Capstone Project

Machine Learning Engineer Nanodegree

Juan Andrés Mora

March 5, 2021

1 Definition

1.1 Project Overview

Machine Learning has been applied to a wide variety of problems. One of the most popular areas of study is Natural Language Processing. Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data. In simpler terms, it allows a computer to understand a person.

On the other hand, one of the most fast and complete sources of information is social media. People everywhere use Twitter and report on real time any emergency they may be observing. But there is no specific signal for warning about a disaster. Therefore, a Natural Language Processing may be used to detect if a user is writing about a disaster or not.

This project originates from an ongoing Kaggle competition "Natural Language Processing with Disaster Tweets"[1] and the dataset was created by the company figure-eight and originally shared in the 'Data For Everyone' site[2].

In this project I implemented a Recursive Neural Network to solve the problem and compared it against a benchmark solution that applies a Linear Ridge Classifier.

1.2 Problem Statement

The objective is to predict which Tweets are about real disasters and which ones are not. The tasks involved are the following:

1. Data exploration: examine the provided dataset and check the data distribution.

[1] https://www.kaggle.com/c/nlp-getting-started/overview
[2] https://appen.com/resources/datasets/

2. Dataset generation: from the labeled file provided generate a training set and a test set.
3. Prepare and process the data: clean the text provided (ex: remove html tags, emojis, stopwords, etc.), stem and split into words.
4. Transform the data: generate a dictionary, convert the data to a numeric notation and pad the text to have homogeneous length.
5. RNN model: build, train and refine our more complex model.
6. Benchmark model: build and train the basic model proposed as benchmark.
7. Evaluation: compare performance of both models on the test set.

1.3 Metrics

As described in the Kaggle evaluation section[3], the competition uses F1 between the predicted and the expected answers. It is calculated as:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

Where:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

TP: True Positive = prediction is 1 and ground truth is 1
FP: False Positive = prediction is 1 and ground truth is 0
FN: False Negative = prediction is 0 and ground truth is 1

We will use this F1 metric to compare between our selected RNN model and the benchmark model.

2 Definition

2.1 Data Exploration

Our dataset, which corresponds to the file as “train.csv”, consists of 7,614 rows (including a header row). We will be using this file as our dataset, as train data in Kaggle is the only labeled set. The test dataset is not labeled because it is used for submission purposes, therefore not being useful to compare our model against a benchmark.

[3] https://www.kaggle.com/c/nlp-getting-started/overview/evaluation

The data provided has several columns:

- id: unique identifier for the tweet.
- text: the text of the tweet.
- location: location the tweet was sent from.
- keyword: particular keyword from the tweet.
- target: only available in the train set, defines a disaster tweet (1) or not (0).

The relevant information for our project comes in the "text" and "target" columns. Here is an example from the text information contained in a row:

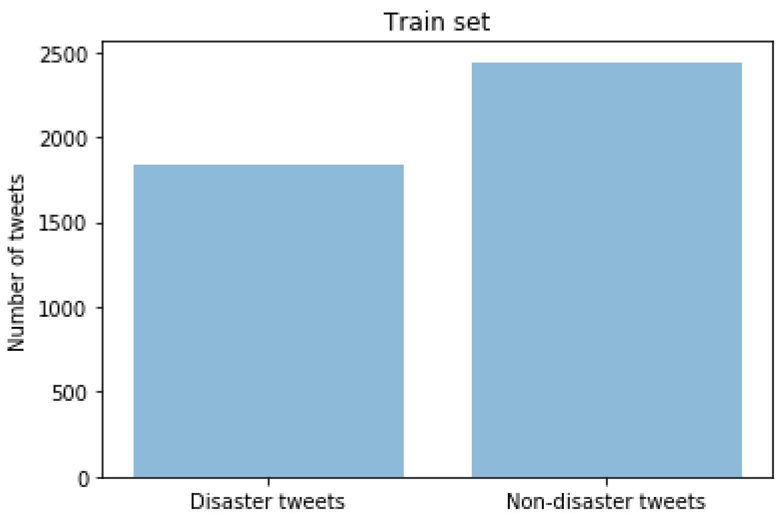
```
'@DougMartin17 Fireman Ed runs into burning buildings while others a
re running out Doug he deserves your respect??????'
```

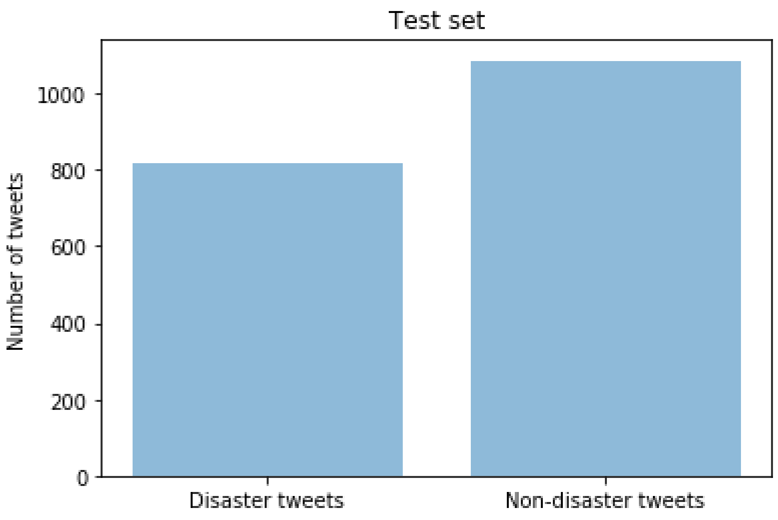
Here we notice the use of special characters such as "@" ("#" is also a common character in twitter) this will have to be cleaned in a data processing stage in order to use the words as inputs for our model.

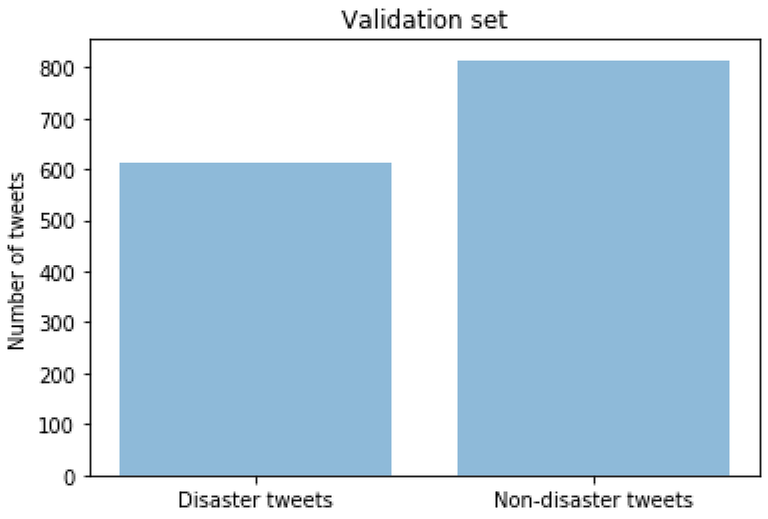
2.2 Exploratory Visualization

In order to split our data we used the `train_test_split` function from the `sklearn.model_selection` package. We used the default option to split the data in 75% and 25% portions, but we used it two times. The first time to split in a test set and a remaining set, then we split the remaining set in a train set and a validation set to be used for the hyperparameter tuning. This process results in 4,281 rows in our train set, 1,904 rows in our test set and 1,428 rows in our validation set. Then size of the train set compared to the train and test set is around 70%.

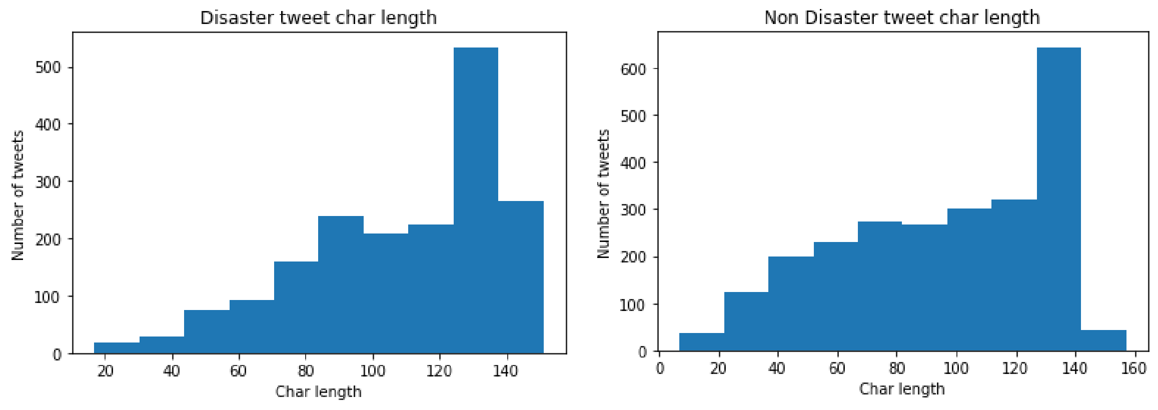
We can see the number of disaster and non-disaster tweets in our sets of data in the following plots. Notice there is data distribution in favor of non-disaster tweets in all datasets (as the `train_test_split` function was configured to maintain the distribution of the original set), but we have enough samples of both labels in all our sets for data imbalance to not be an issue.


Train set
Number of tweets
0
500
1000
1500
2000
2500
Disaster tweets
Non-disaster tweets


Test set
Number of tweets
0
200
400
600
800
1000
Disaster tweets
Non-disaster tweets

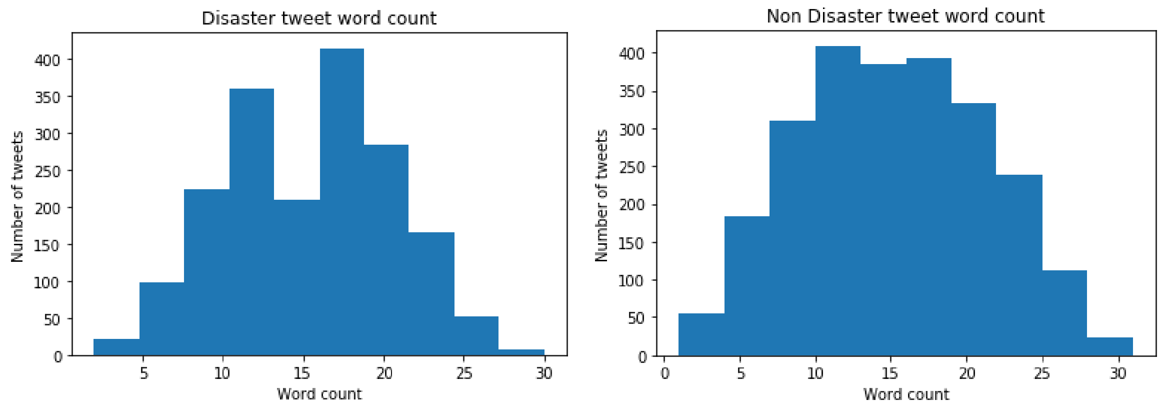

Validation set
Number of tweets
0
100
200
300
400
500
600
700
800
Disaster tweets
Non-disaster tweets

Next, we explore our training set in detail. First we look at the char length of the tweets distribution for both labels ("disaster" and "non-disaster").



We can see that both cases follow a similar distribution, with most of the tweets between 50 and 150 characters.

In the following histograms we look at the word count of each tweet.



3 Methodology

3.1 Data Preprocessing

For our project, we will rely on the "Deploying a Sentiment Analysis Model" project[4] studied in this degree. Following a similar process, we first implement a `tweet_to_words` function which processes a given tweet applying the steps described:

a) Remove html tags
b) Convert text to lowercase

[4] My original submission can be found in: https://review.udacity.com/#!/reviews/2784280

c) Split the text into words
d) Remove stopwords
e) Stem the words to tokenize tweet

We can look at the same example mentioned before after being applying this function:

```
['dougmartin17', 'fireman', 'ed', 'run', 'burn', 'build', 'other', '
run', 'doug', 'deserv', 'respect']
```

After verifying the function works as expected, all the data (training, validation, and test sets) is preprocessed and stored in a file in the cache directory. This is to avoid having to preprocess the data each time we need to run our code.

3.2 Implementation

After pre-processing our data, our implementation begins by creating a vocabulary. This is done with the `build_dict` function, which uses the `Counter` functionality from the `Collections` library. We also save two classifications: one which represents "no-word" in and one for "infrequent" words, which will not be included specifically in the vocabulary. We also choose a parameter now, the size for our vocabulary, in this case 5,000 words.

After creating it, we can look the most frequent 5 words:

```
['co', 'http', 'like', 'fire', 'get']
```

Notice that there are some strange words present:

- `'http'` is expected to be present when sharing links in tweets. We will leave it in the vocabulary as sharing links may or may not be a signal that there is an ongoing disaster.
- `'fire'` is a common word in our vocabulary, but it may or may not refer to a literal fire.

After creating our vocabulary we save it into a file for it to be available without processing all the data again.

Now we proceed to converting our tweets to numerical labels using our dictionary and padding our data. In this case we set the padding to 30 words as we saw it covered most cases of our training data. We apply this process for all our data sets.

We can see how our example from before looks after the `convert_and_pad` function is applied:

```
train_X[100]:
 [4174 2752 2753   90   13   26  856   90 4175 2111 1450    0    0
0
    0    0    0    0    0    0    0    0    0    0    0    0    0
0
```

```
    0    0]

length of train_X[100]:
 30
Review length of train_X[100]:
 11
```

It is now an array of length 30, containing 11 coded words.

Now our data is prepared to create our first model. We will begin implementing the already discussed Recursive Neural Network using Sagemaker and Python. We create an estimator and set “train.py” as the entry point. In this file the default parameters can be observed:

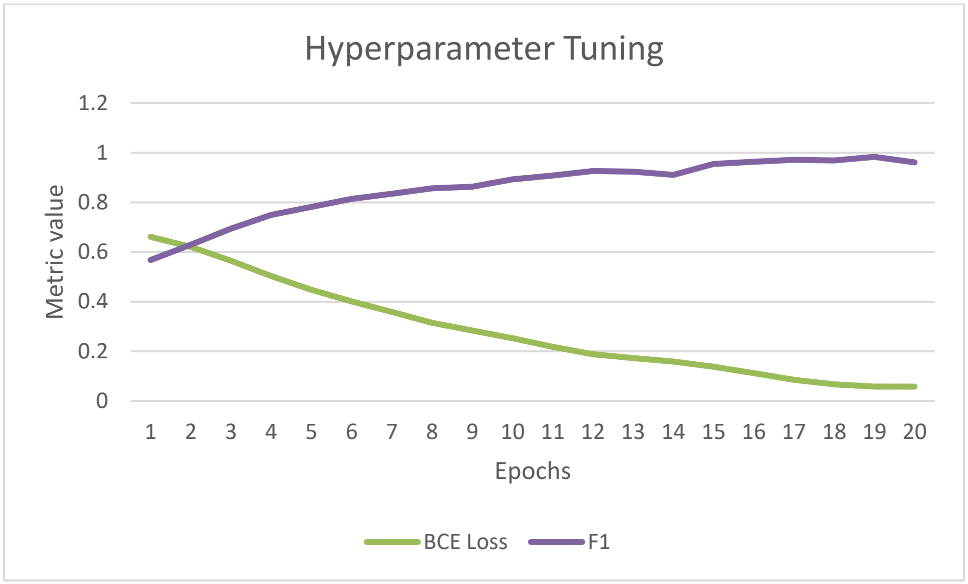
```
# Training Parameters
parser.add_argument('--batch-size', type=int, default=512, metavar='N',
                    help='input batch size for training (default: 512)')
parser.add_argument('--epochs', type=int, default=10, metavar='N',
                    help='number of epochs to train (default: 10)')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')

# Model Parameters
parser.add_argument('--embedding_dim', type=int, default=32, metavar='N',
                    help='size of the word embeddings (default: 32)')
parser.add_argument('--hidden_dim', type=int, default=100, metavar='N',
                    help='size of the hidden dimension (default: 100)')
parser.add_argument('--vocab_size', type=int, default=5000, metavar='N',
                    help='size of the vocabulary (default: 5000)')
```

In this initial model we will be using different hyperparameters: we will begin with 5 epochs and 200 hidden dimesion. We then fit the model with the training data and deploy the predictor. Then we use the predictor with our test dataset in order to obtain the predictions. Using the `sklearn.metrics` package, we apply functions to measure the accuracy and the f1_score of this initial model. The relevant metrics will be discussed in the “Model Evaluation and Validation” section.

3.3 Refinement

To improve our model, we will tune the epochs hyperparameter of our model. To do this, we build a local PyTorch RNN model and train it for different number of epochs. Notice that we must use the training set to train the model, and separately, the validation set must be used to evaluate the performance with a given epoch number. For this, we transform both this datasets into tensors for them to be loaded during the training and validation process.

In each iteration we train our model in batch of 50 rows and then we evaluate its performance in a single batch, printing the number of Epochs used, the BCE Loss (which is used as the loss function in the training process) and the F1 score. This last metric is calculated using the validation set. In the following graph we can see the results obtained.



The highest F1 score was obtained using 19 epochs, so we will modify our initial model and train it using this hyperparameter value. We will call this the "Refined Model". The Refined Model is implemented in the same way as the Initial Model, using SageMaker and PyTorch. After training, the F1 score is calculated using the test set for comparing purposes.

The last step of our implementation refers to what we introduced as "Benchmark Model". This is a Linear Ridge Classifier, a basic model but easy and fast to implement. For our implementation, we use the `linear_model` library included in the `sklearn` package. The model is the trained using the training set and its F1 score calculated.

4 Results

4.1 Model Evaluation and Validation

Through our implementation process we implemented and measured F1 scores for three different models: our "Initial Model", the "Refined Model" and the "Benchmark Model". As stated initially, our chosen metric to compare the models is the F1 score. The following table shows the scores obtained by each model.

Model	F1 Score
Benchmark	0.0912052
Initial Model	0.5697131
Refined Model	0.6644866

We first notice that our Benchmark Model did poorly, reaching around 9% in the F1 Score. Our initial, more complex, RNN did substantially better, achieving an F1 score of around 57%. After improving from 5 epochs to 19 epochs with the hyperparameter tuning process, our Refined Model improved the Initial Model's performance, achieving 66%.

4.2 Justification

Incrementing the number of epochs to improve the model's performance comes with a cost. The fact that our Refined Model has higher epoch directly translates into a higher training time. Also, due to the fact that the training set passes several times through the model, it will make it more prone to overfitting, compared to alternatives with lower epochs. This is why it is also relevant to control the BCE Loss of the model as a complementary metric, as in case it is not lowering, we may be overfitting.

Finally, the results obtained by the Refined Model are well above our initial expectations. As we defined a range for 57% to 64%, we have found to be above this with 66.48%. Looking at the official Kaggle leaderboard for this competition[5], this would land our model in the 1234 position of the public leaderboard.

5 Conclusion

5.1 Reflection

Our objective was to classify tweets to check if they warned of a disaster or not. We proposed the most popular and all-around well performing alternative for Natural Language Processing, a Neural Network implementation.

This solution consisted of many steps, but can be summarized in: Preparing the data, implementing our model, improving them and evaluating against a benchmark. During this process I realized the importance of splitting the data correctly from the beginning. In my case, I initially did not plan for a validation set, and after trying to split the data with an already evaluated model, I realized there was too much opportunity for error and decided to redefine the

[5] https://www.kaggle.com/c/nlp-getting-started/leaderboard

sets from the beginning and implement my solutions again. This made the process much easier and now I am sure there is no data overlapping in the sets used.

Another relevant issue is to consider the amount of computing power that is needed in order to tune hyperparameters. My first attempts of hypertuning were done trying to use the Sagemaker's AutoHyperTuning function to evaluate several parameters in different ranges using my custom model. This proved to be too computationally expensive and resulted on my account being blocked several times. It is relevant to note that this function may work better with SageMaker built-in models. After this I opted for a less computationally expensive tuning, focusing on epochs only and using a PyTorch model in my Jupyter Notebook.

5.2 Improvement

There are many ways in which this model can be improved. For example, we can start evaluating different parameters options:

- A smaller vocabulary size to remove potential infrequent words in our models. This would also reduce pre-processing time and allow the use of bigger datasets.
- Varying the padding size of our tweets. We used 30 words in order to cover most of the tweets presented, but this may result in including many irrelevant information. This parameter may be modified to study the results.
- We can also hypertune the hidden dimensions parameter along with the epochs parameter. Choosing an optimal value may improve the overall performance of our model.

These solutions may improve the model performance under the F1 metric, but as mentioned earlier, one should be wary of not overfitting the model. In order to avoid overfiting, the train set may be split in several subsets and use a cross-validation technique in order to determine the best model and parameters. In conclusion, there is room for improvement, which makes the current results much more promising.