?

Search

Overview Data Code Discussion Leaderboard Rules Team Submissions Submit Predictions

257	Ayush0911	0.81213	2	3d
258	zhuyuqiang	0.81182	1	2mo
259	Callidus	0.81182	1	2mo
260	mLiammm	0.81182	4	21d
261	Sergey Danilov J	0.81152	2	2mo
262	20020131 Khuất Nguyên Cương	0.81152	2	2mo
263	Sprite Shirley	0.81152	7	10d
264	Yasir Akyüzlü	0.81152	14	3d
265	**sandipan**	0.81152	11	3h
266	Teodor Petrovski	0.81121	1	2mo
267	Sonu Kumar #2	0.81121	4	1mo
268	Félix Vergara	0.81121	9	1mo
269	Kea Kohv	0.81121	7	14d
270	Maksym Konevych	0.81121	10	5d
271	MixerTwixer	0.81091	1	2mo
272	MadhuBabuAdiki	0.81060	2	2mo

Your Best Entry!
Your submission scored 0.74103, which is not an improvement of your previous score. Keep trying!

AI, Data Processing, Deep Learning, ML, NLP, Text Mining