CS410 Project Detail

1. Team Information

Team Name:

Team Commonwealth

Team Members:

1. Zuliang Weng / zwe
2. Zijing Chen / zijingc3
3. Liping Xie / lipingx2 (captain)

2. Project Overview

2.1 Project Topic

Text Classification Competition: Twitter Sarcasm Detection.
The Classification task is to classify the list of Tweets into two categories: “SARCASM” or “NOT_SARCASM”. There are 2 datasets: Training set which with 5000 Tweets and Test set which with 1800 Tweets. We researched and tried many cutting-edge models that are suitable for classifying “SARCASM” or “NOT_SARCASM” in this project.
There are two files provided: train.jsonl and test.jsonl. The model is trained with the data provided in train.jsonl , and test.jsonl contains the tweets that we need to classify with the trained model. The classification result is reported with the provided id. All the results are stored in the answer.txt file.

2.2 Overview of the Function of the Code

For this classification task, we used PyTorch as the Deep learning framework, Python(version 3.6.9) as the Programming Language and Pre-trained BERT as the State-of-the-art neural network classifier.

The main function of the code is to classify the list of Tweets into two categories: “SARCASM” or “NOT_SARCASM”. To be more specific, we applied the pre-trained Twitter-specific BERT model, roBERTa-base model on the training data to fine-tune the model and then predicted on the testing data to generate final results. This code can only be used to analyze or classify Tweets data due to the Twitter-specific BERT model we used.

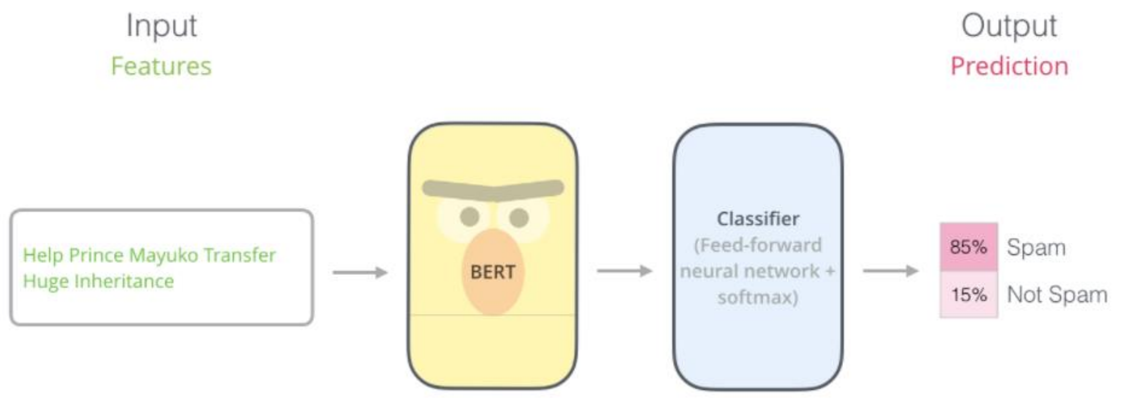
Our code is in file “Final BERT Twitter Sarcasm Classification.ipynb”, here is the detail mappings of the code section and the content:

Code Section No.	Content
1	How to use Google Colab for training the model and perform the test under python 3.6.9 environment and install the Hugging Face Library
2	How to load data from the provided jsonl files and parse the data
3	How to use the pre-trained BERT model. How to formatting the data, tokenize Dataset and split the training data into training set and validation set for the use of BERT
4	How to train and fine-tune the model with different batch size, learning rate and epochs, and how to evaluate the result
5	How to use the trained model to predict the tweets provided in the test.jsonl. How to save the prediction result.

3. Software Implementation Details

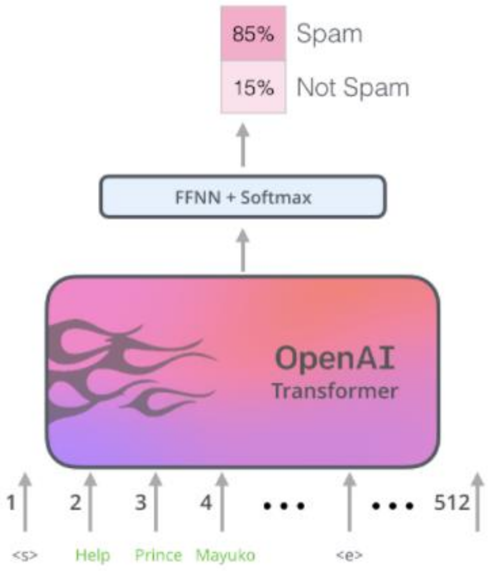
3.1 Classifier Selection and Introduction

We have investigated different types of Classifiers which include Word2Vec, FastText and BERT. We found BERT (Bidirectional Encoder Representations from Transformers) is the State-Of-The-Art Neural Network Classifier.
According to Wikipeda, BERT is a Transformer-based machine learning technique for natural language processing (NLP) pre-training developed by Google. BERT is a model that broke several records for how well models can handle language-based tasks(Jay Alammar, 2018).
This model would look like this:


Input
Features
Output
Prediction
Help Prince Mayuko Transfer Huge Inheritance
BERT
Classifier (Feed-forward neural network + softmax)
85% Spam
15% Not Spam

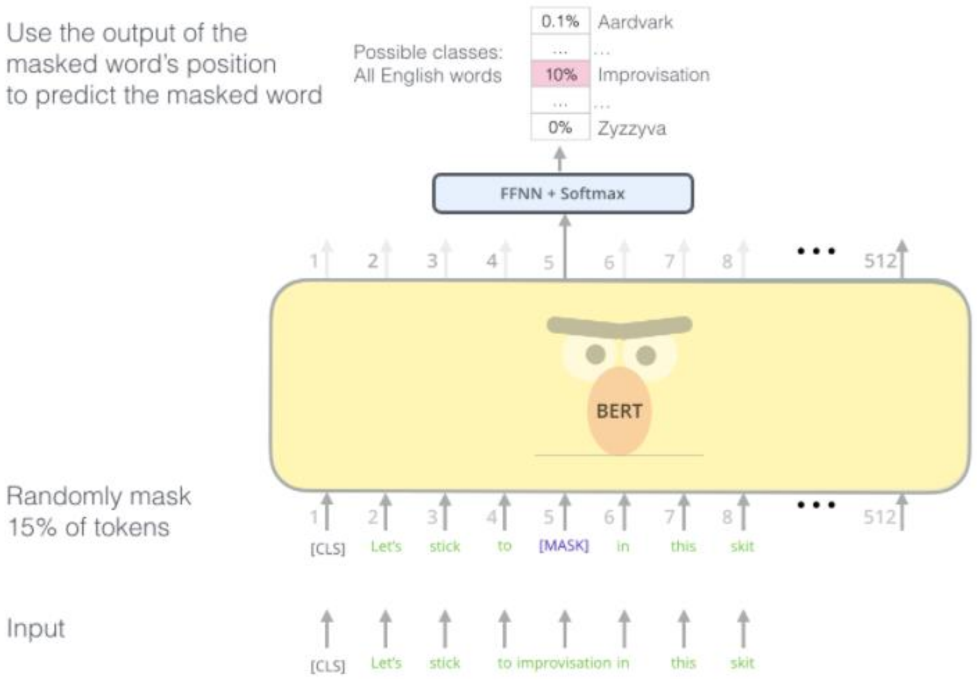
(Above information and image from:http://jalammar.github.io/illustrated-bert/)

To train such a model, we mainly have to train the classifier, with minimal changes happening to the BERT model during the training phase. This training process is called Fine-Tuning (Jay Alammar, 2018). BERT makes use of the encoder mechanism of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. At the moment, the Transformers package from Hugging Face PyTorch library is regarded as the most widely accepted and powerful pytorch interface for working with BERT.



(Above information and image from:http://jalammar.github.io/illustrated-bert/)

BERT's clever language modeling task masks 15% of words in the input and asks the model to predict the missing word (Jay Alammar, 2018).


Use the output of the masked word's position to predict the masked word
Possible classes: All English words
0.1% Aardvark
10% Improvisation
0% Zyzzyva
FFNN + Softmax
BERT
Randomly mask 15% of tokens
[CLS]
Let's
stick
to
[MASK]
in
this
skit
Input
to improvisation in

(Above information and image from:http://jalammar.github.io/illustrated-bert/)

3.1.1 Model Selection

Training a BERT model takes huge time, we can use the pre-trained model, and fine-tune it with our training data. With the consideration of classifying Twitter text, we used the pre-trained Twitter-specific BERT model - roBERTa-base model. The reason why we choose Twitter-specific BERT model is that the characteristics of Tweets are significantly different from traditional text such as research papers or articles. Tweets are generally short, and include frequent use of informal grammar as well as irregular vocabulary such as abbreviations. Thus, it would be inappropriate to apply language models such as bert-base-uncased, bert-large-uncased that are pre-trained on traditional text with formal grammar and regular vocabulary to analyze Tweets data (Nguyen et al., n.d.). Our testing result confirmed that the roBERTa-base model outperforms bert-base-uncased and bert-large-uncased in this classification task.

The architecture for BERTweet is very similar to BERTbase, which is trained with a masked language modeling objective (Devlin, 2019). BERTweet pre-training procedure is based on RoBERTa which optimizes the BERT pre-training approach for more robust performance (Liu, 2019). For pre-training data, BERTweet uses an 80GB pre-training dataset of uncompressed texts, which contain 850M Tweets (Nguyen et al., n.d.). This is very similar to the training and testing dataset since our data is also Tweets data.

Under https://huggingface.co/models, we can find there are different sub models, we compared the performance between twitter-roberta-base and twitter-roberta-base-irony, we found twitter-roberta-base performs a little bit better on the prediction task (we have provided the detail test result in section 3.7.2), we decided to select twitter-roberta-base in our final version code.

3.2 Environment setup

3.2.1 Setup Colab

Since we need to train a large neural network, Google Colab offers free GPUs and TPUs, we used Colab to train our model to shorten our training time.

Steps for setting up Colab:

1. Add Google Colab as an app to Google Drive.
2. Set the runtime type to GPU. To be more specific, we went to "Edit" -> "'Notebook Settings" -> "Hardware accelerator" -> "GPU". If you want to check if the GPU is detected, execute code in section 1.1 to confirm
3. We need to identify and specify the GPU as the device in order for torch to use the GPU. As a user, you need to install the package torch If you have not installed it before. After importing the package torch, you can run the second section of code in 1.1 to specify the GPU as the device.

3.2.2 Install the Hugging Face Library

We have selected the Transformers package from Hugging Face PyTorch library, so we need to install the transformers package from Hugging Face which gives us a pytorch interface for working with BERT. You can run the code in section 1.2 to install the Hugging Face Library.

3.3 Data Analysis and Preparation

There are 2 json format datasets provided:

- train.jsonl: 5000 Tweets
- test.jsonl: 1800 Tweets

In each tweet in test.jsonl, it contains “id”, “response” and “context”:

- Id: String identifier for sample. This id will be required when making submissions.
- Response: the Tweet to be classified
- Context: the conversation context of the *response*. The context is an ordered list of dialogue

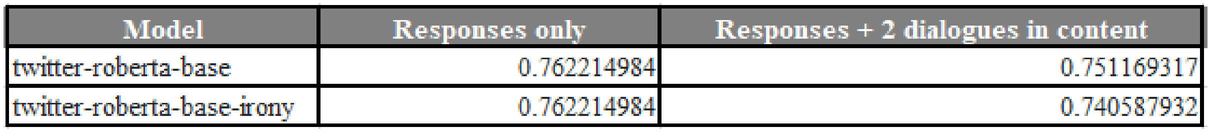
In each tweet in train.jsonl, it contains “label”, “response” and “context”:

- Label: SARCASM or NOT_SARCASM
- Response: the Tweet to be classified
- Context: the conversation context of the *response*. The context is an ordered list of dialogue

The length of all responses are less than 150 words, but for the context, some tweets with very long context and exceed the maximum supported input size (512) of BERT, that means we cannot use the full content of context for classification. So, we have tried two strategies:

1. Classify based on Responses only
2. Classify based on Responses and part of Context

For strategy 2, according to our test, we only can include two dialogues (we have tried the last two items in each context), if we selected 3 dialogues, it reports “CUDA out of memory” error. So, we just use the last two items in each context as part of the content for classification, then compare the f1 result (We are intended to compare the result of two strategies, and we will detail how to get the f1 value in later sections).



Model	Responses only	Responses + 2 dialogues in content
twitter-roberta-base	0.762214984	0.751169317
twitter-roberta-base-irony	0.762214984	0.740587932

From above, we can see that adding the content for classification doesn’t improve the prediction. So, we have decided not to include the “content” in each tweet for our classification task.

Please refer to code section 2.1 for how to load the data into Colab for test, and section 2.2 for how to read and parse the data for later use.

3.4 Tokenization

3.4.1 BERT Tokenizer

There are many different pre-trained BERT models available. Each model comes with its own tokenizer. We need to make sure we use the correct tokenizer as we experiment with different models.
We defined which pre-trained BERT model we will use in this step and if we need to change the model, we just need to simply update the model name. Please refer to code section 3.1 for more details.

3.4.2 Required Formatting

BERT requires tokens to fit certain format:

1. Add special tokens to the start and end of each sentence.
   - For classification tasks, we must prepend the special [CLS] token to the beginning of every sentence. This token has special significance. BERT consists of 12 Transformer layers. Each transformer takes in a list of token embeddings, and produces the same number of embeddings on the output.
   - At the end of every sentence, we need to append the special [SEP] token.
2. Pad & truncate all sentences to a single constant length.
3. Explicitly differentiate real tokens from padding tokens with the "attention mask".

BERT has two constraints:

1. All sentences must be padded or truncated to a single, fixed length.
   - For any sentences that are less than the fixed length, we need to PAD with the same special token.
   - For any sentences that are more than the fixed length, we need to truncate them to the fixed length, otherwise the system will report the errors.
2. The maximum sentence length is 512 tokens.

3.4.3 Tokenize Dataset

For saving the memory in the model training, we set the max_length of the input token based on the max length of all the input sentences.
We used encode_plus methods for token padding and attention masking, it automatically adds “special tokens” which are special IDs the model uses. We need to convert the lists into tensors in order to use the Pytorch properly.
Please refer to the code section 3.3 for the detailed implementation.

3.4.4 Training & Validation Split and Batch Size

Before training our data using BERT, data must be split into tokens, and then these tokens must be mapped to their index in the tokenizer vocabulary. The tokenization must be performed by the tokenizer included with BERT.

For the provided training data, we split them into training set and validation set, the proportion of the training set will impact the prediction accuracy of the model. We have tried the training:validation ratio as 9:1, 99:1 and 8:2. We observed when the training:validation ratio is 9:1 provides the best prediction result. Please refer to section 3.7.2 for the detail test result.

We used DataLoader to help us save memory, and we define the batch size for this step.

Please refer to the code section 3.4 for the detailed implementation of data split, the usage of dataloader and the setting of batch size.

3.5 Fine-tune BERT Models

First we load the defined pre-trained BERT model, examine the result, then we set the key values for training the model:

- Batch size (set when creating our DataLoaders)
- Learning rate
- Epochs

Then we train the model with the training loop and provide the training result statistic (accuracy and validation lost). We use the output of the fine-turn model to predict the tweets in the test.jsonl. After that we update the parameters and train the model again.
When we have a satisfied result for the pre-trained model under test, we change to other models with the same parameters.
Here is the a list that we have experimented:

- Bert Models:
  - Bert-base-uncased
  - bert-large-uncased
- Twitter-roberta models:
  - cardiffnlp/twitter-roberta-base
  - cardiffnlp/twitter-roberta-base-irony
  - cardiffnlp/twitter-roberta-base-offensive

Based on our test result, twitter-roberta models outperforms Bert Models on this task, we have decided not to use this model in early stage, so our testing is mainly focused on the twitter-roberta related models. Comparing the performance of three twitter-roberta models, cardiffnlp/twitter-roberta-base provides the best performance with the following settings:

- Batch size: 32/64 in training, and 32 in prediction
- Learning rate: 2e-5
- Epochs: 4/6

3.5.1 Download the Model

Please refer to the code section 4.1.

3.5.2 Optimizer & Learning Rate Scheduler

We use AdamW optimizer and define the epochs in the scheduler.

Impact analysis on the parameters:

- Batch size: The batch size is the number of words to be used for calculation and updating the weight once, it impacts the training time, memory usage and model's prediction accuracy. The larger the batch size is, the more memory it will consume. When the batch size is too small, the result may not converge, and when the batch size is too big, it will cause over fitting or out of memory. Based on our test, the batch_size = 32/64 provides the best performance.
- Learning rate: We have tried three learning rates: 2e-5, 1e-4, 1e-6. We noticed that when the learning rate is large, it cannot converge, and when the learning rate is small, it overfits. learning rate = 2e-5 provides the best prediction result.
- Epochs: Epochs value need to be set correctly, if the value is too small, the gradient descent will not converge, if the epochs value is bigger than required, we will overfit the model and decrease the accuracy of the prediction result. We use the accuracy and validation lost in each epoch's validation result. When we notice when the epochs number increases, the accuracy increases, we increase the epoch value to test again. If the accuracy decreases or there is not much change, we decrease the epochs to the point when accuracy is not increased. Based on our test, the epochs = 4/6 provides the best performance.

Please refer to the code section 4.2 for the implementation.
Please refer to section 3.7.3 for the detail test result.

3.5.3 Training Loop

We define and create the training loop based on the contribution of Stas Bekman.
The training loop has a training phase and a validation phase. It also detects over-fitting by using validation loss.
After defining the training loop, we start to fine-tune the model.
Please refer to the code section 4.3.

3.6 Predict on the test data

3.6.1 Data Preparation

Prepare the test data just as how we prepare the training data.
Please refer to the code section 5.1.

3.6.2 Predict on the test set

Generate the final prediction based on the score, and download the result as answer.txt.
Submit answer.txt to livelab for the result. When the f1 result is not good, we change the parameters or pre-trained model as mentioned in 3.5.
Please refer to the code section 5.2.

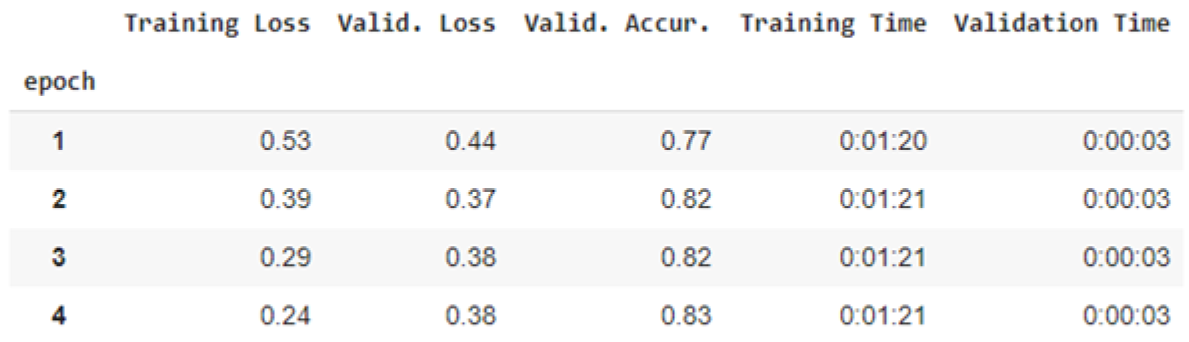
3.7 Results

3.7.1 Final Result

Chosen Model: cardiffnlp/twitter-roberta-base
Parameters setting:

- Batch size: 32 in training, and 32 in prediction
- Learning rate: 2e-5
- Epochs: 4

Model Training Result:



epoch	Training Loss	Valid. Loss	Valid. Accur.	Training Time	Validation Time
1	0.53	0.44	0.77	0:01:20	0:00:03
2	0.39	0.37	0.82	0:01:21	0:00:03
3	0.29	0.38	0.82	0:01:21	0:00:03
4	0.24	0.38	0.83	0:01:21	0:00:03

The Best Prediction Result with the above parameters:

- Precision: 0.7577319587628866
- Recall: 0.8166666666666667
- F1: 0.7860962566844919

3.7.2 Result Discussion

We observed that when using the same parameters to fine-tune the pre-trained model, the prediction result is not the same for each time. We have confirmed that even we set seed for the random sampling and using SequentialSampler in dataloader, we cannot make the prediction consistent.

Based on the information provided in Paperswithcode, the issue is caused by the Attention Dropout and Dropout:

- Attention Dropout: It is a type of dropout used in attention-based architectures, where elements are randomly dropped out of the softmax in the attention equation.
- Dropout: It is a regularization technique for neural networks that drops a unit (along with connections) at training time with a specified probability (a common value is). At test time, all units are present, but with weights scaled by (i.e. becomes). The idea is to prevent co-adaptation, where the neural network becomes too reliant on particular connections, as this could be symptomatic of overfitting. Intuitively, dropout can be thought of as creating an implicit ensemble of neural networks.

According to Huggingface BertConfig, the default dropout rate is set to 0.1:

- hidden_dropout_prob (float, optional, defaults to 0.1) – The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.
- attention_probs_dropout_prob (float, optional, defaults to 0.1) – The dropout ratio for the attention probabilities.

Considering the above factors and the test result, we observe the following parameters' settings ensures we can get the F1 > 0.75 which is well above the baseline:

- Batch size: 32/64 in training, and 32 in prediction
- Learning rate: 2e-5
- Epochs: 4/6

Here is the test result for these settings:

epoch	Training Split	Precision	Recall	F1	Comments
4	0.9-0.1	0.71468927	0.843333333	0.773700306	batch_size=64 Sentences=Response
4	0.9-0.1	0.69802867	0.865555556	0.77281746	batch_size=80 Sentences=Response
6	0.9-0.1	0.72380952	0.844444444	0.77948718	batch_size=64 Sentences=Response
8	0.9-0.1	0.73635427	0.794444444	0.764297167	batch_size=64 Sentences=Response
8	0.8-0.2	0.73236515	0.784444444	0.75751073	batch_size=64 Sentences=Response
4	0.8-0.2	0.70119157	0.85	0.768458061	batch_size=64 Sentences=Response
6	0.9-0.1	0.74476987	0.79111	0.767241379	batch_size=100 Sentences=Response
8	0.9-0.1	0.747288	0.76555	0.75631174	batch_size=100 Sentences=Response
5	0.9-0.1	0.72727272	0.80888	0.7659	batch_size=100 Sentences=Response
6	0.9-0.1	0.7391304	0.79333	0.76527	batch_size=80 Sentences=Response
6	0.9-0.1	0.74545454	0.77444	0.759673024	batch_size=64 Sentences=R learning rate=5e-5
6	0.9-0.1	0.73473	0.77555	0.75459	batch_size=64 Sentences=R learning rate=1e-5
6	0.9-0.1	0.7433264	0.80444	0.77267	learning rate=2e-5, batch_size=32,200 Sentences=Response
6	0.9-0.1	0.73313783	0.833333333	0.780031201	learning rate=2e-5, batch_size=64,32 Sentences=Response
4	0.9-0.1	0.74543611	0.816666667	0.77942736	learning rate=2e-5, batch_size=32,32 Sentences=Response
6	0.9-0.1	0.73615307	0.812222222	0.77231907	learning rate=2e-5, batch_size=32,32 Sentences=Response
4	0.9-0.1	0.75773196	0.816666667	0.786096257	learning rate=2e-5, batch_size=32,32 Sentences=Response

3.7.3 Testing Records

Below is the table that listed all the models we've tried with some parameters and statistics. The thirteenth line achieved the best result, which ranked the second in the leadership board.

epoch	Training Split	Precision	Recall	F1	Comments
		0.591836735	0.805555556	0.682352941	no fine-tuning
4	0.9-0.1	0.74522293	0.78	0.762214984	

4	0.99-0.01				predict all tweets as sarcasm. results not usable
4	0.95-0.05	0.722109534	0.791111111	0.755037116	
4	0.9-0.1	0.74522293	0.78	0.762214984	
4	0.9-0.1	0.691049086	0.797777778	0.740587932	Sentences=Response + Context[-2:]
4	0.9-0.1	0.648626817	0.892222222	0.751169317	Sentences=Response + Context[-2:]
4	0.9-0.1	N/A	N/A	N/A	Sentences=Response + Context[-3:],CUDA out of memory.
4	0.9-0.1	N/A	N/A	N/A	batch_size=64 Sentences=Response + Context[-2:], CUDA out of memory
4	0.9-0.1	0.714689266	0.843333333	0.773700306	batch_size=64 Sentences=Response
4	0.9-0.1	N/A	N/A	N/A	batch_size=100 Sentences=Response,CUDA out of memory.
4	0.9-0.1	0.698028674	0.865555556	0.77281746	batch_size=80 Sentences=Response
6	0.9-0.1	0.723809524	0.844444444	0.77948718	batch_size=64 Sentences=Response
8	0.9-0.1	0.736354274	0.794444444	0.764297167	batch_size=64 Sentences=Response
8	0.8-0.2	0.732365145	0.784444444	0.75751073	batch_size=64 Sentences=Response
4	0.8-0.2	0.701191567	0.85	0.768458061	batch_size=64 Sentences=Response
6	0.9-0.1	0.768564356	0.69	0.727166276	batch_size=64 Sentences=Response
6	0.9-0.1	0.732291667	0.78111	0.7559	batch_size=64 Sentences=Response
2	0.9-0.1	0.5	1	0.6667	batch_size=64 Sentences=Response
6	0.9-0.1	0.74476987	0.79111	0.767241379	batch_size=100 Sentences=Response
8	0.9-0.1	0.747288	0.76555	0.75631174	batch_size=100 Sentences=Response
5	0.9-0.1	0.72727272	0.80888	0.7659	batch_size=100 Sentences=Response
6	0.9-0.1	0.7391304	0.79333	0.76527	batch_size=80 Sentences=Response
6	0.9-0.1	0.74545454	0.77444	0.759673024	batch_size=64 Sentences=R learning rate=5e-5
6	0.9-0.1	0.73473	0.77555	0.75459	batch_size=64 Sentences=R learning rate=1e-5
6	0.9-0.1	0.5	1	0.6667	learning rate = 4e-4 batch_size=64 Sentences=Response
6	0.9-0.1	0.665354331	0.563333333	0.610108303	learning rate = 1e-6 batch_size=64 Sentences=Response
6	0.9-0.1	0.73193	0.7766	0.7536	learning rate=2e-5, batch_size=64 Sentences=Response
6	0.9-0.1	0.751054852	0.791111111	0.770562771	learning rate=2e-5, batch_size=64,200 Sentences=Response

6	0.9-0.1	0.7433264	0.80444	0.77267	learning rate=2e-5, batch_size=32,200 Sentences=Response
6	0.9-0.1	0.73313783	0.833333333	0.780031201	learning rate=2e-5, batch_size=64,32 Sentences=Response
4	0.9-0.1	0.745436106	0.816666667	0.77942736	learning rate=2e-5, batch_size=32,32 Sentences=Response
6	0.9-0.1	0.736153072	0.812222222	0.77231907	learning rate=2e-5, batch_size=32,32 Sentences=Response
4	**0.9-0.1**	**0.757732**	**0.81666667**	**0.7860963**	**learning rate=2e-5, batch_size=32,32 Sentences=Response**

4. Instructions for Using the Software

4.1 Setup and Installation

For this classification task, we used PyTorch as the Deep learning framework, Python as the Programming Language and pre-trained Twitter-specific BERT model, roBERTa-base model as the State-of-the-art neural network classifier.

Here are the steps to execute the classifier:

1. Download these files from https://github.com/lipingxie/CourseProject :
    a. Final BERT Twitter Sarcasm Classification.ipynb
    b. test.jsonl
    c. train.jsonl
2. Add Google Colab as an app to Google Drive.
3. Upload “Final BERT Twitter Sarcasm Classification.ipynb” into Google Colab and open the file in Colab. Our code includes all the required commands to setup the environment, Colab provides the default Python(version 3.6.9)environment.
4. In Colab, in the top menu, select “Runtime” -> “Change runtime type” -> In “Hardware accelerator”, select “GPU” -> click “SAVE”
5. You can run all the code in sequence via the top menu, select “Runtime” -> “Run all”. When the code in section 2.1 is being executed, you need to upload the file manually. The upload button will be enabled once this code is being executed. Upload test.jsonl and train.jsonl at the same time. Once the files are uploaded successfully, the remaining code will continue to be executed automatically.
6. Wait for the code to be finished, it may take a long time around 15-60 minutes which depends on Colab performance on that time period.
7. The answer.txt file will be downloaded automatically (If the download action is permitted in your machine.). Sometimes the answer.txt file cannot be automatically downloaded due to your local environment issue. You can click on the folder button on the left-side bar, the file will be displayed there and you can download it manually.
8. If you would like to verify the prediction result, you need to upload the file to livelab-ClassificationCompetition project with your own account with the similar process of MP2.4. As mentioned in section 3.7.2, due to the BERT default dropout, you may not

get the same result as our report or on the board, but we guarantee the result can pass the baseline.

9. You can check our best prediction result of our fine-tuned model in livelab -> ClassificationCompetition project -> Leaderboard -> Record uploaded by “Liping Xie”

4.2 Troubleshooting

If you would like to execute the code from the beginning again or load the model again, it may fail or you will have memory issues during the training stage. It seems it is caused by the Colab. The following steps can help to solve the problem:

1. Refresh the page
2. Click on the “Additional Connection Options” button (displayed as a small triangle icon that is next to “RAM” and “Disk” and located on top right corner.
3. Select “manage sessions”
4. Select the active section and click “TERMINATE”
5. Execute the code from the beginning again

Due to the dependency of the code, it may generate unexpected results if you re-execute part of the code. Please terminate the session and run from the beginning again.

5. Team Member Contribution

In this project, we all contributed a lot to each of the tasks, including writing the project proposal, researching cutting-edge pre-trained models, implementing the models on training and testing data, writing the progress report, finalizing the source code, and writing the documentation.

6. Reference

- Nguyen, Dat Quoc. BERTweet: A pre-trained language model for English Tweets. Retrieved from: https://www.aclweb.org/anthology/2020.emnlp-demos.2.pdf
- Jay Alammar, 2018. BERT: http://jalammar.github.io/illustrated-bert/
- Yinhan Liu, 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv preprint, arXiv:1907.11692
- Jacob Devlin, 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of NAACL, page 4186.
- Paperswithcode BERT: https://paperswithcode.com/method/bert
- Huggingface BertConfig: https://huggingface.co/transformers/model_doc/bert.html
- Model reference: https://huggingface.co/models
- Transformer coding reference: https://github.com/huggingface/transformers/blob/5bfcd0485ece086ebcbed2d008813037968a9e58/examples/run_glue.py#L109
- Model training reference: https://mccormickml.com/

- Pytorch coding reference: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py
- Stas Bekman: https://github.com/stas00?tab=repositories