Project Report

TOXIC COMMENT CLASSIFICATION CHALLENGE TASK

September 7, 2018

Yongbo Wang

Contents

Abstract

In this experiment, we create four recurrent neural network models (LSTM, GRU, Bi-LSTM and Bi-GRU) which predict a probability of each type of toxicity for each comment.

1 Introduction

Discussing things you care about can be difficult. The threat of abuse and harassment online means that many people stop expressing themselves and give up on seeking different opinions. Platforms struggle to effectively facilitate conversations, leading many communities to limit or completely shut down user comments. The Conversation AI team, a research initiative founded by Jigsaw and Google (both a part of Alphabet) are working on tools to help improve online conversation. One area of focus is the study of negative online behaviors, like toxic comments (i.e. comments that are rude, disrespectful or otherwise likely to make someone leave a discussion). So far they've built a range of publicly available models served through the Perspective API, including toxicity.

But the current models still make errors, and they don't allow users to select which types of toxicity they're interested in finding (e.g. some platforms may be fine with profanity, but not with other types of toxic content). In this competition, we're challenged to build a multi-headed model that's capable of detecting different types of of toxicity like threats, obscenity, insults, and identity-based hate better than Perspective's current models. We'll be using a dataset of comments from Wikipedia's talk page edits. Improvements to the current model will hopefully help online discussion become more productive and respectful.

2 Summary of the Data

The contestants are provided with a large number of Wikipedia comments which have been labeled by human raters for toxic behavior. The types of toxicity are toxic, severe_toxic, obscene, threat, insult and identity_hate. There are three files provided, the training set (train.csv) contains comments with their binary labels, the test set (test.csv), you must predict the toxicity probabilities for these comments. To deter hand labeling, the test set contains some comments which are not included in scoring. The file sample_submission.csv is a sample submission file in the correct format.

The maximum and minimum lengths of comments are 5000 and 6, respectively. The majority of length of comments are focus on between 6 and 1000. Besides, μ and σ of the

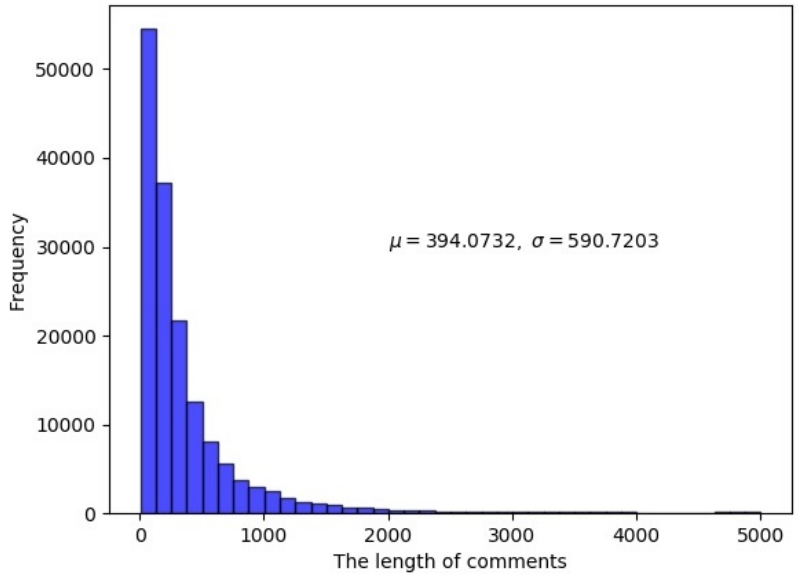

Figure 1: Histogram of word frequency distribution.

dataset are approximately 394 and 590, respectively. With the length increases, the number of comments decreases dramatically as seen in the Figure 1.

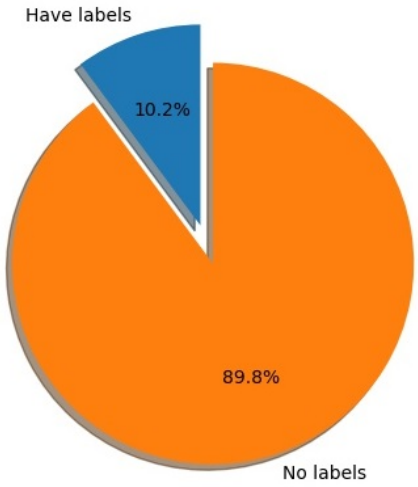

Figure 2: The ratio of whether or not there is a label.

There are 159571 samples in the training set and 153164 samples in the test set. However, more than 89% of training set have no labels (i.e. the value of each type of toxicity is zero) as seen in the Figure 2.

3 Data Preprocessing

Since the raw comment texts contain punctuation and other meaningless symbols, remove the symbols before proceeding with the word embedding.

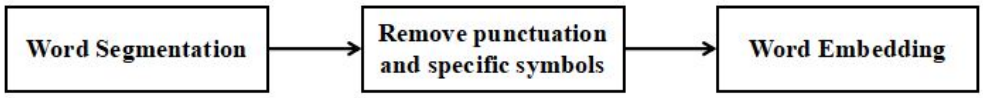

Figure 3: Toxic comment preprocessing.

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. *glove.6B.50d.txt* pre-trained word vectors is used to generate vector representation of words. The entire pre-processing process can be seen in the Figure 3.

4 Bi-LSTM and Bi-GRU Model

In this section, we demonstrate the design of our proposed Model. Figure 4 shows the architecture of our Toxic comment Classification Model which consists of four layers. The first layer is a Bidirectional Long Short Term Memory (Bi-LSTM) layer or a Bidirectional Gated Recurrent Unit (Bi-GRU) layer which generates the high-level abstraction of the input sequence. Both semantic and temporal dependencies of the toxic comment text are considered into Bi-LSTM or Bi-GRU.

The output of Bi-LSTM or Bi-GRU h_n (the last item in the hidden vector) acts as the input for the later fully connected neural network layers. Each output of previous layer will be the input of the later layer. ReLU is the activation function and for the previous fully connected neural networks, and we use a sigmod activation function for the last fully connected neural network.

5 Tune Model Hyperparameters

In this task, we use the grid search capability from the scikit-learn python machine learning library to tune the hyperparameters of Keras deep learning models. Keras models can be used in scikit-learn by wrapping them with the KerasClassifier or KerasRegressor class. The main parameters to be adjusted are *batch_size, epochs, learning rate, dropout rate* and *the number of neurons*.

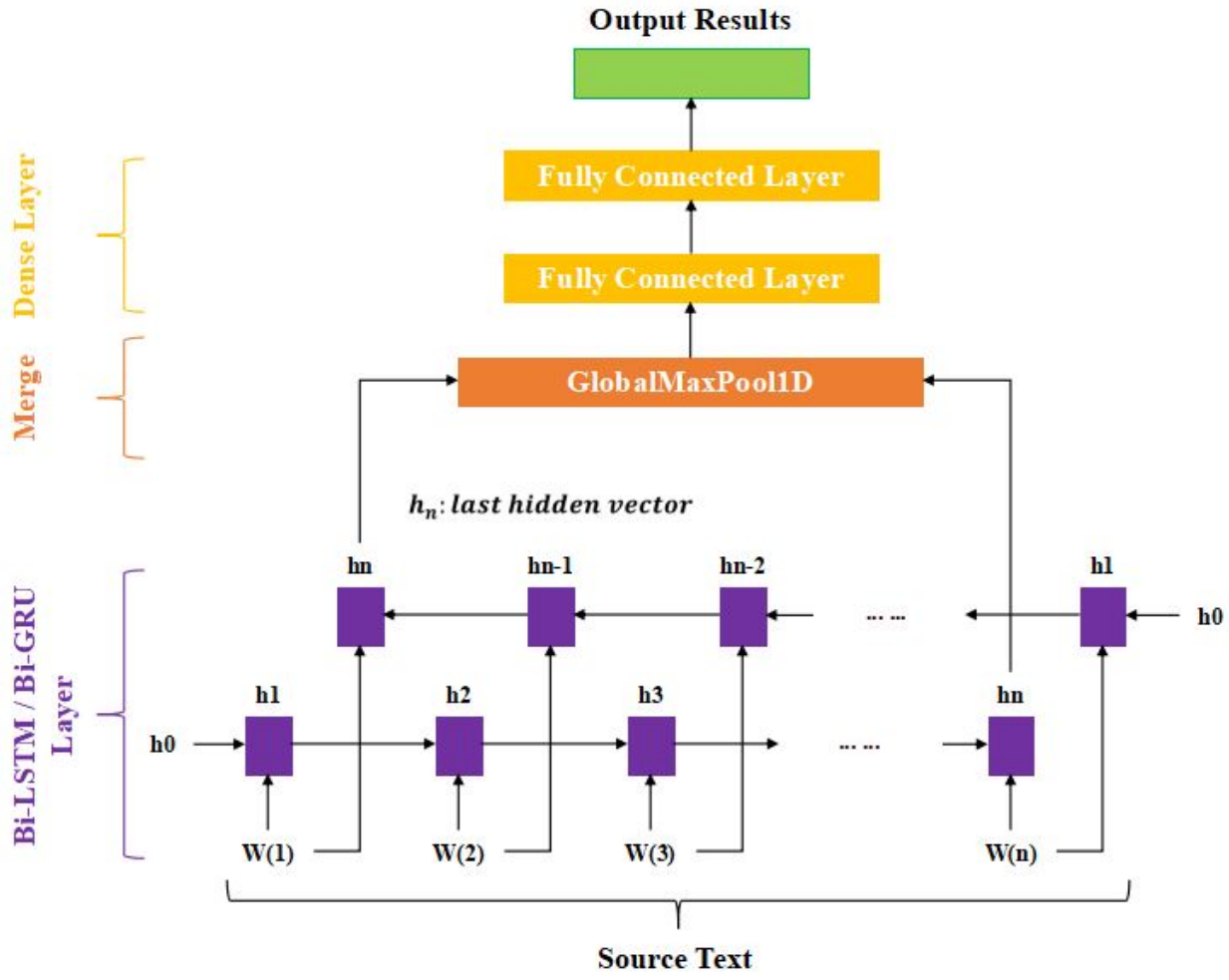

Figure 4: Toxic comment Classification Model.

To use these wrappers you must define a function that creates and returns your Keras sequential model, then pass this function to the build_fn argument when constructing the KerasClassifier class. In scikit-learn this technique is provided in the GridSearchCV class. The GridSearchCV process will construct and evaluate one model for each combination of parameters. Cross validation is used to evaluate each individual model and the default of 3-fold cross validation is used, although this can be overridden by specifying the ***cv*** argument to the GridSearchCV constructor. Once completed, you can access the outcome of the grid search in the result object returned from ***grid.fit()***. The best_score_ member provides access to the best score observed during the optimization procedure and the best_params_ describes the combination of parameters that achieved the best results. We can adjust one parameter first and get its value, then fix the value and adjust other parameters.

5.1 Tune Batch Size and Number of Epochs

Part of the code listing is provided below.

```
def create_model():
    inp = Input(shape=(maxlen,))
    x = Embedding(max_features, embed_size, weights=[embedding_matrix])(inp)
    x = Bidirectional(LSTM(50,
                           return_sequences=True,
                           dropout=0.1,
                           recurrent_dropout=0.1))(x)
    # take the maximum of each column
    x = GlobalMaxPool1D()(x)
    x = Dense(50, activation="relu")(x)
    x = Dropout(0.1)(x)
    x = Dense(6, activation="sigmoid")(x)
    model = Model(inputs=inp, outputs=x)
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
    return model


model = KerasClassifier(build_fn=create_model, verbose=0)
batch_size = [10, 20, 40, 60, 80, 100]
epochs = [2, 3]
param_grid = dict(batch_size=batch_size, epochs=epochs)
```

Running above code produces the following output.

```
Best: 0.983149 using {'batch_size': 10, 'epochs': 3}
0.983149 (0.000182) with: {'batch_size': 10, 'epochs': 2}
0.983149 (0.000096) with: {'batch_size': 10, 'epochs': 3}
0.982789 (0.000508) with: {'batch_size': 20, 'epochs': 2}
0.982929 (0.000081) with: {'batch_size': 20, 'epochs': 3}
```

```
0.982734 (0.000247) with: {'batch_size': 40, 'epochs': 2}
0.983049 (0.000140) with: {'batch_size': 40, 'epochs': 3}
0.982290 (0.000366) with: {'batch_size': 60, 'epochs': 2}
0.982987 (0.000250) with: {'batch_size': 60, 'epochs': 3}
0.982424 (0.000386) with: {'batch_size': 80, 'epochs': 2}
0.982817 (0.000131) with: {'batch_size': 80, 'epochs': 3}
0.981629 (0.001014) with: {'batch_size': 100, 'epochs': 2}
0.982426 (0.000453) with: {'batch_size': 100, 'epochs': 3}
```

We can see that the batch size of 10 and 3 epochs achieved the best result of about 0.983149 accuracy. However, the accuracy is equal both 2 epochs and 3 epochs, the model start to converge when epochs is 2.

5.2 Tune Learning Rate

In the previous process, we've got the batch size and the epochs. Fix this two values when adjust Learning Rate. Part of the code listing is provided below.

```
def create_model(learn_rate=0.01):
    inp = Input(shape=(maxlen,))
    x = Embedding(max_features, embed_size, weights=[embedding_matrix])(inp)
    x = Bidirectional(LSTM(50,
                           return_sequences=True,
                           dropout=0.1,
                           recurrent_dropout=0.1))(x)
    x = GlobalMaxPool1D()(x)
    x = Dense(50, activation="relu")(x)
    x = Dropout(0.1)(x)
    x = Dense(6, activation="sigmoid")(x)
    model = Model(inputs=inp, outputs=x)
    optimizer = optimizers.Adam(lr=learn_rate)
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
    return model
```

```
model = KerasClassifier(build_fn=create_model, verbose=0)
learn_rate = [0.0001, 0.0005, 0.001, 0.01, 0.1, 0.2, 0.3]
param_grid = dict(learn_rate=learn_rate)
```

Running above code produces the following output.

```
Best: 0.982890 using {'learn_rate': 0.001}
0.980526 (0.000394) with: {'learn_rate': 0.0001}
0.982488 (0.000307) with: {'learn_rate': 0.0005}
0.982890 (0.000166) with: {'learn_rate': 0.001}
0.979806 (0.001517) with: {'learn_rate': 0.01}
0.963341 (0.000166) with: {'learn_rate': 0.1}
0.963341 (0.000166) with: {'learn_rate': 0.2}
0.963341 (0.000166) with: {'learn_rate': 0.3}
```

We can see that the learning rate of 0.001 achieved the best result of about 0.980526 accuracy.

5.3 Tune Dropout Regularization

Part of the code listing is provided below.

```
def create_model(dropout_rate=0.0):
    inp = Input(shape=(maxlen,))
    x = Embedding(max_features, embed_size, weights=[embedding_matrix])(inp)
    x = Bidirectional(LSTM(50,
                           return_sequences=True,
                           dropout=0.1,
                           recurrent_dropout=0.1))(x)
    x = GlobalMaxPool1D()(x)
    x = Dense(50, activation="relu")(x)
    x = Dropout(dropout_rate)(x)
```

```
    x = Dense(6, activation="sigmoid")(x)
    model = Model(inputs=inp, outputs=x)
    optimizer = optimizers.Adam(lr=0.001)
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
    return model


model = KerasClassifier(build_fn=create_model, verbose=0)
dropout_rate = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
param_grid = dict(dropout_rate=dropout_rate)
```

Running above code produces the following output.

```
Best: 0.983005 using {'dropout_rate': 0.1}
0.982768 (0.000310) with: {'dropout_rate': 0.0}
0.983005 (0.000318) with: {'dropout_rate': 0.1}
0.982954 (0.000356) with: {'dropout_rate': 0.2}
0.982914 (0.000223) with: {'dropout_rate': 0.3}
0.982553 (0.000244) with: {'dropout_rate': 0.4}
0.982403 (0.000129) with: {'dropout_rate': 0.5}
0.981882 (0.000178) with: {'dropout_rate': 0.6}
0.981953 (0.000363) with: {'dropout_rate': 0.7}
0.981270 (0.000341) with: {'dropout_rate': 0.8}
0.980103 (0.000445) with: {'dropout_rate': 0.9}
```

We can see that the dropout rate of 0.1 achieved the best result of about 0.983005 accuracy.

6 Environment Configuration

The task is done on the following platform:

- **Linux:** *Linux version 3.10.0-693.11.6.el7.x86_64 (builder@kbuilder.dev.centos.org) (gcc version 4.8.5 20150623 (Red Hat 4.8.5-16) (GCC)).*
- **IDE:** *pycharm-2017.3.2.*
- **Python:** *Python 3.6.3 | Anaconda custom (64-bit) | (default, Oct 13 2017, 12:02:49) [GCC 7.2.0] on linux.*
- **Deep Learning Framework:** *Keras==2.1.4.*

7 Implementation Setting

In this model, the learning rate for gradient descent, the batch size and the Number of Epochs are 0.001, 10 and 2, respectively. The hidden state size of Bi-LSTM is 50, the following fully-connected neural network layer have 50 hidden units. The output layer is a fully-connected sigmod layer with 6 outputs for each comment.

8 Experimental Results and Analysis

Table 1: Evaluation results for all compared methods

Method	Score
Bi-GRU	0.9768
GRU	0.9777
LSTM	0.9785
Bi-LSTM	**0.9787**

Table 1 shows the evaluation results. It is can be seen that the Bi-LSTM outperforms the other competitors. In general, the LSTM-based model is superior to the GRU model, however, this gap is very small. The results are given by the official website of Kaggle after my submissions, it is now evaluated on the mean column-wise ROC AUC. In other words, the score is the average of the individual AUCs of each predicted column.

8.1 Data Imbalance Problem

Oversampling and undersampling in data analysis are techniques used to adjust the class distribution of a data set. The other solution is that to choose a appropriate evaluation criteria, such as ROC or F1, not accuracy. A variety of data re-sampling techniques are implemented in the imbalanced-learn package compatible with Python's scikit-learn interface.

8.2 Overfitting and Solutions

In this task, we use the following methods to prevent overfitting:

- **Cross-validation:** Cross-validation is a powerful preventative measure against overfitting. In standard k-fold cross-validation, we partition the data into k subsets, called folds. Then, we iteratively train the algorithm on k-1 folds while using the remaining fold as the test set. A default cross-validation of 3 was used in grid search process.
- **Early Stopping:** Early stopping refers stopping the training process before the learner passes that point. After that point, however, the model's ability to generalize can weaken as it begins to overfit the training data.
- **Regularization:** In this model, We use dropout which is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data.

9 Conclusions

In this task, we employ four models (LSTM, GRU, Bi-LSTM and Bi-GRU) respectively to address toxic comment classification. Experiment results on test dataset demonstrate that our algorithms work well in the official evaluation metrics. Bidirectional RNN can take full account of the contextual semantic information of the text. In the future, we plan to improve our algorithms in the following aspects:

- **Stopwords:** Stopwords are some of the most common, short function words, such as the, is, at, which, and on. In the process of data preprocessing, we can remove stop words.
- **Other Pre-trained Word2Vec Models:** Such as Google's pre-trained Word2Vec model and Pre-trained vectors based on Wikipedia.
- **Resampling:** According to the previous analysis of the dataset we know that there is a sample imbalance problem in the training data. In the next stop, we can consider resampling the training data and then training our model.
- **Attention Mechanism:** Employ an attention mechanism to learn weighting for each word. Under the setting, key words will have a higher weight, and common words will have lower weight. Therefore, the representation of texts not only considers all words, but also pays more attention to key words.
- **Other Algorithms:** Like CNN, and some machine learning mothods such as SVM and Adaboost.

A Information of This Article

This report is edited by LaTeX. The attachment contains all the source code for this report and its directory structure is shown blow.

```
ToxicClassification
├── Data-analysis
│   ├── data-info.py
│   ├── freq_distribution.jpg
│   └── retio_labels.jpg
├── input
│   ├── glove6b50d
│   │   └── glove.6B.50d.txt
│   ├── sample_submission.csv
│   ├── test.csv
│   └── train.csv
└── My-model
    ├── Bi-GRU_model.py
    ├── Bi-LSTM_model.py
    ├── GRU_model.py
    ├── LSTM_model.py
    ├── Tune_batchsize_epochs.py
    ├── Tune_dropout_rate.py
    └── Tune_learning_rate.py
```