**Project 2 Report: Covid-19 NER**

Chapter 1. Group Information

* Kaggle Team Name: Yellow Submarine
* Group Number: 14
* Names & Student IDs & Kaggle Display Name of Team Members:
  + LIU Donghua: 20731100 (Danhuang)
  + YUAN Yanzhe: 20728555 (bighead)
  + SHEN Dinghui: 20726478 (mengnan)
* Final Rank and Scores:
  + Score:0.93073, Rank: 5

Chapter 2. Algorithm Description

1. **Task Description**

**1.1 Named Entity Recognition**

Named Entity Recognition (NER) is an important subtask of Natural Language Processing (NLP) which aims to classify named entities in a text with the corresponding types. Entities refer to entities with specific meanings in the text, which mainly include names of persons, places, etc., as well as words such as time, quantity, currency, etc.

**1.2 Named Entity Recognition**

In this project, based on the corpus of academic literature about Covid-19, we need to train our model on the NER task. In the end, our model is supposed to predict the corresponding type of each token in a sentence.

1. **Algorithm Introduction**

In this project, we mainly use the pre-trained model BERT, which stands for Bidirectional Encoder Representations from Transformers. The pre-trained BERT can be fine-tuned through an additional output layer, which is suitable for the construction of state-of-the-art models for a wide range of tasks.

1. **Algorithm Description**
   1. **Data Preprocessing**

**3.1.1 NER Tags Preprocessing**

For this part, we mainly follow the guide of the ‘playground.ipynb’ file provided by the project tutorial, but with a little bit of change. Besides of the 65 kinds of entity tag we already know, we add two more special tags of ‘[CLS]’ and ‘[SEP]’, for we will add these two special tokens in the later processing.

**3.1.2. BERT Input Data Preprocessing**

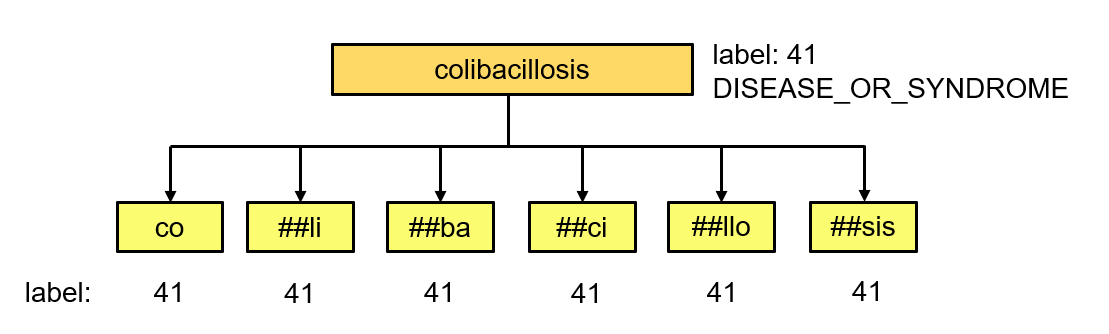
Before defining our classifier, we should customize our data so as to feed into BERT model. We use BERT Tokenizer to tokenize our data. But we encounter a serious problem after we go further using the tokenized data. The problem is that for tokens not in the vocabulary, BERT Tokenizer splits them into multiple tokens, thus the length of tokenized token will not match the length of original labels.

Original sentence: *Protection of calves against fatal enteric colibacillosis ......*

Tokenized tokens: *‘Protection’, ‘of’, ‘ca’, ‘##lves’, ‘against’, ‘fatal’, ‘enter’, ‘##ic’, ‘co’, ‘##li’, ‘##ba’, ‘##ci’, ‘##llo’, ‘##sis/ ......*

**Figure 1. An example of tokenized sentence**

So, we calculate the number of the sub-tokens split from the original token and set the label of each sub-token as the same label of the original token. After that, the length of the label dictionary will be the same as the length of tokenized sentence.



**Figure 2. An example of extending the label**

* 1. **BERT Fine-Tuning**
     1. **DataLoader**

After preprocessing, we use encode\_plus function to encode every sentence to the form of '[CLS] text [SEP]' and add '[PAD]' token to the same 'MAX\_LENGTH' that we defined. After that, we return the input\_ids and attention\_masks of the data for further processing.

Next, we put the data into PyTorch DataLoader, which creates an iterator for the data to convenient the training and testing process.

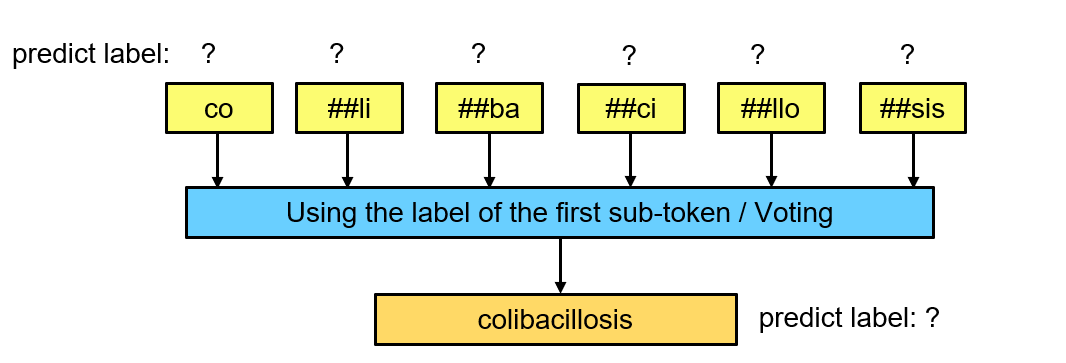
* + 1. **Bert Classifier**

We use the pre-defined model BertForTokenClassification in HuggingFace's Transformer. This model contains two parts. First, there is a Bert Model to learn the representations of text and then there is a linear layer on top of the hidden states output for token classification tasks like NER. In addition, there is a dropout layer in between the two layers. The forward function returns the classification loss, the logits, the hidden states, and the attention weights as the output so that we can choose the return variable for the training or testing process. Besides, we use Adam as the optimizer and cross entropy loss as the loss function. After defining and the model, we then resize the token embeddings according to the tokenizer since we have added additional tokens.

* 1. **Train, Evaluate and Test Model**

In the training and evaluating process, first, for every epoch, we train the model on the train dataset and then evaluate on the validate dataset. We also record the intermediate results like train loss, train accuracy, validation accuracy and validation loss in order to record the best model and avoid problems like overfitting during fine tuning. Then, we use the best model for prediction on the test dataset.

Similarly, we also need to handle the problem of length mismatch problem for the predicted label and the original sentences. For a given word, Bert Tokenizer may split it into many sub-tokens and our model will give many predicted labels, but we only need one tag for an original word. So, we propose two ways to determine the final label: the first is to directly use the predicted tag of the first sub-token, and the second is to use the tag with the most occurrences in all sub-tokens.



**Figure 3. Two different methods of predicting uncommon word for the test data**

During the prediction, we check that the length of labels is not the same as the length of the required output. We compare the number of labels and the number of tokens in each row line by line, and find the problem is that the length of the sentence after tokenized in the 528-th and 2485-th line of the test set exceeds 512. The reason why this happens is that these two lines are not English at all, but two lines of Russian. So, we directly predict all tags in these two lines to be O as the final solution.

Chapter 3 Experiment

1. **Experiment Details**

Hyperparameter Tuning is the process of choosing a set of optimal parameters for a learning algorithm to achieve a best performace. In this project, we tried the following parameter combination:

* BERT type: {'bert-base-cased', 'bert-base-uncased'，'distilbert-base-cased'}
* lr (learning rate): {2e-5, 3e-5,4e-5}
* max\_length: {128,192,256}
* batch\_size: {16, 32}
* epoch: {6, 8, 10, 12, 14}

All in all, the best model we have is with these hyperparameters:

{BERT type: 'bert-base-uncased', lr: 3e-5, max\_length: 256, batch\_size: 32, epoch:12}

As comparison experiment, we also tried to change the model structure in this direction:

Use BERT+CRF+Linear structure, and then fine-tuned the dropout ratio. As a result, the best result is: 0.91785 (public score on Kaggle)

As for the reason why BERT+CRF is not better than BERT+Linear, we think that is because BERT uses a transformer, which only uses positional embedding in the input layer, that is, the position information is weakened during the calculation process. However, in the sequence labeling task, location information is important, as well as direction information, so the performance may not be better. As for the improvement solution, we think it may be better if we use LSTM to learn the dependence on the observation sequence first, and then use the CRF to learn the relationship between the state sequence and get the answer. So in the future, it is worth a try to use BERT+BiLSTM+CRF for this task

1. **Difference Between Local Validation and Online Result**

Because of the handling of the length mismatch problem, we extend the label length when training. As a result, the local validation accuracy is slightly lower than the online result (about 1~2%). But this difference does not affect the model performance. The best model on the local test set is also our best model on the Kaggle leaderboard.

Chapter 4. Other Information

1. **How to Run:**

* Using Google Colab GPU mode
* Run the NER.ipynb file
* Get the prediction results in prediction.csv

1. **Third-party Libraries Dependency**

* pandas, numpy, pickle, Pytorch, Transformers, SKlearn