Machine Learning Project 2

Yifei Song, Haoming Lin, Ruiqi Yu
École Polytechnique Fédérale de Lausanne, Switzerland

***Abstract*—This project investigates the application of machine learning as well as deep learning to text sentiment classification. By pre-processing negative and positive texts, word embedding and model training, we obtain some algorithmic models that can handle text sentiment binary classification well. We also compare the results for different methods of pre-processing and different algorithms. The best model we obtain is based on DistilBERT algorithm, which can reach an accuracy of 85.9%.**

I. Introduction

In recent years, the rise of big data has helped extensive data applications in many fields. However, eighty percent of data are only in unstructured form [1], and the situation prohibits data applications. Making the data structured is one critical problem that is highly related to the quality of analysis and eventual decision. [2] Text classification can structure the data and is one track of natural language processing (NLP). Nowadays, various methods have been developed to deal with NLP tasks. In this project, we use different machine learning methods to solve the language sentiment classification problem, namely SVM, Random Forest, fastText, BERT, and RoBERTa.

The report is organized as follows. Section II describes our original data files and different pre-processing methods used for our models. Section III introduces five models studied and some implementation details. Section IV shows optimization techniques for our model training and hyper-parameters tuning. Section V displays the performances of the models and studies the role of pre-processing.

II. Pre-Processing

A. Datasets

Two datasets are given, one for the negative emotions and the other for the positive emotions. We label the negative dataset being treated as -1 and the positive as 1. We then combine the datasets and labels as our training data and labels. Also, to facilitate testing, the variable Batch_Number is used to control the size of the datasets.

B. Potential Useless Characters

After observing the sentences in the dataset, we found that some words or characters are not helpful or produce side effects for training, so we need to delete or replace them. We tried the following methods:

HTML elements: The dataset has many HTML elements, such as HTML tags like $\langle td \rangle$ and $\langle li \rangle$. We use the get_text method of the BeautifulSoup library to remove the HTML elements.

Punctuations and digitals: Some punctuation marks and numbers may be irrelevant to the expression of emotion, or they may be weakly associated with emotion. Therefore, we removed all punctuation marks and numbers common in string types.

Stopwords: In language-related tasks, removing stopwords is a common method. To implement this method, we followed the stopwords in the nltk.corpus library [3] to remove the stopwords in the dataset.

C. Stemming

The morphology of words may impact the model's training effect, such as the singular and plural of nouns and the different tenses of verbs. Therefore, we try to perform word morphology reduction for all words. To achieve this, we tag each word with the help of nltk library [4] and then perform word reduction according to the tags.

D. Tokenization

To process text more accurately, we need to tokenize the original text. To implement the tokenization, we use word_tokenize from the nltk library[4], and the pre-trained Bert tokenizer provided in the Tokenizers library [5], respectively.

E. Word Embeddings

To convert the text into data that the model can recognize, we do vectorization (word embedding) for the original text. We tried the following different vectorization methods:

Count Vectorizer: CountVectorizer only considers the frequency of each word in the training text and generates a word frequency matrix based on the frequency of each word with dimensions (number of words, number of documents). To implement it, we use the CountVectorizer library from sklearn.feature_extraction.text [6].

TF-IDF Vectorizer: Term Frequency-inverse Document Frequency(TF-IDF) is a statistical analysis method for keywords to assess the importance of a word to a document set or a corpus. It also generates a word frequency matrix similar to the one that appears in CountVectorizer, but with TF-IDF values instead of frequencies.

N-gram: N-Gram is an algorithm based on a statistical language model. Its basic idea is to operate a sliding window of size N on the content inside the text, forming a sequence of segments of length N. It considers not only center words but can also view the connections between the context and center words, which helps a lot in solving semantic problems. We will find the appropriate N by the method of Grid Search.

Word Embeddings: Word embedding can directly transform high-dimensional data into a relatively low-dimensional vector space. In our experiments, we use pre-trained models provided by Sentence-Transformer, such as all-mpnet-base-v2[7], which maps sentences and paragraphs to a 768-dimensional dense vector space. In addition to this, we also used other methods, such as fastText for word embedding, more details of which will be mentioned later.

F. BERT based models

Unlike shallow models, BERT-based models also consider meta-data in sentences, such as punctuations and morphology of words. Therefore the pre-processing data approach is slightly different for these models.

Each of the original datasets will be cleaned with the following steps:

1) remove HTML tags;
2) reduce spaces: every sequence of white-space characters in the sample is reduced to a single space character (' ');
3) remove duplicates in the file.

To facilitate our training process, we also wish to split offline our datasets into training and test files separately. The original datasets, containing positive and negative samples, respectively, are pre-processed as follows:

1) for each of the datasets, perform the previous cleaning;
2) remove from each dataset, intersection between positive dataset and negative dataset;
3) combine two datasets and store each sample alone with its label ('0' for negative and '1' for positive);
4) randomly split the combined datasets into training and test datasets.

The second part of the pre-processing consists of tokenization. Every tokenizer is trained together with the BERT model. It splits a sentence into a list of tokens before transforming it into a list of vectors. Special tokens will be added to the list to mark the sentence's beginning and end. Each sentence will be truncated or padded to have a length of 60. So every sentence will be transformed into a tensor of shape $(60, N)$ where N depends on the BERT model and is compatible with the input of the BERT.

III. Models

A. SVM

Support Vector Machines (SVM) are typical learning models for binary classification. Classifiers based on SVM have good potential for test classification. [8] The basic idea of SVM learning is to solve for the separated hyperplane that correctly partitions the training data set and has the most significant geometric separation.

We adopt "linear" as the kernel function to improve the training speed. We implement the classification model using the LinearSVC library from sklearn.svm [9].

B. Random Forest

Random Forest is also a conventional learning model. According to one specific comparison experiment, Random Forest is the best classifier out of 179 classifiers. [10] Moreover, Random Forest can migrate the high-dimensional, sparse, and noisy feature space problems in the data and, thus, is a high performer in text classification. [11]

We implement the classification model using the RandomForestClassifier library from sklearn.ensemble[12]. To speed up our training process, we determine the hyperparameter n_jobs as -1. And we also optimize the hyperparameter n_estimator using Grid Search.

C. fastText

fastText is an open-sourced word vector calculation and text classification tool. [13] There are three features of fastText. First, fastText is orders of magnitude faster than deep networks in training time while their accuracy is similar. Second, unsupervised learning of Sentence(Word) Vectors can quickly complete word embedding. Third, N-gram can show the relationship between words and words and increase the accuracy by 5%. [14] fastText uses context words to predict central words, whose structure is similar to the CBOW model architecture of word2vec.

We implement the classification model using the fastText library [13] directly. We determine the hyperparameter epoch as 5 and the hyperparameter wordNgram as 5 using Grid Search. We have additionally tagged the documents to support the fastText input format.

D. DistilBERT

The BERT is a transformer-based architecture that uses a bidirectional approach to consider the full context of words in a sentence. [15] Since its conception, this model has achieved great success in natural language processing (NLP). In particular, the BERT model has trained alone with its tokenizer. This tokenizer splits a string into a list of tokens before transforming the latter into a list of vectors. The context is taken into account because the vector representation of the same word could be different when they appear in other sentences.

The model takes the tensor computed from the tokenizer as inputs and outputs hidden states at each position. More precisely, if the input is of shape (b, M, N) where b is the batch size, and N is the length of the sequence, the output will have shape $(b, M, 768)$. For our use, we will only consider the first token in each returned sequence, which corresponds to the hidden state of a special token (referred to as `[CLS]` in the original paper) added at the beginning of the sequence. This value is believed to contain the meaning of the whole sentence.

To speed up our training process, we used pre-trained distilBERT [16]. Compared to the standard BERT, this distilled version is shown to run much faster while preserving most of the BERT's performance.

We define the top-classifier as a one-hidden-layer neural network.

The hidden layer is a length of 128, and the output size is 1. A dropout layer with dropout rate of 0.1 is added after the hidden layer. Our final model consists of a pre-trained distilBERT and a top classifier. Input data will pass through the distilBERT, hidden state of `[CLF]` token will be collected and passed as input to the top-classifier. The output computed by the top classifier is the final output of our model.

E. RoBERTa

RoBERTa (Robustly Optimized BERT Pretraining approach) is a variant of the BERT model. It is designed to be an improved version of BERT and achieves better performance on natural language processing tasks. [17] We will be using the pre-trained model available on hugginggface.

This pre-trained model is to perform feature engineering, and our final model is constructed by adding a top classifier in the same way as for BERT.

IV. Optimization Tricks

A. Grid Search

Grid search uses each set of hyperparameters to train the model and selects the hyperparameter with the lowest error (highest accuracy) in the validation set as the best. We have used this method extensively in our projects.

B. Dimension Reduction

In implementing the word frequency-based vectorization method, we note that the generated word frequency (word-document) matrix tends to be very sparse, and the matrix size depends on the size of the lexicon. To speed up our training process, we use SVD decomposition to reduce the dimensionality of the matrix.

By SVD decomposition, we can get the singular value matrix which is arranged from largest to smallest, and the decrease of singular values is particularly fast; in many cases, the sum of the first 10% or even 1% of the singular values accounts for more than 99% of the sum of all the singular values. In other words, we can also approximate the matrix by the largest k singular values and the corresponding left and right singular vectors:

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V_{n\times n}^T \approx U_{m\times k}\Sigma_{k\times k}V_{k\times n}^T$$

where Σ is the singular value matrix.

C. Fine-tuning of BERT-based models

One difficulty of our model training is the size of the dataset. The whole dataset is too large to fit in memory simultaneously. While another possible solution is to perform tokenization locally and to have all the tokenized data stored in disk (~10G), we decided to take another approach that sacrifices some training speed in exchange for memory.

During our training process, only cleaned raw (not tokenized) texts are loaded into memory and every iteration is as follows:

1) a batch of `batch_size` tweets are randomly sampled from the datasets;
2) the tokenizer performs tokenization on this batch and outputs a tensor of shape (`batch_size`, 60, N);
3) the model (BERT base + top-classifier) make a prediction and output a tensor of shape (`batch_size`, 1);
4) compute loss and optimize weights with `optimizer`.

Our training process consists of two steps. In the first step, only the parameters of the top-classifier are updated, and the parameters of the BERT remain unchanged. This could be achieved by passing only the parameters of the top-classifier to the torch optimizer object. In the second step, we use another optimizer which includes additional parameters of the BERT model. In this stage, we will be using a lower learning rate. We also follow the strategy proposed by Devlin et al. to perform linear decay on the learning rate. [15]

Validation data during the training process are loaded from a separate file. The creation of such a file is explained in the preprocessing section. The hyperparameter `validation_per` defines the interval at which the validation process takes place. At the end of every `validation_per` iterations, validation loss and accuracy will be computed over `validation_per` × `batch_size` tweets which will be randomly sampled from the validation set. The training loss (resp. accuracy) is calculated as the average loss (resp. accuracy) over the last `validation_per` iterations.

V. Experimental Results

In this section, we display the performance of different methods.

A. Convergence of BERT-based models

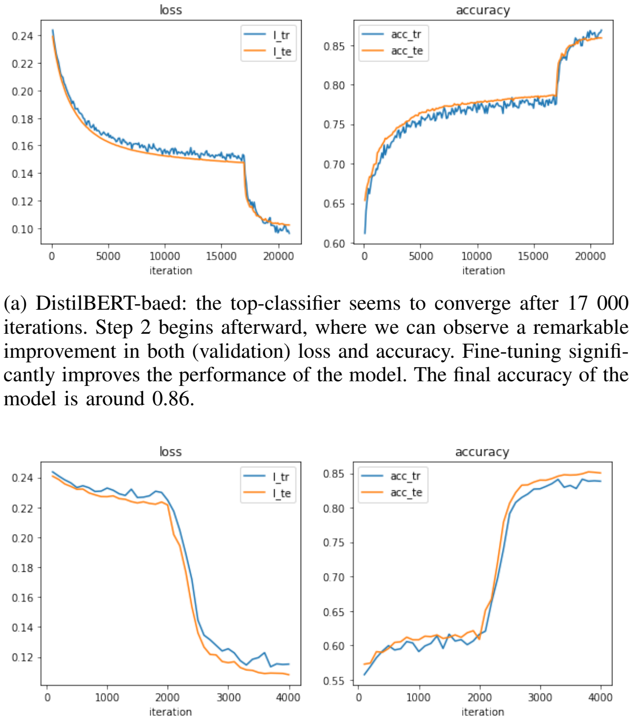
The hyperparameters of BERT-based models are shown in Table I. The choice of iteration numbers is made based on the convergence of the learning process. The convergence of these models can be found in Figure 1.

B. Comparison of different methods

The experiment results for different models can be found in Table II. All time measurements are based on Google Colab Pro+, with GPU acceleration available. Our two BERT-based are trained with the help of GPU, while the

	DistilBERT	RoBERTa
lr (step 1)	fixed at 10^{-4}	fixed at 10^{-4}
lr in step 2	initial at 6×10^{-6}	initial at 6×10^{-6}
batch size (step 1 and 2)	128	128
maxiter (step 1)	17000	2000
maxiter (step 2)	4000	2000

Table I: hyperparameters when training BERT-based models



(a) DistilBERT-baed: the top-classifier seems to converge after 17 000 iterations. Step 2 begins afterward, where we can observe a remarkable improvement in both (validation) loss and accuracy. Fine-tuning significantly improves the performance of the model. The final accuracy of the model is around 0.86.

(b) RoBERTa-based: the model converges after 2000 iterations in the first step and after 2000 iterations in the step 2. It achieves an accuracy of around 0.85 at the end.

Figure 1: Convergence of BERT-based models: `l_tr` and `l_te` indicate training and validation loss

other three methods only support CPU. SVM, Random Forest, and FastText require pre-processing data with word embedding, which was also performed on the CPU. DistilBERT and RoBERTa are run for a different number of iterations which are determined by the convergence of the models. As mentioned previously, pre-processing for BERT-based models is performed on the fly. Therefore the time is measured together with training time.

The accuracy is measured on the dataset where positive and negative samples have roughly the same frequency. So we can set the baseline of performance at 0.5 for accuracy. We can observe from the results that shallow learning methods such as SVM and Random Forest can achieve an accuracy of 0.8 on unseen data when data is pre-processed with well-chosen word embedding methods.

Methods	Preprocess time	Training time	Total time	Accuracy
LinearSVM	70 min	3 min	73 min	0.832
Random Forest	70 min	20 min	90 min	0.813
FastText	-	-	1 min	0.851
DistilBERT	-	-	60 min	0.859
RoBERTa	-	-	20 min	0.858

Table II: Performance result of different methods.

Methods	Preprocess time	Accuracy
Baselines	60 s	0.843
Remove Punctuation	55s	0.812
Stemming	105 s	0.832
Remove Stopwords	60 s	0.828
Tokenization	60 s	0.839

Table III: Performance result of pre-processing methods

We believe that the word embedding captures most of the features in original sentences since the linear kernel SVM is able to achieve 0.83 accuracy, which is a comparable result to BERT-based models.

For BERT-based models, the convergence of DistilBERT-based models is much slower than our RoBERTa-based model, as we can see from Figure 1. It was therefore trained much more iterations.

C. *Impact of pre-processing methods on the performance of models*

In this section, we study how these pre-processing methods affect the final results. The experiment training results of small batch size (40000) for different pre-processing methods based on LinearSVM can be found in Table. III. We find that too much text preprocessing does not improve training performance, especially removing punctuation and removing stopwords, significantly reducing prediction accuracy. The use of stemming or changing tokenization methods did not have a significant impact on accuracy. We conjecture that this is because the text dataset is from the web and not from official articles or publications. Therefore, it is more flexible and concise in terms of words and will use special symbols including punctuation to express emotions.

VI. Conclusions

We have studied different machine learning algorithms for sentiment analysis. We have shown that shallow methods such as SVM and Random Forest could also achieve an accuracy as high as 0.8 on unseen data if we use an appropriate word embedding method. This is a surprising result, considering the simplicity of these models and the complexity of the task. As for deep learning methods, BERT-based models have a very slow fine-tuning process compared to FastText, but performs only slightly better than the latter to our experiments.

References

[1] A. Khan, B. Baharudin, L. H. Lee, and K. Khan, “A review of machine learning algorithms for text-documents classification,” *Journal of advances in information technology*, vol. 1, no. 1, pp. 4–20, 2010.

[2] M. Thangaraj and M. Sivakami, “Text classification techniques: A literature review,” *Interdisciplinary Journal of Information, Knowledge, and Management*, vol. 13, p. 117, 2018.

[3] “nltk.corpus package.” [Online]. Available: https://www.nltk.org/api/nltk.corpus.html

[4] “Natural language toolkit.” [Online]. Available: https://www.nltk.org/index.html

[5] “Tokenizers.” [Online]. Available: https://pypi.org/project/tokenizers/

[6] “Sklearn.feature_extraction.text.countvectorizer.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

[7] “sentence-transformers/all-mpnet-base-v2.” [Online]. Available: https://huggingface.co/sentence-transformers/all-mpnet-base-v2

[8] A. Sun, E.-P. Lim, and W.-K. Ng, “Web classification using support vector machine,” in *Proceedings of the 4th international workshop on Web information and data management*, 2002, pp. 96–99.

[9] “Sklearn.svm.linearsvc.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[10] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, “Do we need hundreds of classifiers to solve real world classification problems?” *The journal of machine learning research*, vol. 15, no. 1, pp. 3133–3181, 2014.

[11] M. Z. Islam, J. Liu, J. Li, L. Liu, and W. Kang, “A semantics aware random forest for text classification,” in *Proceedings of the 28th ACM international conference on information and knowledge management*, 2019, pp. 1061–1070.

[12] “Sklearn.ensemble.randomforestclassifier.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[13] “Fasttext.” [Online]. Available: https://fasttext.cc/

[14] Chercheer, “Fasttext algorithm principle and use,” Dec 2019. [Online]. Available: https://blog.csdn.net/CherDW/article/details/103413587

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[16] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” *arXiv preprint arXiv:1910.01108*, 2019.

[17] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.

Appendix

A. Pre-Processing

Here are some examples of pre-processing in II.

Before	After
buat cewe 2 .. find the love who would fight for you and ur relationship . and when u find that person , fight for him . never give up\n	buat cewe find love would fight ur relationship u find person fight never give
this feeling sucks i miss my 2 favorite teams !\n	feeling suck miss favorite team
#pacerswinagain - for those counting at home , tonight is the pacers seventh consecutive victory (11-1 in april pacers 118 , <user>109 .\n	pacerswinagain count home tonight pacer seventh consecutive victory april pacer
<user>uhm no lorans fake noone of the cyrus know her\n	uhm lorans fake noone cyrus know
late start tomorrow . oh thank god\n	late start tomorrow oh thank god

Table IV: Text Before And After Pre-processing

B. Optimization Trick

We mentioned the grid search in IV-A. Here is an example of optimizing the hyperparameters wordNgram and epoch of method fastText using grid search as shown in Figure 2.

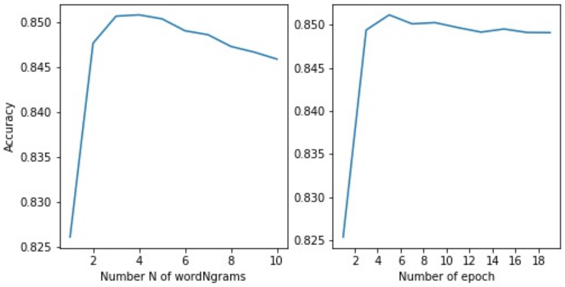


Figure 2: Training hyperparameters of fastText by grid search.