Machine Learning Project

Rating the Complexity of Literary Excerpts

Kavinesh, Ramjan Ali, and Debanshu Biswas

Indian Institute of Technology Jodhpur

November 2021

Abstract

In this project we are going to predict the reading ease of excerpt, a small extract from literature. In the dataset we have chosen excerpts from several time periods. And a wide range of complexity scores named "target" is given to all the entries to understand their complexity. And our aim to use the model for modern literature. We will use language representation model BERT for this task. Specifically we will use Hugging face package.

1 Introduction

In the long period the English language has changed significantly. So, some old literature may be difficult to understand for a person who speaks modern language. So, it is important to have a rating of complexity for different literature when it we want to add them in school text books. This project's outcomes have the potential of enhancing education immensely. So, in our dataset we have taken a vast range of excerpts from different time periods. We will train a machine learning model on it to make it applicable for any literature.

2 Dataset

We have taken the dataset from Kaggle.com and it is called common lit readability prize dataset. There we have two distinct CSV files one for training tran.csv and other one for testing the model test.csv. In the training dataset has six columns,
id - unique ID for excerpt
url_legal - URL of source - this is blank in the test set.
license - license of source material - this is blank in the test set.
excerpt - text to predict reading ease of
target - reading ease
standard_error - measure of spread of scores among multiple raters for each excerpt. Not included for test dataset

And for the test dataset we have four columns,
id - unique ID for excerpt
url_legal - URL of source - this is blank in the test set.
license - license of source material - this is blank in the test set.
excerpt - text to predict reading ease of
And we also observe that the test dataset has a higher proportion of modern languages as our

main aim is to rate modern literature, i.e, we want to apply this model to mainly modern literature. Two different attributes to rate excerpts in our data set is given as target value, standard error. Moreover we will add one more attribute root means square error(RMSE) for a better prediction.

3 BERT

BERT is an open source machine learning framework for natural language processing (NLP) pre-training developped by google. BERT was created and published in 2018 by Jacob Devlin and his colleagues from Google. BERT stands for Bidirectional Encoder Representations from Transformers. BERT is designed to help computers understand the meaning of ambiguous language in text by using surrounding text to establish context. The BERT framework was pre-trained using 2500 million words from Wikipedia and 800 million words from different books. They take any content from the datesets (which is passage from Wikipedia and books) and hide some random words in the content and train BERT model. this is called as masked language model.As side effect of mask language we get word embeddings for each word. Word embedding of a word is a vector (normally of dim = 768 * 1) that we get from the weights of neural network.

Applications

- used in gmail to predict the next word while writing a sentence.
- google search engines
- Polysemy and Coreference (words that sound or look the same but have different meanings) resolution
- Abstract summarization

Advantages or why we are using bert
Take this example,
"**Fair** gameplay"
"Fun **Fair**"
The word "Fair" has two meaning depending on the place we use it , BERT gives different word embedding for the word "Fair" based on its surroundings, which is not present in other model like word2vec. If two words have same meaning they have similar word embedding. For example Fun fair and carnival have similar word embedding even though they are different words
Relationships between words are captured. For example king - man + woman = queen in the sense of word embedding.

Drawbacks
BERT is a technology to generate "contextualized" word embeddings/vectors, which is its biggest advantage but also it's biggest disadvantage as it is very compute-intensive at inference time, meaning that if you want to use it in production at scale, it can become costly.

Whereas some word vectors (word2vec) are pre-calculated for each word/phrase and saved in a database for retrieval whenever one needs it, with BERT, there is a need to calculate/compute vectors every time.

4 Explaining the Code

We wrote the code for this model in Python Language. And we imported Hugging Face which is a startup in the Natural Language Processing (NLP) domain for simply processing the training data. We also imported basis packages like pandas, seaborn and numpy which are commonly used for data processing. Then from transformers we imported

1. BertTokenizer : Construct a BERT tokenizer based on WordPiece. This tokenizer inherits from PreTrainedTokenizer which contains most of the main methods.
2. BertForSequenceClassification : As we will be doing a regression task we are using this as thi is a Bert Model transformer with a sequence classification/regression head on top e.g. for GLUE tasks.
3. Trainer : Trainer is mainly a simple but complete training and eval loop for PyTorch, optimized for transformers.

4. : TrainingArguments is the subset of the arguments we use in our example scripts which relate to the training loop itself.

Lastly, we also used pytorch.

Then we imported the our dataset commonlitreadabilityprize in two distinct variables train_df and test_df separately for training and test dataset. Then we used BertTokenizer for tokenizing the data. And then we used BertForSequenceClassification for modeling.

Now, we gave one example to understand the tokenizer function. Here our main aim is to understand loss and logit functions. The code is as follows,

```
inputs = tokenizer("Hello, my dog is cute",
return_tensors="pt")
labels = torch.tensor([0]).unsqueeze(0)
inputs['labels'] = labels
outputs = model(**inputs)
loss = outputs.loss
logits = outputs.logits
```

We loaded the training dataand the test data in train_dataset and test_dataset.

Then we defined our own tokenizer function where we used the tokenizer function itself. The function is as follows,

We divided f_train_datasets into training and evaluate(eval) sets.

Then we also defined RMSE (Root Mean Squared Error) for better results. We will use defined compute_matrices function for RMSE error.

Then, we trained our data with the following epochs, logging steps and learning rate.

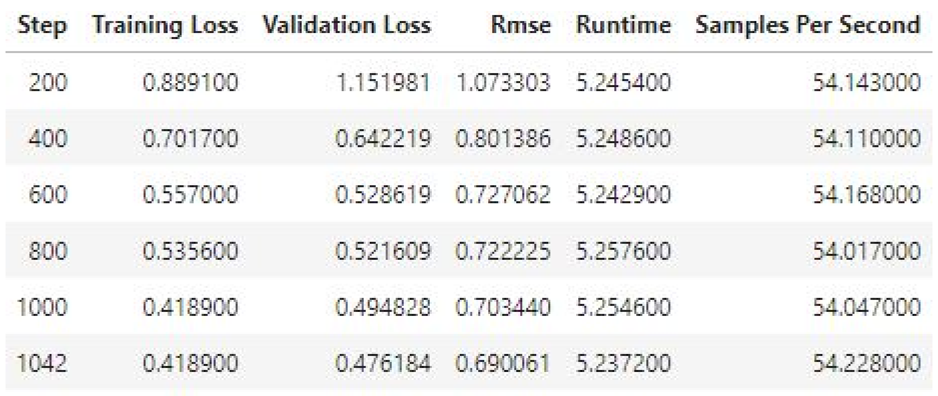
```
training_args = TrainingArguments(
    'training_args',
    num_train_epochs = 5,
    logging_steps = 200,
    learning_rate = 1e-4,
    per_device_train_batch_size = 8,
    per_device_eval_batch_size = 8,
    evaluation_strategy = 'steps'
)

trainer = Trainer(
    model = model,
    train_dataset = f_train_dataset,
    eval_dataset = f_eval_dataset,
    compute_metrics = compute_metrics,
    args = training_args
)
```

At last we evaluated the trainer to get losses.

Fortraining and test dataset ratio 90 : 10 and,

```
    num_train_epochs = 5,
    logging_steps = 200,
    learning_rate = 1e-4,
```

We get results,



Step	Training Loss	Validation Loss	Rmse	Runtime	Samples Per Second
200	0.889100	1.151981	1.073303	5.245400	54.143000
400	0.701700	0.642219	0.801386	5.248600	54.110000
600	0.557000	0.528619	0.727062	5.242900	54.168000
800	0.535600	0.521609	0.722225	5.257600	54.017000
1000	0.418900	0.494828	0.703440	5.254600	54.047000
1042	0.418900	0.476184	0.690061	5.237200	54.228000

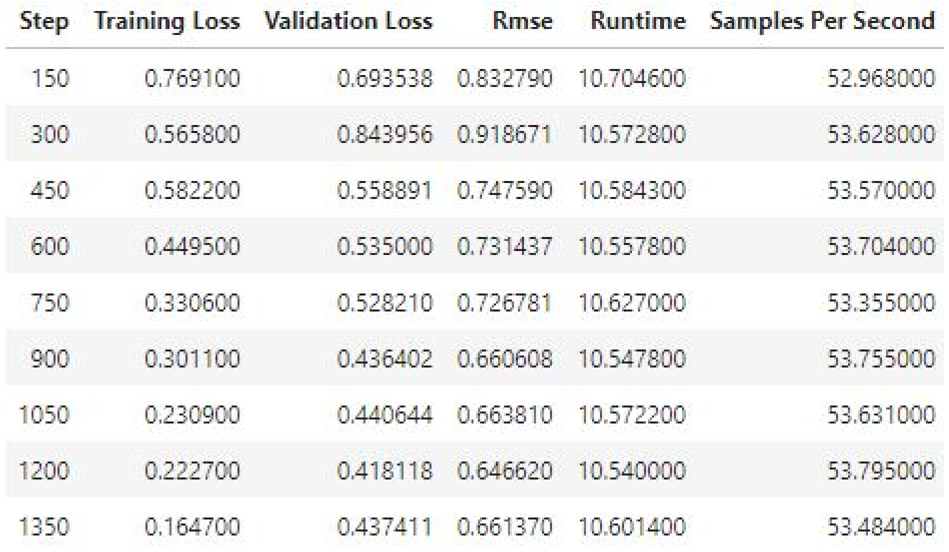
And evaluate gives,

```
 'eval_loss': 0.4761837124824524,
 'eval_RMSE': 0.6900606751441956,
 'eval_runtime': 5.2372,
 'eval_samples_per_second': 54.228
```

For training and test dataset ratio 80 : 20 and,

```
num_train_epochs = 6,
logging_steps = 150,
learning_rate = 1e-4,
```

We get results,



Step	Training Loss	Validation Loss	Rmse	Runtime	Samples Per Second
150	0.769100	0.693538	0.832790	10.704600	52.968000
300	0.565800	0.843956	0.918671	10.572800	53.628000
450	0.582200	0.558891	0.747590	10.584300	53.570000
600	0.449500	0.535000	0.731437	10.557800	53.704000
750	0.330600	0.528210	0.726781	10.627000	53.355000
900	0.301100	0.436402	0.660608	10.547800	53.755000
1050	0.230900	0.440644	0.663810	10.572200	53.631000
1200	0.222700	0.418118	0.646620	10.540000	53.795000
1350	0.164700	0.437411	0.661370	10.601400	53.484000

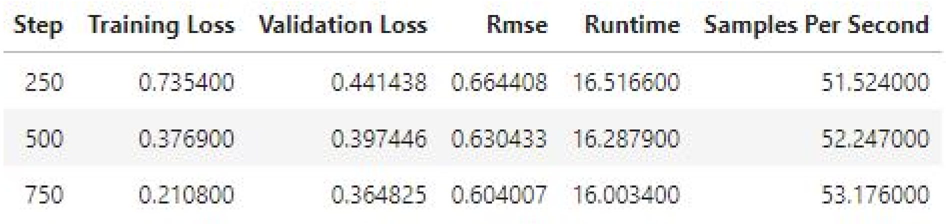
And evaluate gives,

```
'eval_loss': 0.40583258867263794,
 'eval_RMSE': 0.6370499730110168,
 'eval_runtime': 10.6922,
 'eval_samples_per_second': 53.029
```

For training and test dataset ratio 70 : 30 and,

```
num_train_epochs = 4,
logging_steps = 250,
learning_rate = 1e-4,
```

We get results,



Step	Training Loss	Validation Loss	Rmse	Runtime	Samples Per Second
250	0.735400	0.441438	0.664408	16.516600	51.524000
500	0.376900	0.397446	0.630433	16.287900	52.247000
750	0.210800	0.364825	0.604007	16.003400	53.176000

And evaluate gives,

```
'eval_loss': 0.3794867694377899,
'eval_RMSE': 0.6160249710083008,
'eval_runtime': 16.2861,
'eval_samples_per_second': 52.253,
'epoch': 4.0,
'eval_mem_cpu_alloc_delta': 4096,
'eval_mem_gpu_alloc_delta': 0,
'eval_mem_cpu_peaked_delta': 0,
'eval_mem_gpu_peaked_delta': 289534976
```

Conclusions

We observe that as number of steps increases Training Loss, Validation Loss and RMSE decreases. Also, our defined RMSE helped to the model to perform better.

References

http://jalammar.github.io/illustrated-bert/
https://www.analyticsvidhya.com/blog/2021/09/an-explanatory-guide-to-bert-tokenizer/
https://en.wikipedia.org/wiki/BERT_(language_model)
https://searchenterpriseai.techtarget.com/definition/BERT-language-model
https://www.youtube.com/watch?v=7kLi8u2dJz0
https://towardsdatascience.com/hugging-face-a-step-towards-democratizing-nlp-2c79f258c951