IMDB Review Classification

Cedric Yu[♮]

Abstract

We study the IMDB review dataset in the 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 perform sentiment classification of the given reviews, and compare the results and performance of different approaches.

1 Problem Statement

Sentiment classification belongs to a typical class of problems in natural language processing, which involves the classification of a given sequence of objects such as words, symbols or numbers into one or a few classes. These are many-to-one/few problems, in contrast to many-to-many problems such as machine translation. In the present problem, we are tasked with classifying the sentiment of each given movie review from IMDB into good or bad.

1.1 Datasets

We are given a labeled training set `labeledTrainData.tsv.zip` and an unlabeled test set `testData.tsv.zip`. Each review is a string of text in the HTML format. The training set contains 25,000 reviews. The label is the binary sentiment of each review: a rating of < 5 in the 10-point scaled IMDB rating is given a sentiment score of 0, and a rating of ≥ 7 is assigned a sentiment score of 1. No individual movie has more than 30 reviews. The test set contains 25,000 unlabeled reviews, none of which is on the same movie as any review the training set.

The sentiment in the training set has a mean of 0.5; we have balanced classes.

1.2 Evaluation Metric

The results will be evaluated by the area-under-the-curve (auc) of the Receiver operating characteristic (ROC) curve; we want to maximise the auc, so that we can maximise the true positive rate and minimise the false positive rate by adjusting the decision threshold by which we classify an instance into `0` or `1`.

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*. After the next section, we will discuss them one by one in detail.

[♮]cedric.yu@nyu.edu. Last updated: October 25, 2021.

2 Common Text Pre-Processing Steps

Each of the machine learning approaches requires a different way of text pre-processing. But for all these approaches, there are some common steps, which we discuss in this section.

To outline the strategy for pre-processing a given raw text, we look at an example; see `dataset_study.py`. The raw text of the third instance in the training set partly reads

```
'It must be assumed that those who praised this film (\\the greatest filmed opera ever
    ,\\" didn\'t I read somewhere?) either don\'t care for opera, don\'t care for
    Wagner, or don\'t care about anything except their desire to appear Cultured.
    Either as a representation of Wagner\'s swan-song, or as a movie, this strikes me
    as an unmitigated disaster, with a leaden reading of the score matched to a tricksy
    , lugubrious realisation of the text.<br /><br />It\'s questionable'

...

'the 1951 Bayreuth recording, and Knappertsbusch, though his tempi are often very slow
    , had what Jordan altogether lacks, a sense of pulse, a feeling for the ebb and
    flow of the music -- and, after half a century, the orchestral sound in that set,
    in modern pressings, is still superior to this film."'
```

We see that it contains HTML syntax, such as `<br /><br />` for line break, as well as various backslashes especially preceding a quotation mark. There are also punctuation marks such as **...** and `-`.

We thus adopt the following three text pre-processing steps for all our machine learning approaches:

1. Remove HTML syntax using the BeautifulSoup package
2. Keep only alphabets and full stops using regular expression: `re.sub(r'[^a-zA-Z_\'\s]+', ' ', text)`. We note that we also used other slight variables of this regex pattern which amount to keeping/dropping the single quotation mark, hyphen and digits; see the `.py` files. In the end, they made no noticeable difference.
3. lower cases with `text.lower()`

These are done in the function `text_preprocess()` for each instance, which are applied to each dataset using the function `pd_text_preprocess()`.

3 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.

3.1 Workflow

1. load datasets and clean text in `text_preprocess()` and `pd_text_preprocess()`

- get rid of HTML syntax using `BeautifulSoup`
- get rid of unimportant punctuation marks including apostrophe
- lower case
- lemmatise with `nltk.WordNetLemmatizer()`
- tokenise with `nltk.word_tokenize()`
- optional: remove stop words

2. train-validation split
3. fit and transform using `CountVectorizer` or `TfidfVectorizer` from `nltk`
4. fit the models: `LogisticRegression`, `MultinomialNB` and `XGBClassifier`
5. predict

They are implemented in `master_no_embedding.py`.

3.2 Pre-Processing

For the pre-processed text with stop words, we keep both digits, alphabets and full stops. For that without stop words, we drop the digits.

3.3 Train-Validation Split

We randomly split the 25,000 training instances into training set (80%) and validation set (20%).

3.4 Fitting the Vectorizers: Hyperparameters

For both vectorisers, we use `ngram_range = (1,5)`. We perform grid searches of the hyperparameters `min_df` and `max_df` using the training set with stop words. For both cases, the set of values that result in the largest auc from `LogisticRegression()` is `min_df=0, max_df=0.2`. We also use this set when we fit the vectorisers with the training set without stop words.

3.5 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(max_iter = 10000)`, the auc of the validation set with/without stop words are respectively 0.8875 and 0.8833. For `MultinomialNB(alpha = 0.1)`, the corresponding auc are 0.8776 and 0.8759. For `XGBClassifier`, the auc of the validation set with stop words is 0.8522; since it is lower than the those from the other two models, we did not consider the case without stop words.

Among them, `LogisticRegression` fitted on the training set with stop words gave the best auc. We find the 10 smallest and largest coefficients of this model to be respectively

'worst' 'awful' 'waste' 'boring' 'worse' 'terrible' 'poor' 'dull', 'horrible' 'avoid'

and

'excellent' 'wonderful' 'perfect' 'favorite' 'amazing' 'best' 'enjoyed', 'loved' ' fantastic' 'today'.

Thus, it appears that positive words are generally associated with large coefficients, vice versa.

Submitting the test predictions of this model on Kaggle, we gtt a test auc of 0.87424.

The results for `TfidfVectorizer` are as follows.

For `LogisticRegression(max_iter = 10000)`, the auc of the validation set with/without stop words are respectively 0.8939 and 0.8911. For `MultinomialNB(alpha = 0.1)`, the auc of the validation set with stop words is 0.8828. For `XGBClassifier`, the auc of the validation set with stop words is 0.8457.

Among them, `LogisticRegression` fitted on the training set with stop words gave the best auc. We find the 10 smallest and largest coefficients for this model to be respectively

'worst' 'awful' 'waste' 'boring' 'worse' 'poor' 'terrible' 'horrible' 'script' ' supposed'

and

'excellent' 'best' 'wonderful' 'perfect' 'love' 'amazing' 'favorite' 'loved' 'today' ' enjoyed'.

Thus, it appears that, again, positive words are generally associated with large coefficients, vice versa.

Submitting the test predictions of this model on Kaggle, we got a test auc of 0.88020.

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

4 LSTM with GloVe embedding vectors

We use LSTM neural networks on Tensorflow with GloVe embedding vectors [1]. Specifically, we use the 50-dimensional GloVe embedding vectors of 40,000 words.

4.1 Workflow

1. load GloVe vectors
2. load datasets and clean text
    - get rid of HTML syntax using `bs4.BeautifulSoup`
    - get rid of unimportant punctuation marks including apostrophe
    - lower case
    - lemmatise with `nltk.WordNetLemmatizer()`
    - tokenise with `nltk.word_tokenize()`

- (remove stop words)
- set maximum sequence length and pad with `'-1 empty'`

3. train-validation split
4. map words in pre-processed datasets to GloVe indices, save to files
   unknown/padded words are set to index = 0, which will be mapped to zero embedding vectors
5. define model: embedding –> layers of Bidirectional LSTM –> Fully Connected layers with tanh –> Dropout –> output with sigmoid
6. optimise with Adam
7. predict

They are implemented in `master_embedding_LSTM.py`.

4.2 GloVe vectors

We first load the GloVe embedding vectors from `glove.6B.50d.txt` using the function `read_glove_vecs`. 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; see below.

4.3 Pre-Processing

For the case keeping stop words, we keep alphabets, digits, hyphen and full stops; for the other case, we only keep alphabets and full stops. The lists of tokens are padded by the token `'-1 empty'` to a maximum length of either `max_len=2700` (with stop words) or `max_len=1500` (without stop words). The maximum lengths are chosen given the fact that, after the aforementioned steps, there are at most 2600 (with stop words) and 1416 (without stop words) tokens in each instance of the datasets. These are all done in the function `text_preprocess`.

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; see next sub-section.

The arrays of word indices are saved to files to save time in future evaluations.

4.4 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`.

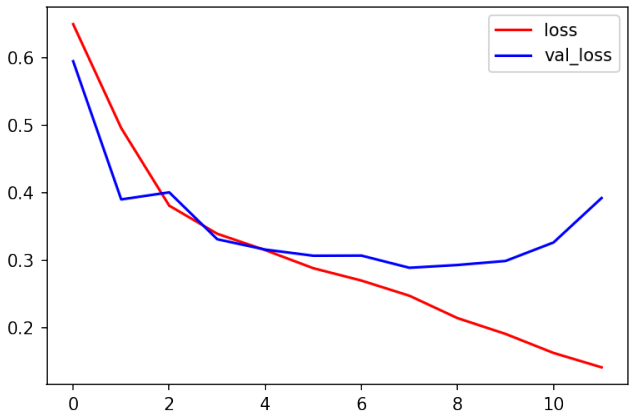
In the function `sentiment_classification_model`, we define our LSTM model. The model takes as input a (batch size, `max_len`) array of word indices. It is then passed to the pre-trained embedding layer, followed by layers of bi-directional LSTM layers, then Dense fully connected layers with tanh activation, a Dropout layer and finally the output layer with sigmoid activation. We optimise the model on the binary cross-entropy with Adam.

4.5 Model Validation and Test Performance

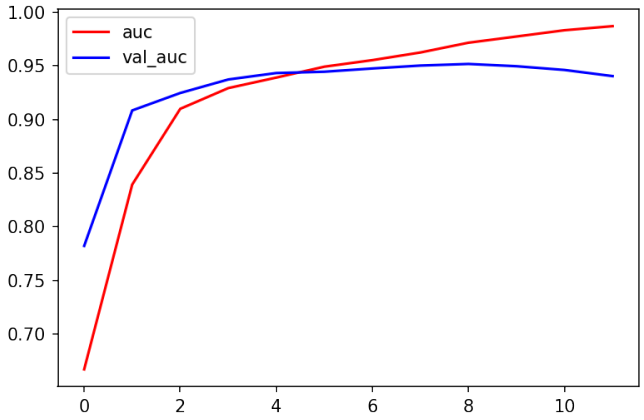
We experimented with several architectures of the neural network, and found that the best performance was achieved with two bi-directional LSTM layers.

For the pre-processed sets with stop words (`max_len=2700`), we considered the network:
Embedding —> (Bidirectional LSTM with 128 units) x 2 —> Dropout(.2) —> Dense layer with 64 units and tanh activation —> output layer with sigmoid activation.
With the default `learning_rate=0.001` in Adam, we achieved the highest validation auc of 0.9518 after the 9-th epoch. The training curves are shown in Figure 1.

The test set auc is 0.85080.



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

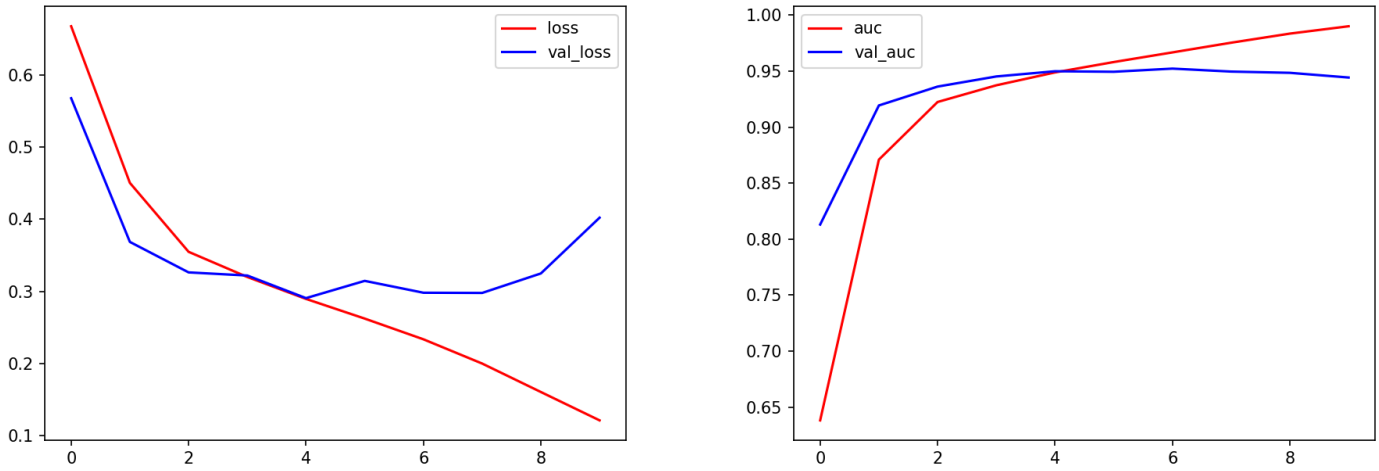


(b) Training (red) and validation (blue) auc.

Figure 1: Training curves for the model with two layers LSTM with 128 units, with stop words.

With stop words, we also considered the network:
Embedding —> (Bidirectional LSTM with 256 units) x 2 —> Dropout(.2) —> Dense layer with 64 units and tanh activation —> output layer with sigmoid activation.
With the default `learning_rate=0.001`, we achieved the highest validation auc of 0.9495 after the 8-th epoch. The training curves are shown in Figure 2.

The test set auc is 0.87644, which is higher than the previous case despite the slightly lower validation auc.



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

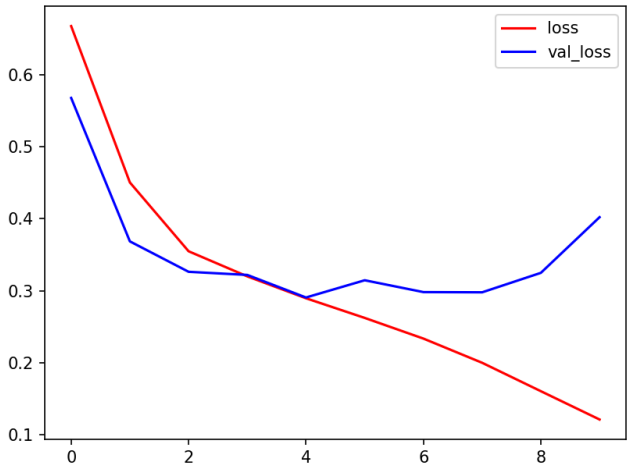
(b) Training (red) and validation (blue) auc.

Figure 2: Training curves for the model with two layers LSTM with 256 units, with stop words.

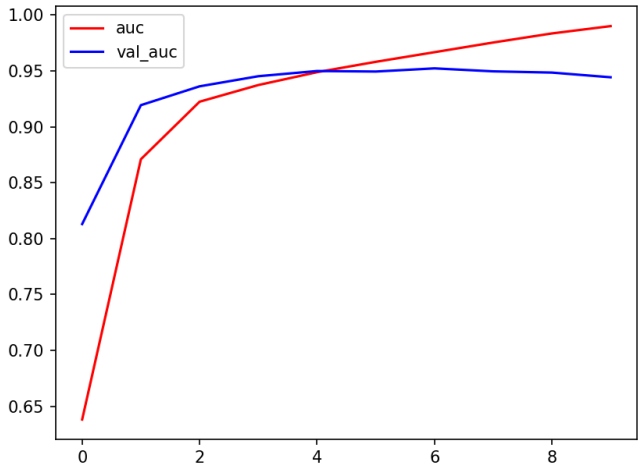
We also considered a deeper network: Embedding —> (Bidirectional LSTM with 128 units) x 3 —> Dense layer with 64 units and tanh activation —> Dense layer with 16 units and tanh activation —> Dropout(.2) —> output layer with sigmoid activation.
Using a smaller `learning_rate=0.00005`, after 25 epochs, the validation auc lingered at around 0.89. Since this did not lead to better performance and took much longer to train, we did not continue.

Without stop words (`max_len=1500`), we considered the network:
Embedding —> (Bidirectional LSTM with 256 units) x 2 —> Dropout(.2) —> Dense layer with 64 units and tanh activation —> output layer with sigmoid activation.
With `learning_rate=learning_rate=0.0001`, we achieved the highest validation auc of 0.9183 after the 17-th epoch. The training curves are shown in Figure 3.

The test set auc is 0.83696.



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



(b) Training (red) and validation (blue) auc.

Figure 3: Training curves for the model with two layers LSTM with 256 units, without stop words.

Thus, once again, we find that keeping stop words leads to better scores.

5 Hugging Face Transformer

We use `BertTokenizerFast` and `TFBertForSequenceClassification`, and the pre-trained parameters from `bert-base-uncased`, all from Hugging Face Transformers [2].

Since a transformer network, heuristically, works by learning the query and key relations between tokens in a given sentence, we expect intuitively that the form of the word (without stemming) and stop words such as prepositions could be important. As such, we only perform minimal text pre-processing— only those mentioned in Section 2.

It is also important that we use the tokeniser associated to the transformer model; here we use `BertTokenizerFast` instead of the `nltk` tokenizer.

5.1 Workflow

1. load datasets and clean text
    - get rid of HTML syntax using `bs4.BeautifulSoup`
    - keep alphabets and full stops only
    - lower case
2. train-validation split
3. tokenise with `BertTokenizerFast`
4. load pre-trained `TFBertForSequenceClassification`

5. re-train the model with Adam for two epochs
6. predict

They are implemented in `master_transformer.py`.

5.2 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 sub-word tokenised by `BertTokenizerFast`. Here, we do not have this trouble because the target of each entire sentence is one single label.

5.3 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 1/2
5000/5000 [==============================] - 1641s 326ms/step - loss: 0.2711 - auc:
    0.9547 - val_loss: 0.1858 - val_auc: 0.9787
Epoch 2/2
5000/5000 [==============================] - 1630s 326ms/step - loss: 0.1373 - auc:
    0.9872 - val_loss: 0.2475 - val_auc: 0.9757
```

The validation auc, 0.9757, is much higher than that from any of the previous approach. Indeed, the test auc is 0.92304, also the highest.

6 Discussion

We have analysed the problem in three different approaches. In the end, the most cost-effective one turns out to be the `TfidfVectorizer+LogisticRegression` combination, trained with text with stop words. The fact that the non-sequential approaches (`CountVectorizer` and `TfidfVectorizer`) give comparable scores to the neural network approaches (LSTM and transformer) suggests that, in this particular problem, the order of the words does not matter very much. And in this case, a simple logistic regression sufficed; no need for a deep neural network.

For the LSTM models, we find that two bi-directional LSTM layers followed by one dense layer and a dropout layer gives the best scores. Further increasing the number of LSTM layers decreases the model performance and significant increases training time. It appears that the model has a hard time learning the parameters in the deep layers. Increasing the number of units in each layer improves the scores, but it comes at a computational cost: each epoch takes longer to train and has to be done with a much (an order of magnitude) smaller learning rate.

The transformer network gives the best scores. On the other hand, the re-training time is long despite the few epochs required. More importantly, it takes significantly longer to make predictions.

All in all, it is doubtful, for the purpose of classifying movie reviews, whether the slight improvement in performance of the neural network-based approaches justifies the much higher computational costs. My personal take is, the simple, good old `TfidfVectorizer+LogisticRegression` is the winner.

7 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.