Project December 17, 2022

Yahoo Troll Question Detection Project Report

IMT2020131 Amar Pratap Singh
IMT2020128 Ujjwal Agarwal

Course: Machine Learning

1 Problem Statement

We were given a hypothetical scenario where Yahoo's CEO Merissa Meyer had tasked us with creating a machine learning model to detect spam and troll questions so that they can be removed. We were given a hand-labeled dataset which consisted of questions with unique IDs and whether they are troll questions or not.
Each question provided in the training dataset is mapped to either 0 or 1.
$0 \longrightarrow$ HAM
$1 \longrightarrow$ SPAM

The full dataset for the given problem description can be found here.

2 Dataset Analysis

We thoroughly analyzed the dataset.

- Each row in the dataset consists of a question ID, a question, and a class label. The class label tells which classification class (0 or 1) the question belongs to.
- There are 938130 questions classified as 0 and 61870 questions classified as 1. Therefore, majority class 0 is 15.162 times larger than minority class 1.
- There are no duplicate rows or rows with null values.
- Testing data consists of 306122 rows/questions.

3 Preprocessing

Lowering the text: Since a sentence in uppercase or lowercase does not change its meaning. Hence, we converted all questions to lowercase to avoid the same sentence in different cases to have different meanings.

Removing punctuation/special characters: Punctuation and numerical data were not important to the models used, and hence, were removed. Basically, we only needed English words to train the models, and hence, anything other than the 26 English characters was removed.

Stop words removal: Words like this, an, is, which are inevitably present in almost every sentence do not make any contribution to the meaning of the sentence. We used the list of stop words from the NLTK module to identify and remove the stop words from the questions.

Tokenization: We "tokenized" the questions, that is, created a list of words that denotes every question.

Similar to what the "split" function does in python. We used the "word_tokenize" module of the NLTK package for tokenization

Stemming: Stemming is the process of converting the words of a sentence into their stem, basically removing the "suffix". This is necessary because a stem may have various versions after adding

different suffixes, which mean the same. We used the PorterStemmer that NLTK provides. PorterStemmer applies a set of five sequential rules (also called phases) to determine common suffixes from sentences.

Example: Stem word for Eating, Eats, Eaten is Eat

Lemmatization: Lemmatization is the process of converting any word to its base form. We used the WordNetLemmatizer that NLTK provides. Lemmatization converts the words based on the context of the word in a sentence(part of speech).

Example: Better, Best to Good

We observed from the dataset that all the pre-processing steps mentioned above is actually reducing the model accuracy. Some of the reasons for this are:-

- We should consider words in block form also. A spam question sent by an angry shouting user is very likely to have words in block letters.
- We should consider punctuations also. A spam question sent by an angry user is very likely to contain a lot of ?, !, etc.

Therefore, we decided to drop all the above pre-processing steps and continued with vectorizing step explained below.

Vectorizers:

We used the vectorizer from Sklearn library.

1. **CountVectorizer:** It uses the bag of words model to convert arbitrary text into fixed-length vectors by counting how many times each word appears.
2. **Tf-Idf Vectorizer:** The Tf-idf score for each word was calculated as given above to generate the feature vectors for each sample question.The Tf-idf technique is good as it gives importance to rarely occurring words.

We tried both of the above vectorizer but for our dataset TF-IDF vectorizer was giving a better result. So we further changed the parameters of TF-IDF vectorizer which are described below:

analyzer: Whether the feature should be made of word or character n-grams.

ngram_range: The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n $\leq$ n $\leq$ max_n will be used. For example an ngram_range of (1, 1) means only unigrams, (1, 2) means unigrams and bigrams.

max_df: When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold.

max_features: If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

We horizontally stacked "char vectorizer" with "word vectorizer" by setting the analyzer first as 'char' and then as 'word' respectively. We set the n_gram range to (1, 3), max_df to 0.5, and max_features to 150000. The feature extracted table looks like:-

	word 1	word 2	..
Doc1	Freq(word1) in Doc1	Freq(word2) in Doc1	..
Doc 2	Freq(word1) in Doc2	Freq(word2) in Doc2	..
.	.	.	..

4 Balancing The Dataset

4.1 OverSampling

The random oversampler provided by imblearn was used to balance the dataset as a majority of the samples were of the non-troll category. The random oversampler added duplicate samples of the minority class to the dataset in order to make the count of troll and non-troll questions equal.

SMOTE was another technique (based on generating synthetic samples of the minority class rather than duplicates) that we tried but was later discarded due to it taking too much time to balance the given data.

4.2 UnderSampling

Similar to OverSampling, the random undersampler balance the dataset by removing samples of the majority class. This can be done if we have an abundant number of training data.

We used both the above mentioned way for balancing the data. But, it was good for validation data alone. It gave a very poor f1-score on testing data. So, we dropped it.

5 Models

5.1 Logistic Regression

```
logistic_regression_model = LogisticRegression(
    dual = False,
    class_weight = {0: 0.9, 1: 2}
)
logistic_regression_model.fit(X_train1, Y_train)
```

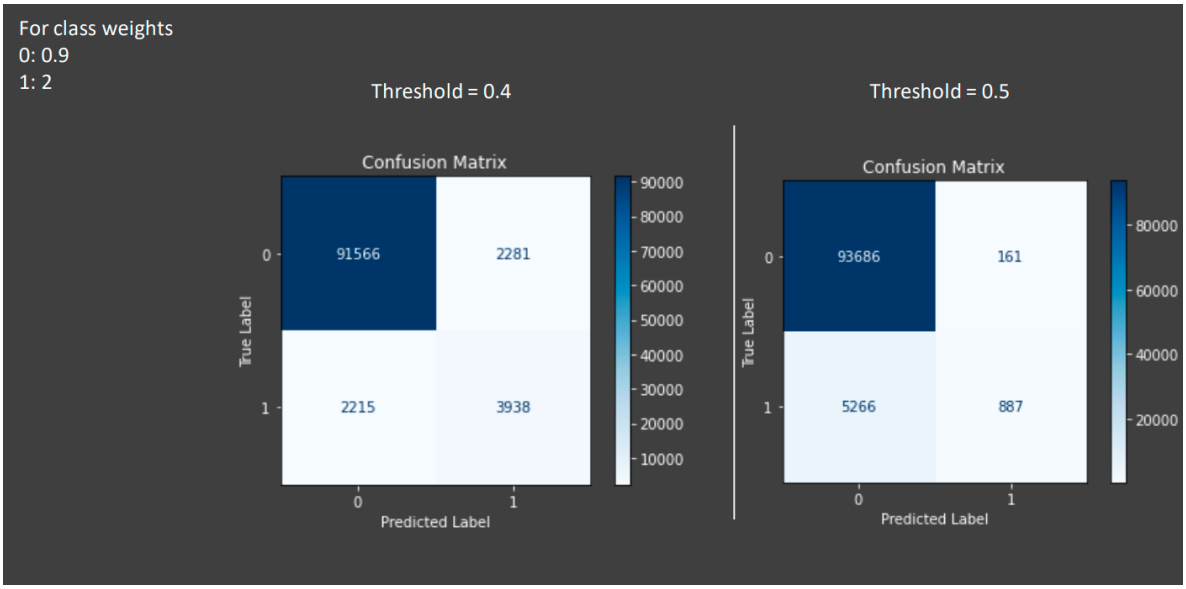
We focused on Logistic Regression as the main classification algorithm to train our model. LR performed poorly on the imbalance dataset. So, we added appropriate class_weight to balance the dataset. With LR, we also needed to tune the threshold to classify test data more accurately.

Following is the code snippet to classify the test data with a different threshold.

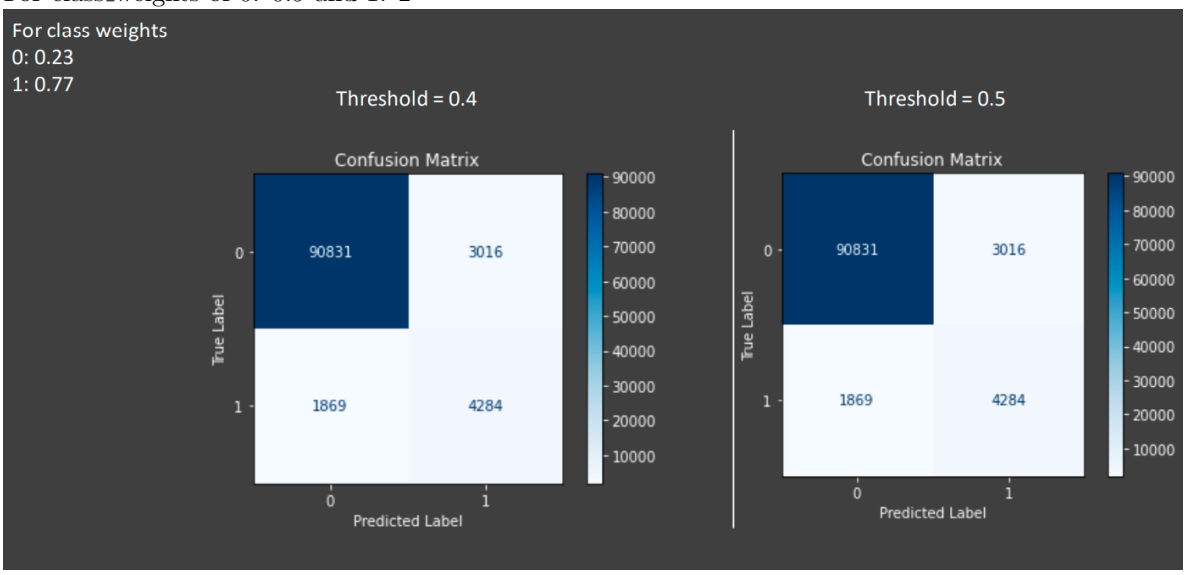
```
def custom_predict(X, threshold):
    probs = logistic_regression_model.predict_proba(X)
    return (probs[:, 1] > threshold).astype(int)
```

A threshold of 0.4 (by default, it's 0.5) was giving a better result on testing data than the default threshold.

Below are two images that show "Confusion matrix" for different class_weights and different threshold.



For class_weights of 0: 0.9 and 1: 2



For class_weights of 0: 0.23 and 1: 0.77

We got the highest f1-score of 0.651 with the above-mentioned parameters in LR.

5.2 Other Models

We have also tried the following well-known models:-

- Multinomial Naive Bayes
- Support Vector Machine
- XG Boosting
- Random Forest Classifier

But, Logistic Regression outperformed all the above models.

We also tried a voting based classifier which take votes of three models and choose the best classification class for a given data. These three models were Multinomial Naive Bayes, Logistic Regression, and SVM.

Still, it couldn't match up to an f1-score given by Logistic Regression.

6 Results

Following are the results that we got from various classification models on the validation dataset.

Models	**f1-score**
Multinomial Naïve Bayes	0.61
Logistic Regression	0.64
Linear SVM	0.59
Vote-based Classifier	0.60
XG-Boost	< 0.40
Random Forest Classifier	< 0.50

Slight variations from the above mentioned score were seen by tweaking/tuning some parameters while training the model.

7 Conclusion

We were successfully able to achieve an f1-score of 0.64 on the complete testing dataset with Logistic Regression.