CS 378: Final Project Report

Pranav Eswaran (pve84) Viren Velacheri (vv6898)
Pranav and Viren

Due Friday May 15th Midnight CST

$\mathcal{V}$	A vocabulary; the set of token types
x	A token
$\bar{x}$	A sentence; a sentence of length n is sequence of tokens $\bar{x} = \langle x_1, \ldots, x_n \rangle$
x'	A retokenized token produced from a pretrained HuggingFace tokenizer
$\bar{x'}$	A retokenized sentence produced from a pretrained HuggingFace tokenizer
$\bar{d}$	A document; a document with n sentences is $\bar{d} = \langle \bar{x}_1, \ldots, \bar{x}_n \rangle$, where the tokens in sentence $\bar{x}_i$ of length m are $\bar{x}_i = \langle x_{i,1}, \ldots, x_{i,m} \rangle$. If sentences are not considered (e.g., for bag-of-word modeling), one can denote a document as a sequence of n tokens $\bar{d} = \langle x_1, \ldots, x_n \rangle$
y	POS tag, or a classification label
$\bar{y}$	POS tag sequence; a sequence of length n is $\bar{y} = \rangle y_1, \ldots, y_n$
y'	POS tag for a given retokenized token
$\bar{y'}$	POS tag sequence on a retokenized input sequence
$\mathcal{Y}$	A set of POS tags (e.g., to represent all possible POS tags), or the set of all possible labels for classification
L	A log-likelihood function or a loss function

1 Introduction (6pt)

Our task was to come up with a named-entity recognition model for a corpus of text from Twitter. The idea is that for each tweet, the sub-spans of the words that represent the entities should be identified. Some of the latest models today include Stanford NER and Spacy. The training dataset was comprised of 2394 pretokenized tweets with each token tagged as either **B, I,** or **O**. **B** means that this token is the start of an entity, **I** means that this token is part of an entity mention (but not the first token in the mention), and **O** means that this token is not part of an entity. Our development set had 959 tweets in the same format as the training set. We also had an unlabeled test set of 2377 tweets.

Our approach to this was to give each tweet from the corpus to a model comprised of a state-of-the-art BERT language model transformer and a linear layer which takes in the final hidden state of the transformer and outputs logits which we can then softmax over to get probability distributions for each label for each token.

We found that using all of the training data from our corpus produced the best results, and that using a dropout layer of 0.2 on the final hidden state of the `bert-large-cased` transformer before passing it to the linear layer produced the best results: F1-scores of 0.7028 and 0.6477 on the development and test sets respectively.

2 Task (5pt)

The model's task is given an input sentence $\bar{x}$ comprised of tokens $(x_1, \ldots, x_n)$, output $\bar{y}$ comprised of **BIO** labels $(y_1, \ldots, y_n)$.

3 Data (4pt)

Data Statistics			
Dataset	Num_ Tweets	Avg_Toks	Vocab_ Size
Training	2394	19.411	10586
Dev	959	13.931	4991
Test	2377	14.032	10548

The shallow statistics such as number of tweets, average number of tokens per tweet, and vocabulary size are displayed for each Dataset (Training, Development, and Training) in the above table. As far as pre-processing goes, we basically create encodings for both the tags and tokens.

For the tags, we manually specified the mapping where tag **O** goes with 0, **I** goes with 1, and **B** goes with 2. For encoding the tokens, we used our pre-trained DistilBert tokenizer where we deal with already-split tokens instead of full sentences. The sequences are padded as well. The main reason we did pre-processing was that we noticed Twitter Handles are overtokenized by our tokenizer. For instance, the Twitter handle **@huggingface** would be split into tokens **['@', 'hugging', 'face']** and this is a problem as we will have a mismatch between our tokens and labels. To fix this issue, we basically wrote our function `encode_tags` that allows us to only train on the first subtoken of a split token. The rest of the tokens are ignored as their labels are set to -100. Our preprocessing code was taken directly from here: **Hugging Face Preprocessing Code Reference**

4 Model (25pt)

At a high level, our model first feeds an input sentence into a BERT language model transformer, and then feeds the final output hidden state of this into a linear layer with input dimension equal to the size of the final hidden state output and output dimension 3 (so we can have a logit for each possible label (**BIO**).

More specifically, given an input sentence/tweet $\bar{x}$, we first retokenize it using a pretrained `DistilBertTokenizerFast` from HuggingFace to get $\bar{x'}$. We then feed this into a pretrained BERT transformer stack, also from HuggingFace. Using pretrained transformer stacks is known to produce state-of-the-art results, so we opted to use this as the first part of our model.

This transformer stack outputs the final hidden layer for each token in the input, and this output is fed into a dropout layer. The dropout layer was used to prevent overfitting to the training data by dropping tokens with some probability from the transformer output before passing it to the rest of the model. This is also a commonly used practice.

The dropped-out transformer output is then used as the input to a fully connected linear layer with input dimension equal to the size of the final transformer hidden state output and output dimension 3. We used a linear layer because it's a simple way to produce logits (or scores) for each possible label for each token. Running these logits through a log softmax effectively normalizes these scores, and this is the final output of our model (found in the `forward()` function of each model architecture).

For inference, given an tokenized tweet $\bar{x}$, we first retokenize in the same way as we mentioned earlier. Then we run a model on $\bar{x'}$ to get the softmaxed logits for each token x'. We then take the `argmax` over the logits for each token x' to get the final predicted **BIO** labels $\bar{y'}$ for each token x'. To project $\bar{y'}$ onto $\bar{x}$ to get $\bar{y}$, we rely on the fact that the `DistilBertTokenizerFast` only splits our input tokens $\bar{x}$ further. That is to say, for each retokenized token x', there exists an original token x such that $x' \in x$. Thus, we can rebuild the the original tokens $\bar{x}$ from $\bar{x'}$. From here, we assign each token x the label y' of the first x' in x. For example, if the token "Cedar" is retokenized into "C" and "edar", we assign the label that was given to "C" to "Cedar". After this process, we get our true $\bar{y}$ **BIO** labels.

We didn't do anything in our model architecture to handle unknown words or tokens. The retokenization process converts unknown words and tokens to **[UNK]** with a special index in its internal vocabulary. We took this approach since there will be many unknown words in our dataset (since it was taken from Twitter), and since the majority of these unknown words are emojis (which don't really carry any semantic meaning– only mainly tonal information – and almost certainly are not part of a named entity), we thought it was appropriate to not account for unknown words in either learning or testing. With this approach we still achieved and surpassed our goal model performance, so it's safe to say that this approach was the right one to take.

5 Learning (10pt)

We used standard batched gradient descent with a learning objective of minimizing L where L is the cross entropy loss between the gold labels and the predicted labels from inference. Specifically, we used the `AdamW` optimizer from HuggingFace to train our models.

6 Implementation Details (3pt)

Our implementation includes hyperparameters for the dropout probability of our dropout layer, the batch size used in training, and the learning rate used by the `AdamW` optimizer. We used the standard GPU training techniques like using CUDA with PyTorch, but other than that, no special optimizations were necessary for reasonable training times.

7 Experimental Setup (4pt)

It is pretty self explanatory, but basically we use all the training data to train each of our models and during this time, model checkpoints are generated at each epoch. From there, we run each checkpoint on the development data to get the performance results. Lastly, we just generate the development performance graph and find the epoch that generated the best F1 score as well. The baseline model we compare our BertModel against is the BertModelForTokenClassification, an off the shelf pretrained Named Entity Recognition Model. We compared it against DistilBertModel in the experiments we ran. The primary evaluation metric we used were F1 scores, but other metrics (precision and recall) were generated.

8 Results

Test Results (3pt)

Test Results			
Model	F1 Score	Precision	Recall
Bert (bert-based-cased)	0.6477	0.6922	0.6085
Pretrained	0.6517	0.6525	0.6508
Bert (bert-large-cased)	0.6477	0.6922	0.6085

The test results are displayed above. For the bert models based on large-cased and base-cased pretrained data, the hyperparameters used were a dropout of 0.2, batch-size of 32, and learning rate of 5E-5. Other hyperparameters such as hidden layer size are set in the model architecture itself. Since the BertModelForTokenClassification is a

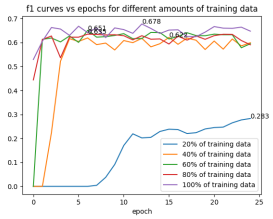
fully pretrained model, hyperparameters didn't need to be specified.

Development Results (7pt)

Best Model Results on Development Data			
Model	F1 Score	Precision	Recall
Bert (bert-large-cased)	0.703	0.699	0.706

Above are the results for our best model that was run with Bert with large-cased or more pretrained data, dropout rate of 0.2, learning rate of 5E-5, batch size of 32, and all other hyperparameters like hidden layer size were kept as is since part of model architecture. We arrived at this due to the various experiments we ran. The first was doing a comparison between DistilBert and Bert with no hyperparameter tuning done.With no dropout rate, batch size of 32, and learning rate of 5E-5, the results can be seen in graphs (see Appendix A) We then proceeded to run dropout rate experiment on just Bert Model. We ran with dropout rates of 0.1, 0.2, 0.3, and 0.5. The results can be seen in graphs (see Appendix B). As is evident, dropout of 0.2 got the best result and hence why we chose it. From there we experimented with learning rate and batch_size. However, the results were not better as expected. It seemed like our default learning rate and batch size was already the baseline and making them smaller couldn't make a difference.For ablation study, the first thing we did was changed the pretrained data that we were finetuning from **bert-base-cased** to **bert-large-cased** which was 4 times the amount of data. The result can be seen in graph (see Appendix C). From there, we removed different parts of the model post processing that we were doing in run_model file to see the affects. This can be seen in table (see Appendix D). It showed us how important our accounting for examples with 's was, but also showed suprisingly how little the effect of all our other checks like BIO Consistency and UNK Checking had on results for best Bert Model based-cased.

Learning Curves (4pt)



It is no surprise that using all the data leads to the best results. However, it appears that if time is an important factor, it would be worthwhile to not run on all the data, but rather just a subportion (somewhere between 40 and 60%) to get slightly lower performance at much faster speed

Efficiency Analysis (4pt) The hardware used to estimate these times is the **nvidia tesla t4 gpu** on Google Colab. For large cased Bert model, costs of inference per example are 0.04 seconds and 0.02 seconds for test and development data respectively. For base-cased one, it is 0.017 and 0.011 seconds. Based on the average number of tokens per example being 13.931, this means that large-cased Bert model must process about 348 and 174 tokens per second for test and development data respectively, with the base-cased Bert model must process about 819 and 1266 examples for test and development data. See Appendix E

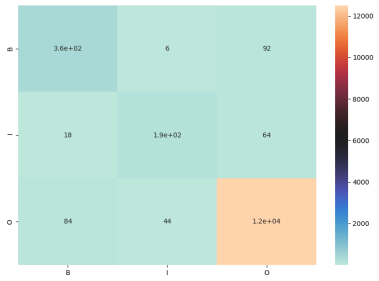
9 Analysis

Error Analysis (7pt)

Error Analysis based on Development Data		
Error Class	Actual	Predicted
UNK (ram)	B	O
Emojis	O	Error without our decoding of Emojis Code

It appears for the UNK cases that I think our problem just assumes that they are O's for the most part, especially if they are not cased. However, as shown in above table with ram example, even though not capitalized, it should be tagged as a B. For emjois, since they are not ASCi based, we at first got errors when trying to process them. Due to this, we just treated them link UNKs and assumed that they are tagged as O's which they normally are for our luck. It seems like our model gets it wrong for things that unknown and not cased as it commonly identifies them as O instead of B or even I.

Confusion Matrix (5pt)



It is clear from above confusion matrix that the most common confusions were those that revolved around **B** and **O** tags where **B** tags were predicted incorrectly as **O** tags and vice versa. The next most common confusion were those that involved the **O** and **I** tags.

10 Conclusion (3pt)

Overall, based on our results, a small dropout rate did help improve the results of our Bert model along with using more pretrained data. To achieve better results from this point forward, it is more model post processing and other changes to model architecture.

11 References

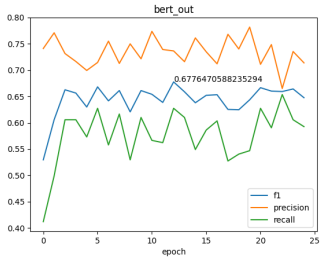
BERT: Pre-Training of Deep Bidirectional Transformers for ... www.aclweb.org/anthology/N19-1423.pdf.

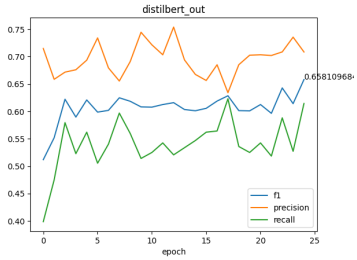
https://huggingface.co/transformers/custom_datasets.html#token-classification-with-w-nut-emerging-entities

https://huggingface.co/transformers/custom_datasets.html#token-classification-with-w-nut-emerging-entities

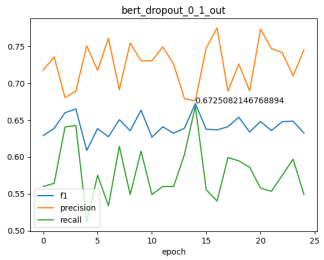
12 Appendices

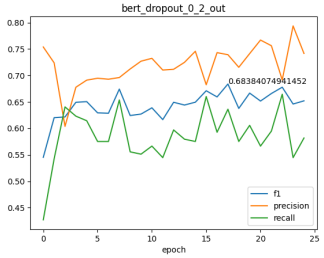
Appendix A

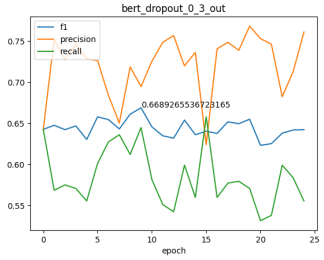


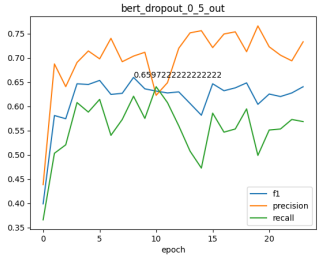


Appendix B

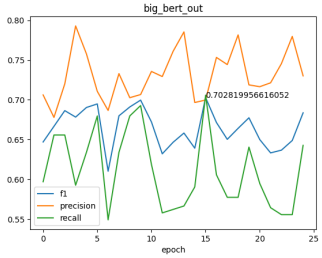








Appendix C



Appendix D

Ablation Study on Post Processing			
Change	F1 Score	Precision	Recall
None	0.683	0.73	0.636
No BIO Consistency Check	0.683	0.737	0.636
No Hashtags Check	0.053	0.433	0.028
No UNK Code Check	0.683	0.743	0.631

Appendix E

```
!python3 run_model.py --ex-path data/dev/dev.nolabels.txt --load-path ber

2021-05-15 21:11:36.636877: I tensorflow/stream_executor/platform/default
959 examples read in
Downloading: 100% 213k/213k [00:00<00:00, 22.7MB/s]
Downloading: 100% 436k/436k [00:00<00:00, 18.4MB/s]
Downloading: 100% 29.0/29.0 [00:00<00:00, 49.6kB/s]
elapsed time: 11.325494527816772
time per ex: 0.011809691895533651
```

```
!python3 run_model.py --ex-path data/dev/dev.nolabels.txt --load-path big_bert/bert_epoch_16.pt -

2021-05-15 21:25:45.574443: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Succes
959 examples read in
elapsed time: 21.506041049957275
time per ex: 0.02242548597492938
```

```
!python3 run_model.py --ex-path data/test/test.nolabels.txt --load-path big_bert/bert_epoch_16.pt --cuda

2021-05-15 18:19:59.076527: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully
2377 examples read in
elapsed time: 103.71044063568115
time per ex: 0.04363081221526342
```

```
!python3 run_model.py --ex-path data/test/test.nolabels.txt --load-path bert_dropout_0_2/bert_epoch_18.p

2021-05-15 18:21:53.882582: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully
2377 examples read in
elapsed time: 41.813562870025635
time per ex: 0.017590897294920336
```