Disaster Tweets Classification

Cedric Yu[♮]

Abstract

We study the Disaster Tweets dataset in this Kaggle competition. Using various techniques in natural language processing: `CountVectorizer` and `TfidfVectorizer` from `nltk`, LSTM with GloVe embedding vectors, and a transformer network from *Hugging Face*, we classify each tweet into whether it refers to a disaster, and compare the results and performance of different approaches.

1 Problem Statement

We are tasked with classifying whether each given tweet refers to a disaster or not.

1.1 Datasets

We are given a labeled training set `train.csv` and an unlabeled test set `test.csv`. Each instance consists of the following columns:

- `id`: a unique identifier for each tweet
- `text`: the text of the tweet as a string
- `location`: the location the tweet was sent from (may be blank)
- `keyword`: a particular keyword from the tweet (may be blank)
- `target`: Boolean label denoting whether a tweet is about a real disaster (1) or not (0); in the training set only.

Each tweet text may have hashtags #, mentions @ and/or URLs (http:// or https://).

Preliminary Observations

The training set contains 7,613 tweets and The test set contains 3,263 unlabeled tweets. The target in the training set has a mean of 0.43; we have a balanced class. In the training set, there are 61 and 2,533 missing values for `keyword` and `location` respectively, while in the test set, the numbers are 26 and 1,105 respectively. Due to the large number of missing values in `location` (and the messiness of the non-missing values), we drop this column from now on. Examples of `text` are

```
train['text'].iloc[36]
"@PhDSquares #mufc they've built so much hype around new acquisitions but I doubt they
    will set the EPL ablaze this season."
train['text'].iloc[31]
'@bbcmtd Wholesale Markets ablaze http://t.co/lHYXEOHY6C'
train['text'].iloc[83]
"#TruckCrash Overturns On #FortWorth Interstate http://t.co/Rs22LJ4qFp Click here if
   you've been in a crash&gt;http://t.co/LdounIYw4k"
```

[♮] cedric.yu@nyu.edu. Last updated: December 9, 2021.

In particular, the last example contains `&gt` which stands for $>$ in HTML. Thus we will need to process the HTML syntax. This is performed in `dataset_study.py` and `preprocessing.py`, by applying the following function on the datasets:

```
from bs4 import BeautifulSoup
def no_mojibake(row):
    row['text_no_mojibake'] = BeautifulSoup(row['text']).get_text().strip()
    return row
```

From now on, we will use the HTML-processed tweets of the datasets.

1.2 Evaluation Metric

The results will be evaluated by the F1 score, defined as

$$\mathrm{F}_1 \equiv 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \times \text{True Positive}}{2 \times \text{True Positive} + \text{False Positive} + \text{False Negative}}.$$

This is the F_β score with $\beta = 1$. When one prefers high precision (less false positives), one chooses a small $\beta < 1$, and when one prefers high recall (less false negative), one chooses a large $\beta > 1$; using the F_1 score means we wish to strike a balance between precision and recall. Our goal is to maximise the F1 score.

1.3 Overview of Approaches

We will make use of a few different approaches: `CountVectorizer` and `TfidfVectorizer` from `nltk`, LSTM with GloVe embedding vectors, and a transformer network from *Hugging Face*. Starting from Section 4, we will discuss them one by one in detail.

2 Exploratory Data Analysis

We perform an exploratory data analysis in `dataset_study.py`.

2.1 Character Count, Punctuation Marks and Capital Letters

We first look at the character counts[1] of the (HTML-processed) tweets in both datasets. This is plotted in Figure 1, from which we find that the distributions in the training and test sets are very similar, with a maximum at around 140 characters as expected.

[1] There was a 140 character limit, which was doubled in 2017.

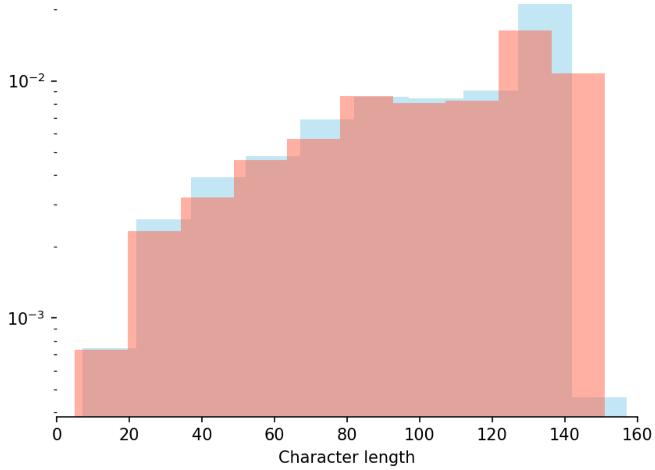
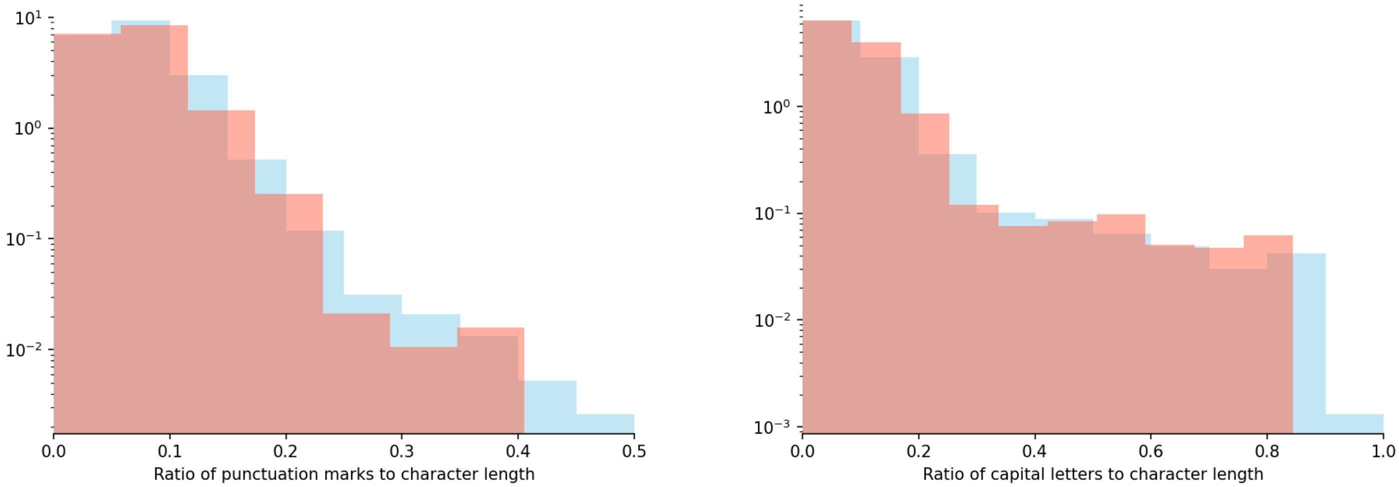


Figure 1: Histogram of character count (in log scale) in the training (blue) and test (red) sets.

Relative to the character count, the portions of punctuation marks and capital letters in both datasets are again very similar, as shown in Figure 2.



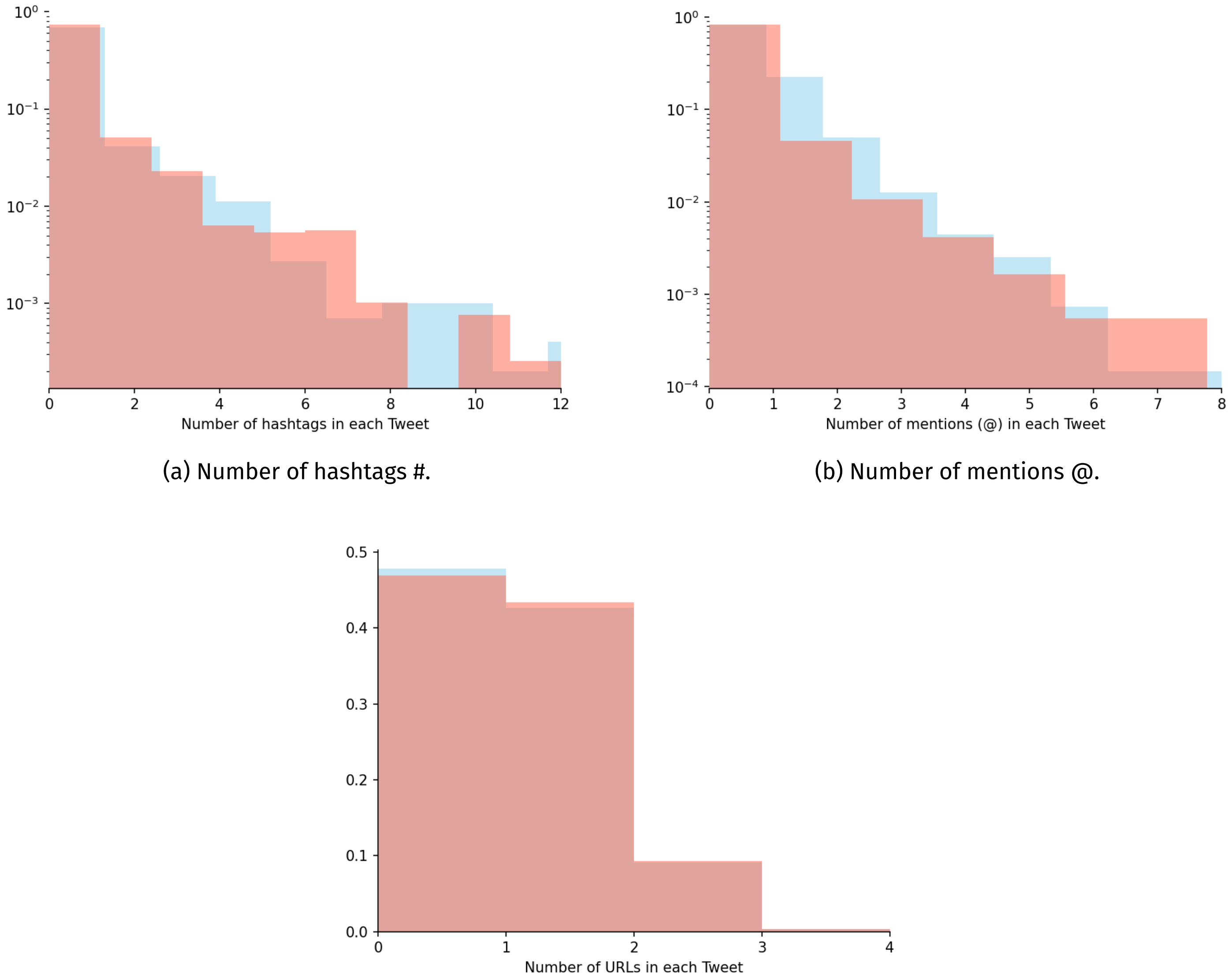
(a) Punctuation mark-to character length ratio.

(b) Capital letter-to character length ratio.

Figure 2: Punctuation mark- and capital letter-to character count ratios in the training (blue) and test (red) sets, in log scale.

2.2 Hashtags, Mentions and URLs

Using regular expressions, we extract the number of hashtags #, mentions @ and/or URLs (http:// or https://) from each tweet, which are plotted in Figure 3. Once again, the distributions for the training and test sets are very similar.



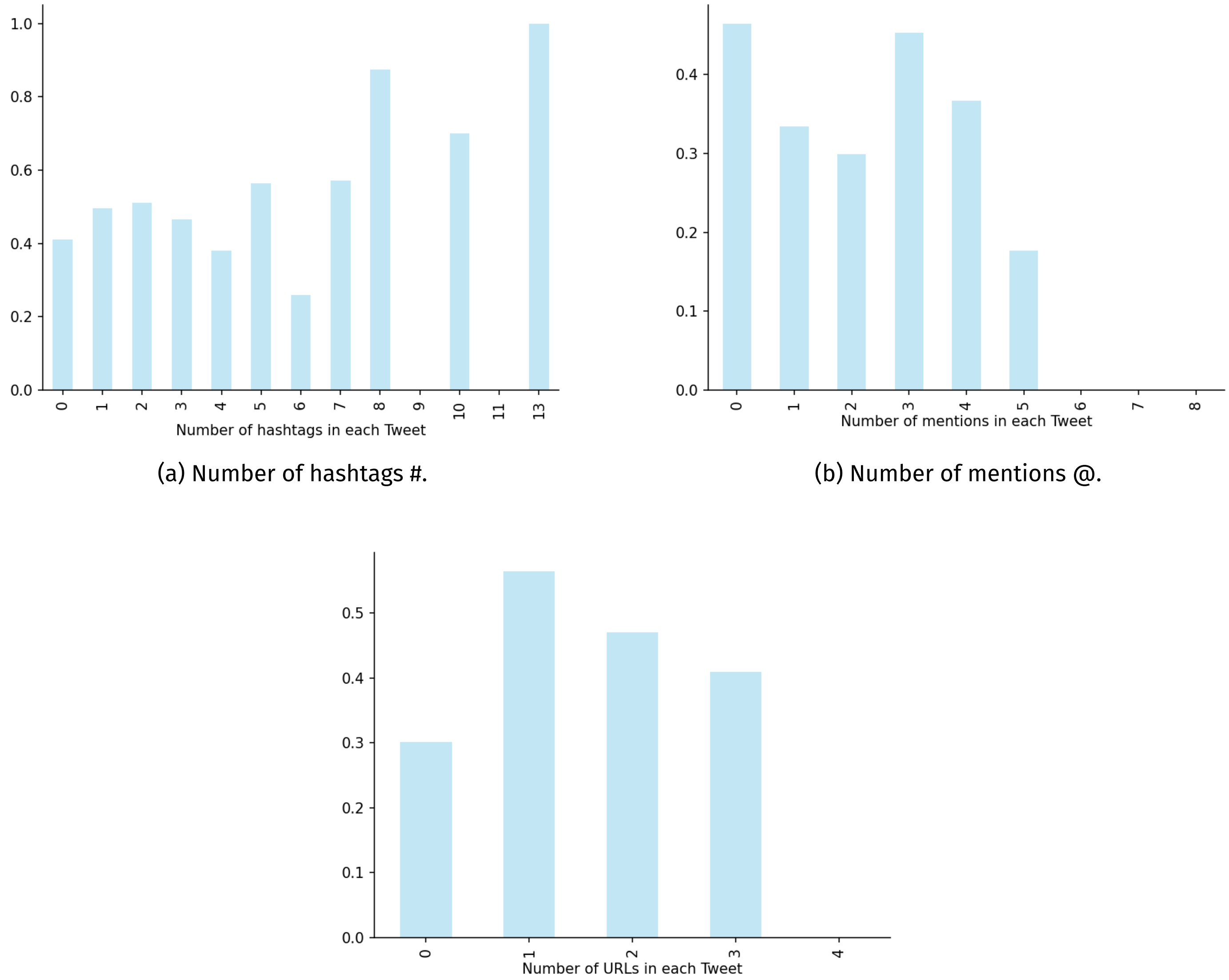
(a) Number of hashtags #.

(b) Number of mentions @.

(c) Number of URLs.

Figure 3: Number of hashtags #, mentions @ and URLs in the training (blue) and test (red) sets, in log/linear scale.

Moreover, we also calculate the target mean in the training set, grouped by the number of hashtags #, mentions @ and/or URLs respectively, as shown in Figure 4. The plots suggest that these numbers do not have apparent strong correlations with the target, and that we should consider encoding methods other than target encoding.



(a) Number of hashtags #.

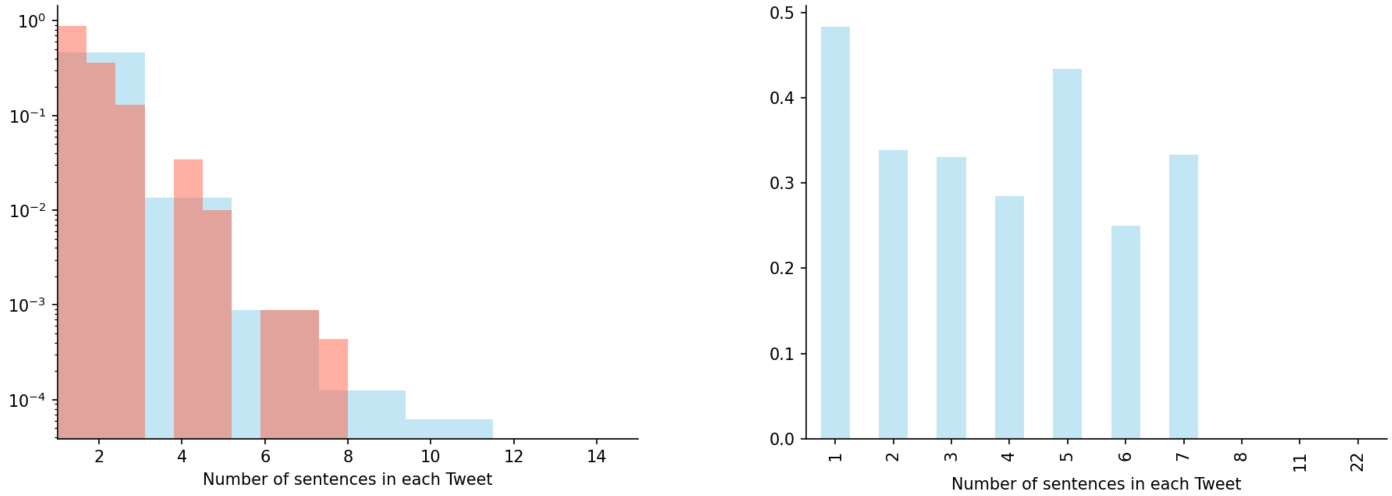
(b) Number of mentions @.

(c) Number of URLs.

Figure 4: Target mean grouped by the number of hashtags #, mentions @ and URLs in the training set, in linear scale.

2.3 Sentence Count and Stop Word Count

Using `nltk.sent_tokenize`, we find the number of sentences in each tweet. The distributions as well as the target means are shown in Figure 5. Once again, the lack of an apparent correlation with the target suggests that we should consider encoding methods other than target encoding.

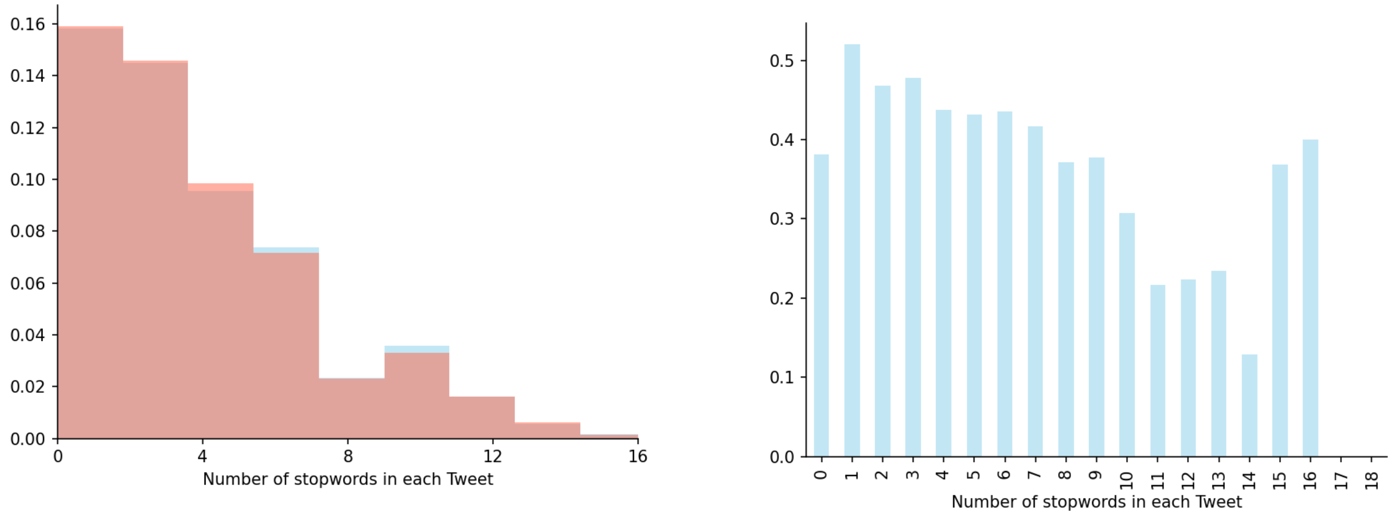


(a) Number of sentences in the training (blue) and test (red) sets.

(b) Target means in the training set.

Figure 5: Number of sentences in each tweet and target means.

The same can be said for the number of (English) stop words (as provided by NLTK) in each tweet; see Figure 6.



(a) Number of stop words in the training (blue) and test (red) sets.

(b) Target means in the training set.

Figure 6: Number of stop words in each tweet and target means.

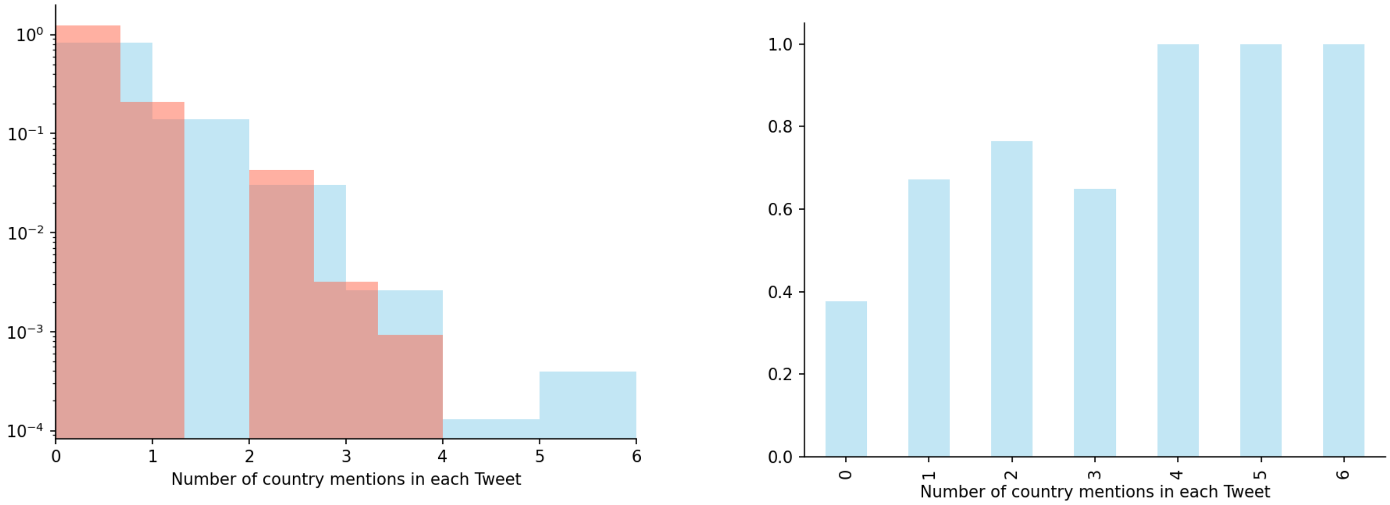
2.4 Country Mentions

Next, we extract from each tweet the number of times a geographical location is mentioned (not necessarily with an @), with the use of

```
from geotext import GeoText
GeoText(<text >).country_mentions
```

We remind the reader that we dropped the original given `location` column due to the large number of missing values and how messy the values are.

We find the following distributions:



(a) Number of country mentions in each tweet in the training (blue) and test (red) sets.

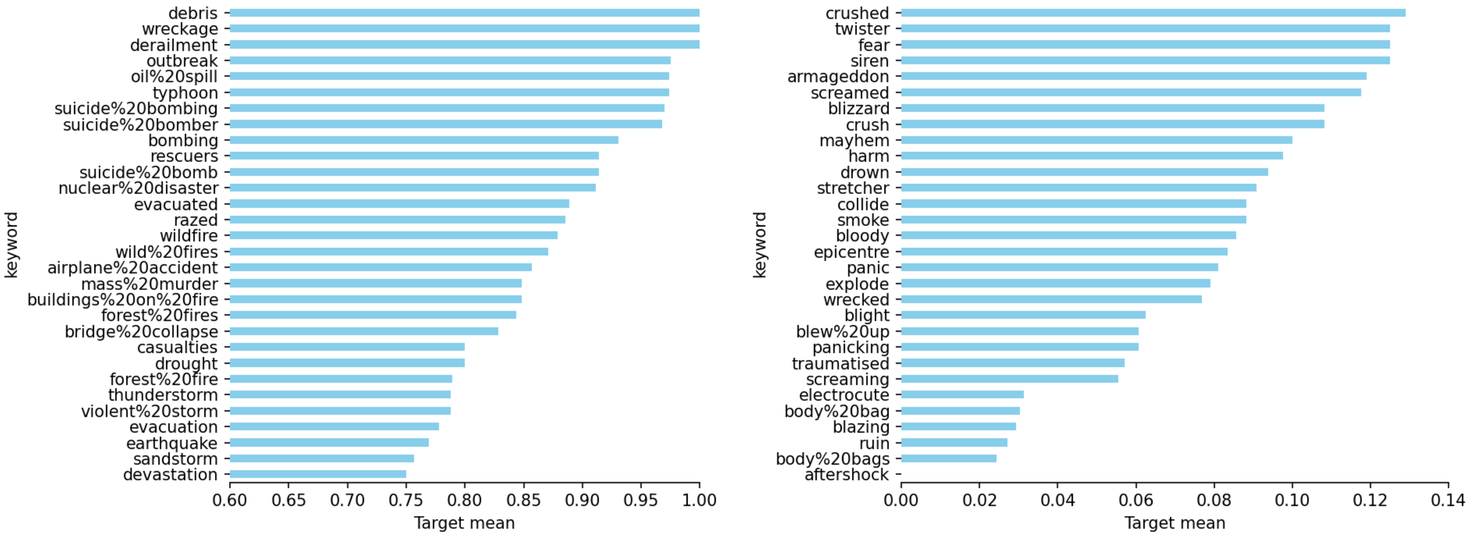
(b) Target means in the training set.

Figure 7: Number of country mentions in each tweet and target means.

Note that the large target mean $(= 1)$ for the number of country mentions $= 4, 5, 6$ in Figure 7b should not be taken seriously; the number of samples for them are respectively $1, 2, 2$ so the numbers are not statistically significant.

2.5 Keyword

`keyword` is the single keyword assigned to each tweet, if any. We plot some of the training set target means in Figure 8. The plots suggests for using target mean encoding the this feature.



(a) Top 30 keywords with highest target means.

(b) Top 30 keywords with lowest target means.

Figure 8: Target means of keywords in the training set.

2.6 Summary

To summarise the findings in our exploratory data analysis, we find that it appears very suitable to target-encode the `keyword` column. Moreover, from `text`, we extract various features: character count, punctuation mark and capital letter ratios, number of hashtags, mentions and URLs, sentence count, stop word count, and country mentions. Their distributions in the training and test sets are very similar and we should not use target encoding for any of them.

3 Pre-Processing: Text and Non-Text Features

We perform pre-processing in `preprocessing.py`. First, we perform a **train (0.8)-validation (0.2) split** on the training dataset.

3.1 Text Features

We first discuss the processing of `text` (tweets). Anticipating the use of GloVe Twitter embedding vectors [1], we adopt the following three text pre-processing steps for all our machine learning approaches, in which some steps are adopted from the Ruby script in [1]:

1. remove HTML syntax, discussed in Section 1.1
2. strip front and end whitespaces
3. expand abbreviation `"n’t"` to `:not"` and `"’s"` to `"  s"`
4. record the ratios of capital letters and punctuation marks to total characters

5. replace each URL by `’<url>’` token
6. replace @mention by `’<user>’` token
7. replace `#<hashtag_body>` by `<hashtag>` token, and split the `<hashtag_body>` by capital letters unless it is all cap
8. represent numbers by `’<number>’` token
9. replace repeated punctuation marks by punctuation mark + `<repeat>` token
10. replace punctuation marks (except `?!.`) by `<punc>` token
11. replace all whitespaces by single space `’ ’`
12. lower case
13. (we do not handle emojis in this exercise)
14. for LSTM: pad sequences to length 128 (maximum token length in the datasets is 107)

These are done in the function `text_preprocess` for each sample, which are applied to each dataset using the function `pd_text_preprocess`. The pre-processed texts without other features are then saved as numpy files.

Note that we do not remove stop words from the texts. This is informed by our previous exercise on IMDB sentiment classification: removing stop words reduced the auc scores across the board.

GloVe Embedding Vectors, Tokenising and Padding

Further processing is needed before we use GloVe Embedding Vectors for LSTM neural networks.

We first load the GloVe embedding vectors from `glove.twitter.27B.100d.txt` using the function `read_glove_vecs`. This function is taken from the assignment of Module 2 of the “Sequence Models" Course on Coursera offered by DeepLearning.AI. One important modification is required: the vocab on line 38522 in `glove.twitter.27B.100d.txt` is `\x85`, which in both Python and the re (regex) package is interpreted as whitespace. To take care of this, in the code, we replace the line `line = line.strip().split()` by `line = re.split(r’ ’, line.rstrip())`. From this function, we obtain the dictionary `word_to_vec_map` mapping each word to the embedding vector, as well as the dictionary `word_to_index` mapping each word to an integer index. The indices are used to construct the embedding matrix in the neural network embedding layer; see Section 5.

Each processed tweet is converted into a list to tokens by splitting under `’ ’`. The lists of tokens are padded by the token `’-1 empty’` to a maximum length of `max_len=128`. The maximum length is so chosen because the maximum token length in the datasets is 107). They are carried out in the function `tokenize_pad_text`.

Next, in order to map each token to the corresponding embedding vector, we need to map it to the corresponding unique index. This is done using the function `sentences_to_indices(X, word_to_index)`, which takes a padded array of tokens of shape (batch size, max_len), X, and the dictionary word_to_index that maps each word to its index. The unknown tokens which do not show up in word_to_index, as well as the padding token '-1 empty', are mapped to index 0, which will be mapped to zero embedding vectors. The resulting (batch size, max_len) array of indices corresponding to the tokenised sentences from X is what will be fed into an LSTM network. The arrays of word indices are saved to numpy files.

3.2 Non-Text Features

From the HTML-processed tweets, using the function get_features we extract the following features: character count, punctuation mark and capital letter ratios, number of hashtags, mentions and URLs, sentence count, stop word count, and country mentions. Note that from get_features we also extract the lists of hashtags and mentions from each tweet. We have not found a good way to use them, but save them in numpy files anyway.

The feature keyword is target mean encoded, while all other categorical features are frequency encoded, both performed using the category_encoders package. After that, the features are scaled with MinMaxScaler.

4 CountVectorizer and TfidfVectorizer from nltk

In essence, CountVectorizer and TfidfVectorizer both perform token counts in slightly different ways: the former does a simple count of the tokens, while the latter computes the “term frequency–inverse document frequency" (TFIDF) of each token. Neither makes use of embedding vectors. We will include the use of ngrams to capture local word orders.

4.1 Workflow

1. load the processed train, validation and test datasets
2. fit and transform processed texts using CountVectorizer or TfidfVectorizer from nltk
3. stack the transformed vectors with the non-text features
4. fit the models: LogisticRegression, MultinomialNB and XGBClassifier
5. predict

They are implemented in master_no_embedding.py. In particular, we use for the vectorisers the parameters min_df=3, max_df=0.3, stop_words='english', ngram_range = (1,3).

4.2 Model Validation and Test Performance

We use the following models: LogisticRegression and MultinomialNB from sklearn, and XGBClassifier.

We first report the results for CountVectorizer.

For LogisticRegression, MultinomialNB and XGBClassifier, the F_1 score of the validation set are respectively 0.7564, 0.7116 and 0.7297.

So LogisticRegression gives the best score. We find the 10 smallest and largest coefficients of this model to be respectively

```
['mode' 'best' 'buy' 'special' 'super' 'august number punc' 'cake' 'user punc' '
    entertainment' 'longer']
```

and

```
['hiroshima' 'california' 'storm' 'crash' 'plane' 'war' 'riots' 'train' 'fires' '
    tornado']
```

They kind of make sense intuitively.

Submitting the test predictions of this model on Kaggle, we get a test score of 0.79190.

From XGBClassifier, we find the top 10 feature importances to be

```
keyword_mean_encoded                  0.038112
country_mention_num_freq_encoded      0.016730
hiroshima                             0.012004
url_num_freq_encoded                  0.010691
california                            0.008889
number                                0.008023
killed                                0.007949
hashtag jobs                          0.007540
chile                                 0.007475
near                                  0.006474
```

Some make sense, e.g. keyword_mean_encoded, hiroshima and killed, while some should be intuitively less important, e.g. near and hashtag jobs.

The results for TfidfVectorizer are as follows.

For LogisticRegression, MultinomialNB and XGBClassifier, the F_1 score of the validation set are respectively 0.7496, 0.7119 and 0.7235.

Among them, LogisticRegression gives the best score. The 10 smallest and largest coefficients for this model are respectively

```
['best' 'special' 'mode' 'good' 'super' 'august number punc' 'new' 'better' 'user punc
    ' 'united']
```

and

```
['hiroshima' 'california' 'crash' 'report' 'storm' 'train' 'war' 'riots' 'cos' 'plane'
    ]
```

which are similar to those from CountVectorizer.

Submitting the test predictions of this model on Kaggle, we get a test score of 0.78118.

In short, the test score for CountVectorizer is slightly better than that of TfidfVectorizer, as indicated by the validation scores.

5 LSTM with GloVe embedding vectors

We use LSTM neural networks on Tensorflow with GloVe embedding vectors [1]. Specifically, we use the 100-dimensional GloVe Twitter embedding vectors of 1,193,514 words.

5.1 Workflow

1. load GloVe vectors
2. load arrays of indices of padded, tokensed processed texts and non-text features of the datasets
3. find optimal neural network architecture involving bi-directional LSTM layers using Keras Tuner
4. re-train the optimal model
5. predict

They are implemented in `master_embedding_LSTM.py`.

5.2 LSTM with Embedding Layer

To construct the embedding layer from the GloVe vector, in the function `pretrained_embedding_layer` we construct the embedding matrix `embedding_matrix` in which row i is the embedding vector of the word of index i. We then construct a Tensorflow `Embedding` layer which is set non-trainable, and set its weight to be the `embedding_matrix`.

For each dataset, our neural network takes as inputs both the array of indices of padded, tokensed processed texts, as well as the encoded non-text features. The basic layout of the architecture is:

text —> Embedding —> bi-directional LSTM layers, non-text features —> Dense layers, concatenate —> fully connected Dense layers —> output with sigmoid.

Informed by the search using `kt.tuners.BayesianOptimization` in `master_keras_tuner.py`, we use the following architecture:

text —> Embedding —> Bidirectional(LSTM(512)) + Dropout(0.2) + BatchNormalization —> Bidirectional(LSTM(1024)) + Dropout(0.2) + BatchNormalization —> Bidirectional(LSTM(256)) + Dropout(0.2) + BatchNormalization —> Bidirectional(LSTM(256)) + Dropout(0.2) + BatchNormalization,
non-text features —> Dense(512) + Dropout(0.2) + BatchNormalization —> Dense(16) + Dropout(0.2) + BatchNormalization
concatenate + Dropout(0.2) + BatchNormalization —> Dense(512) + Dropout(0.2) + BatchNormalization —> Dense(32) + Dropout(0.2) + BatchNormalization —> Dense(32) + Dropout(0.2) + BatchNormalization —> output with sigmoid. For all hidden dense layers, we use `activation = 'tanh'`, `kernel_regularizer = tf.keras.regularizers.l2(l2 = 0.01)`.

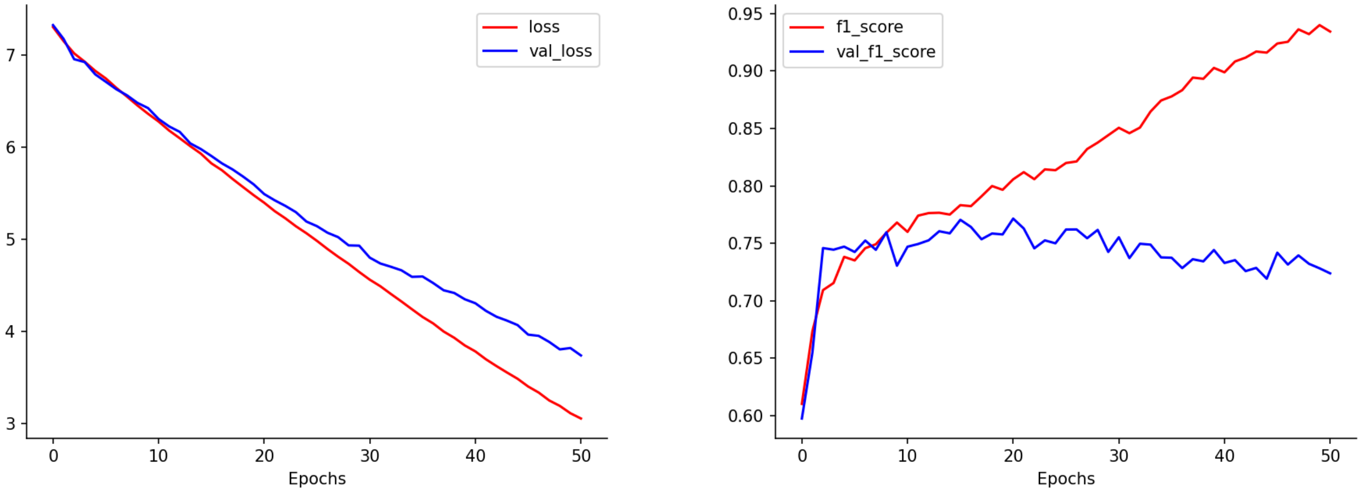
In the function `sentiment_classification_model` in `master_embedding_LSTM.py`, we define this LSTM-based model. The use of kernel regularisers and dropout layers is intended to reduce overfitting.

5.3 Model Validation and Test Performance

Our training hyperparameters are as follows:

```
lr_decay = PolynomialDecay(initial_learning_rate = 0.0001, decay_steps = 20,
    end_learning_rate=0.00001, power = 2)
optimizer = tf.keras.optimizers.Adam(learning_rate = lr_decay)
model.compile(
    optimizer=optimizer,
    loss="binary_crossentropy",
    metrics=['accuracy', tfa.metrics.F1Score(num_classes=1, threshold = 0.5)]
)
callback = tf.keras.callbacks.EarlyStopping(
    monitor='val_f1_score', mode = 'max', patience=30, min_delta = 0.00001,
        restore_best_weights=True)
```

The training curves are as follows:



(a) Training (red) and validation (blue) loss.

(b) Training (red) and validation (blue) F_1 score.

Figure 9: Training curves for the optimal LSTM-based model.

As far as the F_1 score is concerned (the training curves for accuracy are qualitatively the same), the validation score achieves maximum after the first few epochs, after which there is overfitting. This is the case despite the regularisation techniques we employed.

The validation score is 0.7715. The test set score is 0.80784, which is only slightly higher than the previous approach with NLTK vectorisers.

6 Hugging Face Transformer

We use `BertTokenizerFast` and `TFBertForSequenceClassification`, and the pre-trained parameters from `bert-base-uncased`, all from Hugging Face Transformers [2]. It is important that we use the tokeniser associated to the transformer model; here we use `BertTokenizerFast` instead of the `nltk` tokenizer.

6.1 Workflow

1. load list of processed texts of the datasets
2. tokenise with BertTokenizerFast
3. load pre-trained TFBertForSequenceClassification
4. re-train the model with Adam for 4 epochs
5. predict

They are implemented in master_transformer.py.
Note that for convenience, we do not attempt to include the non-text features.

6.2 Tokenisation

We use the BertTokenizerFast tokeniser with config file from bert-base-uncased:

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased/')
```

For each list X of pre-processed non-tokenised text, we obtain the 'input_ids' by

```
X_ids = tokenizer(X, truncation=True, is_split_into_words=False, padding='max_length',
     max_length=max_len)['input_ids']
```

Here we set max_length=max_len=512, which is the maximum length allowed by the tokeniser. We truncate sentences that are longer than this, and pad shorter ones to this length.

In other natural language processing problems such as named-entity recognition, one also needs to align the label of each word from the labeled training set with the label of each subword tokenised by BertTokenizerFast. Here, we do not have this trouble because the target of each entire sentence is one single label.

6.3 TFBertForSequenceClassification Model

We load the transformer model TFBertForSequenceClassification with the pre-trained weights by

```
Bert_trans_model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased/
    ', num_labels=1)
```

Next, we need to re-train the model with our training set. Note that this model predicts logits, so we need to add to the loss function and metric the parameter from_logits=True. With Adam at learning_rate=1e-5, we re-train the model for two epochs with batch size of 4 (with a larger batch size the model cannot fit into the VRAM of the local machine). Each epoch took about 30 minutes. We obtained the following training statistics:

```
Epoch 1/4
1523/1523 [==============================] - 535s 352ms/step - loss: 0.4728 - accuracy
    : 0.7714 - auc: 0.8444 - val_loss: 0.4060 - val_accuracy: 0.8313 - val_auc: 0.8917
Epoch 2/4
```

```
1523/1523 [==============================] - 531s 349ms/step - loss: 0.3464 - accuracy
    : 0.8604 - auc: 0.9145 - val_loss: 0.3924 - val_accuracy: 0.8345 - val_auc: 0.8919
Epoch 3/4
1523/1523 [==============================] - 524s 344ms/step - loss: 0.2629 - accuracy
    : 0.8980 - auc: 0.9482 - val_loss: 0.4083 - val_accuracy: 0.8286 - val_auc: 0.8879
Epoch 4/4
1523/1523 [==============================] - 514s 337ms/step - loss: 0.1863 - accuracy
    : 0.9342 - auc: 0.9717 - val_loss: 0.4586 - val_accuracy: 0.8372 - val_auc: 0.8821
```

Note that we have not figured out how to use the F_1 score as one of the metrics during training. The test F_1 score is 0.83021, which is the highest.

7 Discussion

We have analysed the problem in three different approaches.

For the LSTM-based models, it took one whole night to find an optimal architecture. Training each model is not very time consuming, due to the small dataset size. But the latter is probably the very reason why we suffer from overfitting and cannot achieve a higher validation score, despite the various regularisations techniques used.

The transformer network gives the best scores, as expected. Re-training time does not take too long either, given the small dataset size. More importantly, it takes significantly longer to make predictions. Besides, we should try to also make use of the non-text features, given that the `keyword` feature should be quite important.

All in all, If we disregard the time taken to find the optimal neural network architecture, these methods take similar training time. The transformer network gives the best performance, with a disadvantage that it takes significantly longer compared to the other approaches to make predictions.

In this work, we have not studied how to incorporate the hashtag and mention words extracted from the tweets. This is left for a future investigation.

8 Acknowledgment

Some of the codes are adopted (with modifications and optimisations) from the assignments of the Sequence Model course on Coursera, offered by DeepLearning.AI. We also acknowledge the authors from various sources online, whose tools and techniques were borrowed and implemented in our codes. (I didn't keep track of the references.)

References

[1] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543. 2014. http://www.aclweb.org/anthology/D14-1162.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” *CoRR* **abs/1810.04805** (2018) , arXiv:1810.04805. http://arxiv.org/abs/1810.04805.