My approach to solve the “StumbleUpon Evergreen” Classification Challenge.

- Abhishek Mohanty

The problem statement outlines that our mission is to build a classifier to evaluate a large dataset of URLs and texts and label them as either evergreen or ephemeral. Thus, it is a text categorization problem at it’s core.

As with any task, it is essential to follow a roadmap that defines a general path of the steps to be taken up to solve it. The very first step was to gather the data, and to look and explore for any insights it could provide. The dataset includes two tab-separated value files, the train and test files, containing 7395 and 3171 entries respectively. A whole lot of information is contained, such as alchemy scores and categories, hyperlink word score, ratio of spelling errors in the webpage among several others.

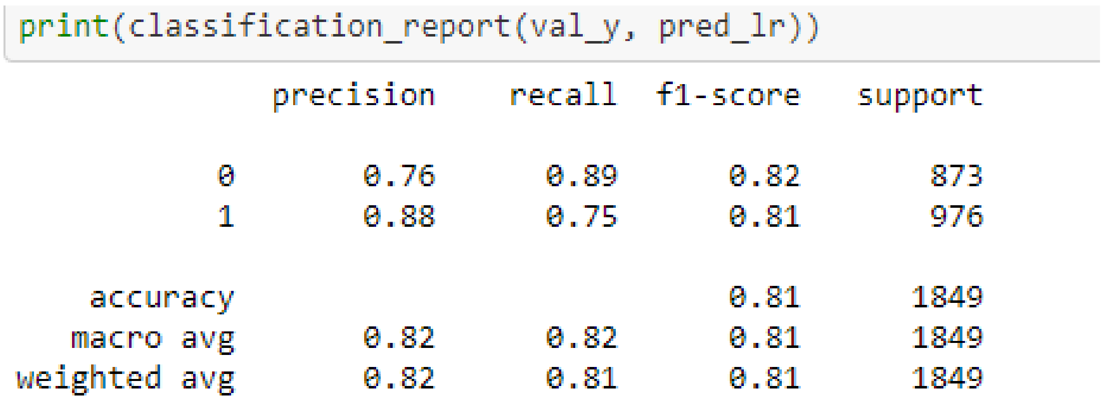
Up next was the step to choose an evaluation metric to assess the performance of the system later on as well as to be able to make comparisons with other models. Being a classification problem, the metrics chosen were: accuracy with precision-recall and F1 score.

The following course of action was to establish a baseline model. It is always better to start off with something simple, and then look to improve upon it with more complex methodologies and techniques. Hence it is important to have a simple and easily reproducible baseline which helps in understanding the task at hand much better. Thus, I decided on a fairly simple Logistic Regression algorithm to go with.

On having established the metric and the initial model, next up was to understand the data, and what better ways to do so than visualizations? A good manner of visualization can help grasp complex relationships in the data quickly and rather easily. Therefore, I loaded in the data in the train tsv file using Pandas. First and foremost, I wanted to check if there were any missing values, since if it were the case then there may have had to be steps to inpute said missing values. However, there were no instances of any missing values. I further used a couple of other plots and charts in attempts to find any meaningful correlations between the feature data points and the labels. Due to the lack thereof, I simply stuck with the ‘boilerplate’ column/section of the data as the key feature to categorize the URLs. Since the distribution of data points for each labelled class were nearly the same, the issue of skewness could be discarded.

With all of the initial steps verified, I looked to then get into the whole process of building the classifier. Dealing with text data, it needs to be transformed it into a format that can be parsed by the system. For the initial model being Logistic Regression, I used a TF-IDF transformation to transform the words in the text. This is an information retrieval technique to quantify a word in documents, generally a weight to each word is computed which signifies the importance of the word in the text corpus. Essentially, it allows for the text data to be vectorized for the system to make sense of.

The next move was to finally define our model and train it upon the data present. I defined the regression model with associated parameters, and then split the overall training data into train and validation data sets in order to measure the performance of the system. After fitting the data to the model and training it, I evaluated the performance by making predictions for the validation data set aside before training. Here’s the result I obtained:



```
print(classification_report(val_y, pred_lr))
```

```
              precision    recall  f1-score   support

           0       0.76      0.89      0.82       873
           1       0.88      0.75      0.81       976

    accuracy                           0.81      1849
   macro avg       0.82      0.82      0.81      1849
weighted avg       0.82      0.81      0.81      1849
```

With the baseline model done, I looked to then see if there was any way I could improve on the scores by maybe using a more complex model. Ultimately, I chose to go with DistilBERT, a lightweight version of the Transformer based ML Technique for Natural Language Processing, BERT, which stands for "(Distilled) Bidirectional Encoder Representations from Transformers." Evidently, this was quite a leap, going from regression analysis to a complicated technique involving intricate concepts and methodologies. Here’s an overview of how and why I chose to go ahead with such a model over anything else:

- I considered RNNs as they are specialized in structure to handle sequential data. However, it’s major drawbacks of exploding and vanishing gradients, as well as their inefficiency to process long sequences when tanh or relu activation functions are utilized, paved the way for Transformers.
- Transformers, since their inception, have revolutionized the domain of Natural Language Processing since they clearly outclassed RNNs, for they did not suffer from long dependency issues. Apart from amazing characteristics such as non sequentaility and positional embeddings, a newly introduced technique of "self-attention" gained traction. A research scientist at Google Brain terms it as "The Most Important formula in Deep Learning after 2018"; source - Twitter.
- BERT was created using Transformer architecture and published by Google in 2018 that achieved state of the art performance on a number of NLP tasks such as text classification, summarization and generation among others.
- As per HuggingFace, "DistilBERT is a small, fast, cheap and light Transformer model trained by distilling BERT base. It has 40% less parameters than bert-base-uncased, runs 60% faster while preserving over 95% of BERT’s performances".

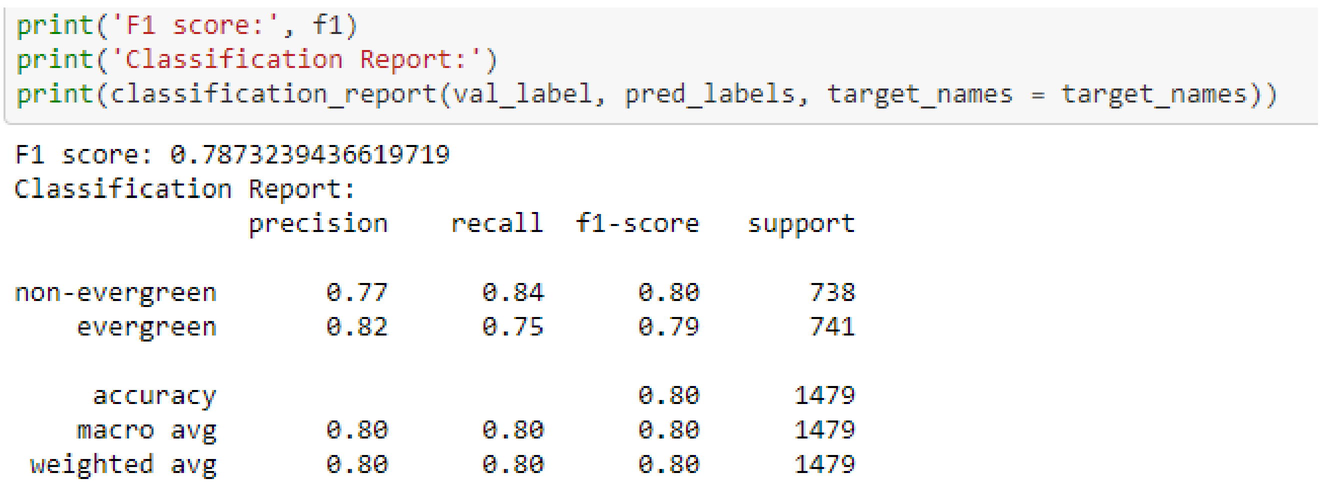
Thus, in brief, I considered LSTMs (variant of RNNs) for this classification problem initially. However, with Transformer architecture performing outright better, and considering how easy the Transformers library makes to import and utilize such complex architectures to provide state of the art Natural Language Processing for everyone, I chose to go with

DistilBERT which gives BERT-like performance faster and with lesser parameters.

Before moving ahead with building the model, an important action was to preprocess and clean up the data as it directly would affect the ability of the model to learn. This whole process included the conversion of the entire text data into lowercase, removing 'stopwords' such as 'a', 'and', 'the' - all in order to provide more focus to the words that hold the "meaning" of the text, getting rid of special symbols, characters and extra spaces in the text, as well as eradicating any words with less than 2 characters in it's composition. I also made sure to shuffle the train dataset in hopes that it would limit bias creeping in and would improve the model quality and predictive performance.

Therefore, the next step was to import and initialize the DistilBERT Tokenizer and Model. The text data must be transformed into tokenized ids and attention masks after padding, which serve as input for the aforementioned model. After having instantiated all the input parameters, I defined a neural network using the DistilBERT Embeddings, which has DistilBERT model as a layer along with the required layers for the input to the model layer, a hidden dense layer as well as another dense layer for the output. I settled on such structure since it yielded good results after hyper tuning the parameters for certain other combinations. Furthermore, all of the text data in the train and test files, and also the labels in the training data was required to be tokenized into the appropriate input ids and it's attention masks too so that it could be in the format required to behave as input for the model. I set aside 20% of the train data as validation data to evaluate the model's performance estimate.

After defining the parameters loss, metric and optimizer, the model was fit using the mentioned specifications and ultimately trained using the training data. On the completion of training, predictions were made upon the validation data to analyze the performance. Generated the F1 score along with the accuracy of the model and the precision and recall for each of the two classes. Here’s what the results were:



```
print('F1 score:', f1)
print('Classification Report:')
print(classification_report(val_label, pred_labels, target_names = target_names))
```

```
F1 score: 0.7873239436619719
Classification Report:
               precision    recall  f1-score   support

non-evergreen       0.77      0.84      0.80       738
    evergreen       0.82      0.75      0.79       741

     accuracy                           0.80      1479
    macro avg       0.80      0.80      0.80      1479
 weighted avg       0.80      0.80      0.80      1479
```

Note: results for the accuracy, precision and recall metrics varied quite a few times, ranging from 0.78 to 0.84. I believe that such occurrence could be the influence of a few factors including, but not limited to, shuffling the dataset causing the model to learn slightly different each time and thus providing a varying measure of performance on evaluation, on account of it’s probable stochastic nature.

Lastly, I made the predictions on the test data, and created a submission csv file containing the URLID and it’s obtained forecasted labels generated.

Inference: both techniques gave nearly similar values of accuracy, precision and recall despite the heavy differences in their methodologies. This emphasized how important it is to not skip the primary steps of considering traditional approaches before making the leap to neural networks. I also believe that DistilBERT could prove to be more uniform and slightly better with the right tweaks and fine tuning.

Thank You.