CS 6320.501 final

Humza Salman

TOTAL POINTS

97 / 100

QUESTION 1

1 Introduction of task and data 15 / 15

✓ **- 0 pts** Correct

- 0 pts Define the task. – Specify the inputs and outputs. eg. with CoNLL-2003 dataset input Sequence of words output corresponding NER tags. And the with F1 score XXX

- 0 pts Describe the **data** you use, including how **many examples** are in the training, development, and test sets. Please also provide **shallow statistics** of your data, with at least the **vocabulary size, document length** (when relevant). Describe how you **pre-process** the data, if you conduct additional preprocessing on top of our preprocessing.

- 0.5 pts no **inputs and outputs**. eg. with CoNLL-2003 dataset input Sequence of words output corresponding NER tags. And the with F1 score XXX

- 0.5 pts no mentioning the **data** you use. eg. CoNLL-2003

- 1 pts No descriptions of **task** and **data** (e.g. how many examples are in the training, development, and test sets.)

- 1 pts No **shallow statistics** of your data, with at least the **vocabulary size, document length** (when relevant).

- 1 pts For student having `additional preprocessing`: No description of how you **pre-process** the data(if you conduct additional preprocessing on top of our preprocessing)

QUESTION 2

2 modeling and learning 20 / 20

✓ **- 0 pts** Correct

- 0 pts (15pts) Describe your model. If you use a **neural network, you should define the architecture**. There's no need to write equations of common units (e.g., LSTM), so you can just use **functions** to refer to using such **units**. If you use linear models with **features**, provide **details**. Briefly describe what **learning algorithm **you are using. Specify your **learning objective**; Briefly Describe your **inference** procedure. This it the process you use to make **predictions** during testing.

- 3 pts For student using `neural network`: No description of your model. For instance, using **functions** to refer to using such **units**(e.g., LSTM).

- 3 pts For student using `linear models`: No description of your model. For instance, If you use linear models with **features**, provide **details**.

- 2 pts No description about what **learning algorithm **you are using.

Supervised learning? Or what? (other possible algo: semi-supervised? boot strapping?)
Do you do multiclass classification? Or binary?

- 2 pts Not mentioned your **learning objective** what is your objective function? What is your loss? You may use cross entropy?(How do you calculate the loss between the golden label and predict result)

- 2 pts No description for your **inference** procedure. _(This it the process you use to make **predictions** during testing.)_

the inference procedure can be performed by feeding each sentence in the test set into the trained network and obtaining the eg. Softmax probability for each token(Based on how you design). (And I think the reason to mentioned how to inference is because some time training way differ from predicting. Some architecture may be used in training but not in testing, that's why it is important to write down how you design your inference stage)

- **0 pts** (5pts) Provide a description of how you handle **unknown** words as expressed in your model design. If you don't do anything in your model to handle unknown words, you should justify why this is the right approach. For example, if this is something that happens in learning only (or handled by the library), justify this **choice** and provide **details**.
- **3 pts** No descriptions for **unknown word** handling

QUESTION 3

3 implementations 15 / 15

- **0 pts** Correct
- ✓ - **0 pts** **Explain how you implemented the model (e.g. which **algorithms/data structures** you used). What existing **packages** are used. When describing your implementation details (e.g. **decoding**), we suggest that you include corresponding but only **important code pieces **(e.g. `classes`, `data structures`, `algorithms`, maybe in the form of **screenshot**).**
- **5 pts** no proper code files in GitHub repo
- **5 pts** no link to GitHub repo

QUESTION 4

experiments and results 45 pts

4.1 dev results 12 / 15

- **0 pts** Correct
- ✓ - **0 pts** **Briefly explain here how you **evaluated** the models and **metrics** used. Put the results into clear **tables** or **diagrams** and include your **observations** and **analysis**.**
- **3 pts** no explanations of metrics
- ✓ - **3 pts** **no precision and recall**
- **3 pts** no tables
- **2 pts** table/diagrams not clearly visible
- **3 pts** no observations (xx model achieves ..% F1)

4.2 describing test results on leaderboard 5 / 5

- **0 pts** Correct
- ✓ - **0 pts** **Briefly describe your test results from the **leaderboard**. This must match the number on the leaderboard. Put a **screen shot** from the leaderboard or put your "`team name (your name)`" here.**
- **3 pts** no kaggle name or screenshot

4.3 test performance 15 / 15

+ **0 pts** Correct
+ **15 pts** 15 if F1 score ≥ 82,

14, elif F1≥ 80,
13, elif F1≥ 75,
12, elif F1≥ 70,
11, elif F1≥ 65,
10, elif F1≥ 59,
8, elif F1≥ 55,
6, elif F1≥ 45,
4, elif F1≥ 35,
2 elif F1≥ 25,
0, otherwise.

+ 15 Point adjustment

4.4 error analysis 10 / 10

- **0 pts** Correct
- ✓ - **0 pts** ****Qualitative** analysis of **selected failure examples**. Show and discuss error examples from your **development** set. Identify certain classes of errors and use the examples to illustrate them. Please analyze with respect to **unknown** words as well.**
- **6 pts** no error cases (showing both sentence plus its predicted tags)
- **3 pts** tag level analysis but not instance (sentence/doc) level
- **5 pts** no unk analysis
- **3 pts** incomplete unk analysis
- **3 pts** no general/common identifications (context,

unk, features, etc.) of errors (only ad hoc analysis), supported by possibility multiple examples of the same errors

+ **3 pts** Speed Analysis (bonus 3pt) (final maximum 100) Speed analysis and computation needs are a major issue of the task. Makes sure to report the costs of inference per example.

Speed analysis and **computation** needs are a major issue of the task. Makes sure to report the costs of **inference** `per example`.

QUESTION 5

5 conclusion **5 / 5**

- **0 pts** Correct

✓ **- 0 pts Brief conclusion summarizing findings (from both **numerical** results and **qualitative** analysis).**
Your brief **feedback for the final (2-3 sentences would suffice).**

QUESTION 6

6 late penalty and invited talk **0 / 0**

✓ **- 0 pts not attending invited talk (no bonus points)**

+ **1 pts** bonus points: attending invited talk `(final maximum 100)`

+ **2 pts** bonus points: asking questions (you need to specify the question and the answer in final report) `(final maximum 100)`

CS6320: Final Report

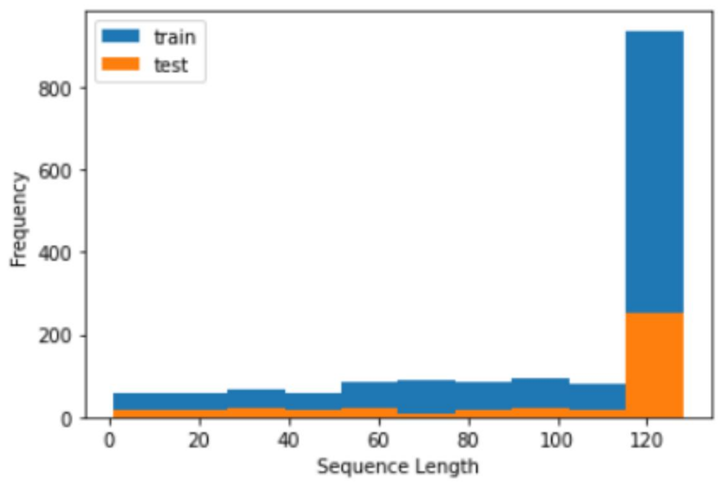
https://github.com/cs6320-501-fall22/final-Humza-S
Humza Salman
MHS180007

Introduction and Data

The project aims to implement a model capable of Named Entity Recognition. NER is an information extraction technique for the purposes of identifying and categorizing key information about entities within textual data. The model implemented in this report will be a fine-tuned pre-trained BERT model for the purposes of identifying five distinct NER tags: ORG (Organization), PER (Person), LOC(Location), MISC (Miscellaneous), and O (Not Named Entity). For inputs, we will be using a modified version of the CoNLL-2003 provided to us. Using these inputs, we will output the appropriate tagging sequence for the test data.

In our analysis of the data we learned the training and testing data consisted of sequences of lengths greater than 512. In our training data, we originally had 756 sequences, of which 49 sequences had a length over of 512 and 477 had a length over 128. This was corrected and we were left with 1614 sequences of length 128 or less. This is important to note as the pre-trained BERT model we fine-tuned has a maximum sequence length of 512. To ensure data integrity was kept, we split each data sequence constrained to a maximum length, *limit*, of 128. We chose the *limit=128* so that we could safely tokenize our inputs. The following figures describe the data distribution:

	Train	**Test**
Over 512	49	16
Over 128	477	125
Total Count	756	189
After Split	1614	412



Figures depicting count of train and test data, as well as sequence length frequency

Model and Learning

Model

As mentioned, we took the approach of fine-tuning the pre-trained language model BERT to our NER task. We utilized "bert-base-cased" as it was pre-trained to have token embeddings on the English language with a vocabulary of size 28996. We load the pre-trained BertTokenizer "bert-base-cased" tokenizer and create our tokenized datasets using our function *address_subwords* which is meant to address the sub-tokens generated by tokenization. We then create our BertForTokenClassification model since we will be doing NER classification tasks. We then train our model with varying epochs and learning rates

Architecture

The BertForTokenClassification model has a pre-trained embedding layer. It then has an encoding layer, followed by a Dropout layer with p=0.1, and finally a classifier Linear Layer with five output labels.

Learning Algorithm/Learning Objective

The model uses an AdamW optimizer to facilitate its training process with β_1 = 0.9, β_2 = 0.9, and a weight decay of 0.00001. We vary the epochs and learning rates to facilitate finding our best performing model.

1 Introduction of task and data 15 / 15

✓ **- 0 pts** Correct

- 0 pts Define the task. – Specify the inputs and outputs. eg. with CoNLL-2003 dataset input Sequence of words output corresponding NER tags. And the with F1 score XXX

- 0 pts Describe the **data** you use, including how **many examples** are in the training, development, and test sets. Please also provide **shallow statistics** of your data, with at least the **vocabulary size, document length** (when relevant). Describe how you **pre-process** the data, if you conduct additional preprocessing on top of our preprocessing.

- 0.5 pts no **inputs and outputs**. eg. with CoNLL-2003 dataset input Sequence of words output corresponding NER tags. And the with F1 score XXX

- 0.5 pts no mentioning the **data** you use. eg. CoNLL-2003

- 1 pts No descriptions of **task** and **data** (e.g. how many examples are in the training, development, and test sets.)

- 1 pts No **shallow statistics** of your data, with at least the **vocabulary size, document length** (when relevant).

- 1 pts For student having `additional preprocessing`: No description of how you **pre-process** the data(if you conduct additional preprocessing on top of our preprocessing)

CS6320: Final Report

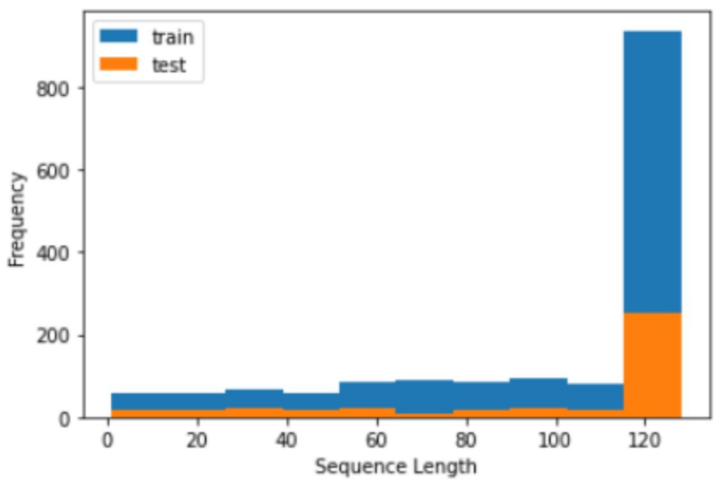
https://github.com/cs6320-501-fall22/final-Humza-S
Humza Salman
MHS180007

Introduction and Data

The project aims to implement a model capable of Named Entity Recognition. NER is an information extraction technique for the purposes of identifying and categorizing key information about entities within textual data. The model implemented in this report will be a fine-tuned pre-trained BERT model for the purposes of identifying five distinct NER tags: ORG (Organization), PER (Person), LOC(Location), MISC (Miscellaneous), and O (Not Named Entity). For inputs, we will be using a modified version of the CoNLL-2003 provided to us. Using these inputs, we will output the appropriate tagging sequence for the test data.

In our analysis of the data we learned the training and testing data consisted of sequences of lengths greater than 512. In our training data, we originally had 756 sequences, of which 49 sequences had a length over of 512 and 477 had a length over 128. This was corrected and we were left with 1614 sequences of length 128 or less. This is important to note as the pre-trained BERT model we fine-tuned has a maximum sequence length of 512. To ensure data integrity was kept, we split each data sequence constrained to a maximum length, *limit*, of 128. We chose the *limit=128* so that we could safely tokenize our inputs. The following figures describe the data distribution:

	Train	Test
Over 512	49	16
Over 128	477	125
Total Count	756	189
After Split	1614	412



Figures depicting count of train and test data, as well as sequence length frequency

Model and Learning

<u>Model</u>

As mentioned, we took the approach of fine-tuning the pre-trained language model BERT to our NER task. We utilized “bert-base-cased” as it was pre-trained to have token embeddings on the English language with a vocabulary of size 28996. We load the pre-trained BertTokenizer “bert-base-cased” tokenizer and create our tokenized datasets using our function *address_subwords* which is meant to address the sub-tokens generated by tokenization. We then create our BertForTokenClassification model since we will be doing NER classification tasks. We then train our model with varying epochs and learning rates

<u>Architecture</u>

The BertForTokenClassification model has a pre-trained embedding layer. It then has an encoding layer, followed by a Dropout layer with p=0.1, and finally a classifier Linear Layer with five output labels.

<u>Learning Algorithm/Learning Objective</u>

The model uses an AdamW optimizer to facilitate its training process with β_1 = 0.9, β_2 = 0.9, and a weight decay of 0.00001. We vary the epochs and learning rates to facilitate finding our best performing model.

Validation Procedure

For our validation procedure we utilize our tokenized validation dataset from earlier. We use a *DataLoader* object to process our validation data in batches. We retrieve the *attention_mask*, *input_ids*, and *label_ids* from the batch. We utilize these as inputs passed into our model to generate an output. We then utilize the loss and logits obtained from this output to create our prediction and true-labeled data. Once we have obtained our prediction and true-labeled data we utilize the *format_output_labels* function provided to retrieve our *y_pred_dict* and *y_true_dict* values. An important note to consider is empty dictionaries caused by our initial data-preprocessing. We omit these dictionaries when computing the F1-score utilizing the *mean_f1* function provided and summing over all validation samples and then dividing by their length to obtain the final F1-score.

Inference Procedure

For our inference procedure we utilize our tokenized test dataset from earlier. We retrieve the text for each sample and then tokenize the text to retrieve the *attention_mask*, *input_ids*, and *label_ids*. We utilize these as inputs passed into our model to generate an output. We then utilize the logits obtained from this output to create our prediction data. An important note to consider is that *label_ids* is generated from the text from the sample and is used to determine which logits to keep. We use a function similar to *address_subwords*, called *address_subwords_single* (the only difference being it is for a single piece of text), to generate our *label_ids*. Once we have our prediction data we do some additional preprocessing on our predictions and utilize the *create_submission* function provided to create our submissions.

Unknown Word Handling

Since BERT was trained on a large corpus, we find that most words are not unknown and can be handled by the extensive vocabulary of the model we train. So, for simplification, we let the model handle any potential unknown words. Given that tokenization also splits tokens into sub-tokens it becomes even more unlikely we will encounter unknown words.

Implementations

The following section will briefly describe the most important pieces of code relevant to training, evaluation, validation, and inference which made it possible to perform NER. Decoding is a part of the validation and inference code pieces.

Algorithms

- *fix_long_sequences*: this function will take in a pandas DataFrame as an input and correct long sequences according to the *limit* defined for maximum length of a sequence. It will also need to be run multiple times to accommodate for sequences that are much greater than twice the *limit.*

```
def fix_long_sequences(df):
  fixed_df = pd.DataFrame(columns=['text', 'index', 'NER'])

  for i, row in df.iterrows():
    text = row['text']
    index = row['index']
    ner = row['NER']

    if len(text) > limit:
      print(limit)
      text1, text2 = text[:limit], text[limit:]
      index1, index2 = index[:limit], index[limit:]
      ner1, ner2 = ner[:limit], ner[limit:]

      fixed_df.loc[len(fixed_df)] = [text1, index1, ner1]
      fixed_df.loc[len(fixed_df)] = [text2, index2, ner2]

    else:
      fixed_df.loc[len(fixed_df)] = [text, index, ner]


  return fixed_df
```

- *address_subwords:* this function will take in a dataset and create an array with the appropriate *label_id* which is retrieved from the NER data. It will also only consider tokens once, so it will label any subtokens or BERT [CLS], [SEP], or [PAD] tags with -100.

```
def address_subwords(data):
  tokenized_inputs = tokenizer(list(data["text"]), padding='max_length', max_length=split_limit, truncation=True, is_split_into_words=True)

  labels = []
  for i, tag in enumerate(data['NER']):
    word_ids = tokenized_inputs.word_ids(batch_index=i)

    prev = None
    label_ids = []

    for idx in word_ids:
        if idx is None:
            label_ids.append(-100)

        elif idx != prev and idx < len(data['text'][i]): # tokens with tags in the text
          label_ids.append(tag2idx[tag[idx]])

        else:
            label_ids.append(-100)

        prev = idx

    labels.append(label_ids)

  tokenized_inputs['labels'] = labels
  return tokenized_inputs
```

2 modeling and learning 20 / 20

✓ **- 0 pts** Correct

- 0 pts (15pts) Describe your model. If you use a **neural network, you should define the architecture**. There's no need to write equations of common units (e.g., LSTM), so you can just use **functions** to refer to using such **units**. If you use linear models with **features**, provide **details**. Briefly describe what **learning algorithm **you are using. Specify your **learning objective**; Briefly Describe your **inference** procedure. This it the process you use to make **predictions** during testing.

- 3 pts For student using `neural network`: No description of your model. For instance, using **functions** to refer to using such **units**(e.g., LSTM).

- 3 pts For student using `linear models`: No description of your model. For instance, If you use linear models with **features**, provide **details**.

- 2 pts No description about what **learning algorithm **you are using.

Supervised learning? Or what? (other possible algo: semi-supervised? boot strapping?)
Do you do multiclass classification? Or binary?

- 2 pts Not mentioned your **learning objective**
what is your objective function? What is your loss? You may use cross entropy?(How do you calculate the loss between the golden label and predict result)

- 2 pts No description for your **inference** procedure. _(This it the process you use to make **predictions** during testing.)_

the inference procedure can be performed by feeding each sentence in the test set into the trained network and obtaining the eg. Softmax probability for each token(Based on how you design). (And I think the reason to mentioned how to inference is because some time training way differ from predicting. Some architecture may be used in training but not in testing, that's why it is important to write down how you design your inference stage)

- 0 pts (5pts) Provide a description of how you handle **unknown** words as expressed in your model design. If you don't do anything in your model to handle unknown words, you should justify why this is the right approach. For example, if this is something that happens in learning only (or handled by the library), justify this **choice** and provide **details**.

- 3 pts No descriptions for **unknown word** handling

Validation Procedure

For our validation procedure we utilize our tokenized validation dataset from earlier. We use a *DataLoader* object to process our validation data in batches. We retrieve the *attention_mask*, *input_ids*, and *label_ids* from the batch. We utilize these as inputs passed into our model to generate an output. We then utilize the loss and logits obtained from this output to create our prediction and true-labeled data. Once we have obtained our prediction and true-labeled data we utilize the *format_output_labels* function provided to retrieve our *y_pred_dict* and *y_true_dict* values. An important note to consider is empty dictionaries caused by our initial data-preprocessing. We omit these dictionaries when computing the F1-score utilizing the *mean_f1* function provided and summing over all validation samples and then dividing by their length to obtain the final F1-score.

Inference Procedure

For our inference procedure we utilize our tokenized test dataset from earlier. We retrieve the text for each sample and then tokenize the text to retrieve the *attention_mask*, *input_ids*, and *label_ids*. We utilize these as inputs passed into our model to generate an output. We then utilize the logits obtained from this output to create our prediction data. An important note to consider is that *label_ids* is generated from the text from the sample and is used to determine which logits to keep. We use a function similar to *address_subwords*, called *address_subwords_single* (the only difference being it is for a single piece of text), to generate our *label_ids*. Once we have our prediction data we do some additional preprocessing on our predictions and utilize the *create_submission* function provided to create our submissions.

Unknown Word Handling

Since BERT was trained on a large corpus, we find that most words are not unknown and can be handled by the extensive vocabulary of the model we train. So, for simplification, we let the model handle any potential unknown words. Given that tokenization also splits tokens into sub-tokens it becomes even more unlikely we will encounter unknown words.

Implementations

The following section will briefly describe the most important pieces of code relevant to training, evaluation, validation, and inference which made it possible to perform NER. Decoding is a part of the validation and inference code pieces.

Algorithms

- *fix_long_sequences*: this function will take in a pandas DataFrame as an input and correct long sequences according to the *limit* defined for maximum length of a sequence. It will also need to be run multiple times to accommodate for sequences that are much greater than twice the *limit.*

```
def fix_long_sequences(df):
  fixed_df = pd.DataFrame(columns=['text', 'index', 'NER'])

  for i, row in df.iterrows():
    text = row['text']
    index = row['index']
    ner = row['NER']

    if len(text) > limit:
      print(limit)
      text1, text2 = text[:limit], text[limit:]
      index1, index2 = index[:limit], index[limit:]
      ner1, ner2 = ner[:limit], ner[limit:]

      fixed_df.loc[len(fixed_df)] = [text1, index1, ner1]
      fixed_df.loc[len(fixed_df)] = [text2, index2, ner2]

    else:
      fixed_df.loc[len(fixed_df)] = [text, index, ner]


  return fixed_df
```

- *address_subwords:* this function will take in a dataset and create an array with the appropriate *label_id* which is retrieved from the NER data. It will also only consider tokens once, so it will label any subtokens or BERT [CLS], [SEP], or [PAD] tags with -100.

```
def address_subwords(data):
  tokenized_inputs = tokenizer(list(data["text"]), padding='max_length', max_length=split_limit, truncation=True, is_split_into_words=True)

  labels = []
  for i, tag in enumerate(data['NER']):
    word_ids = tokenized_inputs.word_ids(batch_index=i)

    prev = None
    label_ids = []

    for idx in word_ids:
        if idx is None:
            label_ids.append(-100)

        elif idx != prev and idx < len(data['text'][i]): # tokens with tags in the text
          label_ids.append(tag2idx[tag[idx]])

        else:
            label_ids.append(-100)

        prev = idx

    labels.append(label_ids)

  tokenized_inputs['labels'] = labels
  return tokenized_inputs
```

- *evaluation:* this piece of code will process batches of validation data by retrieving the *labels, attention_masks,* and *input_ids* - all of which are used as inputs to pass into the model for inference. We retrieve the loss and logit values and compute the predicted labels, making sure to ignore the aforementioned -100 values for tokens we do not care about. Once we have obtained our prediction and true-labeled data we utilize the *format_output_labels* function provided to retrieve our *y_pred_dict* and *y_true_dict* values. An important note to consider is empty dictionaries caused by our initial data-preprocessing. We omit these dictionaries when computing the F1-score utilizing the *mean_f1* function provided and summing over all validation samples and then dividing by their length to obtain the final F1-score.

```python
f1 = 0
count = 0

for batch in eval_dataloader:

  indices = torch.cat(batch['index'])
  labels = torch.cat(batch['labels'])
  mask = torch.cat(batch['attention_mask'])
  input_ids = torch.cat(batch['input_ids'])

  labels = labels.reshape(1,labels.size()[0]).to(device)
  mask = mask.reshape(1,mask.size()[0]).to(device)
  input_ids = input_ids.reshape(1,input_ids.size()[0]).to(device)


  b = {'labels': labels, 'attention_mask': mask, 'input_ids': input_ids}

  outputs = model(**b)

  loss = outputs.loss
  logits = outputs.logits

  for i in range(logits.shape[0]):

    predictions = logits[i][labels[i] != -100]
    true = labels[i][labels[i] != -100]


    predictions = predictions.argmax(dim=1).tolist()
    true = true.tolist()


    predictions_labeled = [idx2tag[i] for i in predictions]
    true_labeled = [idx2tag[i] for i in true]

    indices = indices.tolist()

    y_pred_dict = format_output_labels(predictions_labeled, indices)
    y_true_dict = format_output_labels(true_labeled, indices)


    empty = {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
    if y_pred_dict != empty and y_true_dict != empty:
      f1 += mean_f1(y_pred_dict, y_true_dict)


    count += 1
print(f1)
f1_total = f1 / len(eval_dataloader)
print(f'F1-Score: {f1_total: .4f}')
```

- *inference:* this piece of code will process the tokenized testing data by retrieving the *labels, attention_masks,* and *input_ids* - where only *attention_masks and input_ids* are used as inputs into the model for inference. We retrieve the logit values and utilize *labels* to determine which values to ignore when computing our predicted labels.

```python
out_text = []
out_pred = []
out_indices = []

for data in test_tokenized_datasets:

  example = data['text']
  indices = data['index']
  text = tokenizer(example, padding='max_length', max_length=split_limit, truncation=True, is_split_into_words=True, return_tensors='pt')

  mask = text['attention_mask'].to(device)
  input_ids = text['input_ids'].to(device)
  label_ids = torch.Tensor(address_subwords_single(example)).unsqueeze(0).to(device)

  b = {'attention_mask': mask, 'input_ids': input_ids}

  outputs = model(**b)

  logits = outputs.logits

  logits_clean = logits[label_ids != -100]

  predictions = logits_clean.argmax(dim=1).tolist()
  predictions_labeled = [idx2tag[i] for i in predictions]

  out_text.append(example)
  out_pred.append(predictions_labeled)
  out_indices.append(indices)
```

Packages

Here we will list some additional packages we utilized that were previously not included in the starter repository.

- **Transformers -** used to access powerful pre-trained models
  - **AutoTokenizer -** used to create a “bert-base-cased” tokenizer
  - **AutoModelForTokenClassification -** used to create a “bert-base-cased” model
  - **AutoConfig -** used to pass configuration options for a “bert-base-cased” model into our model
  - **DataCollatorForTokenClassification -** dynamically pads inputs received
  - **TrainingArguments -** used to specify training arguments for optimization
  - **Trainer -** used to conduct training and evaluation on datasets
- **Datasets**
  - **Dataset -** used to create Dataset

- **Sklearn**
  - **Train_test_split -** used to split training data into training and validation sets
- **Torch**
  - **DataLoader -** used to load data for processing validation data
- **Pandas -** used for data preprocessing purposes
- **MatPlotLib -** used for exploratory data analysis

Experiments and Results

Development Results

Recall that we used an AdamW optimizer with values of $\beta_1 = 0.9$, $\beta_2 = 0.9$, and a weight decay of 0.00001. We kept these constant and decided to alter our epochs and learning rates as hyperparameters. To develop our results we focused on the validation f1-score that we computed for each model. We chose our learning rates from values of: 1e-4, 5e-5, and 5e-4. For each learning rate we ran a different number of epochs: 3, 6, and 10. An interesting note of observation is that a learning rate of 1e-4 yielded the best results consistently. Given that there is a degree of randomness, we see this holds consistent throughout different epoch lengths. While 3 epochs is generally considered sufficient for training a classification model, we went all the way up to 10 epochs. Interestingly enough, we note that our best model was produced with 10 epochs and a learning rate of 1e-4 and a lower validation f1-score compared to its predecessors. Something else to note is that a high learning rate of 5e-4 yielded poor results as the model converted to suboptimal solutions. We also note that training this learning rate for 10 epochs had no positive results as our f1-score and kaggle score were both 0.

Epoch	Learning Rate	Validation F1-Score	Kaggle Score
3	1.00E-04	0.9009	0.92915
6	1.00E-04	0.9163	0.93109
10	1.00E-04	0.8891	0.9357
3	5.00E-05	0.9292	0.9262
6	5.00E-05	0.9012	0.92839
10	5.00E-05	0.9149	0.935
3	5.00E-04	0.7269	0.77281
6	5.00E-04	0.8194	0.83528
10	5.00E-04	0	0

Test Results Presentation

The best test results from the leaderboard were given by our model which was trained for 10 epochs at a learning rate of 1e-4. It achieved a validation f1-score of 0.8891 and a Kaggle Score of 0.9375.

#	Team	Members	Score	Entries	Last	Code	Join
1	SmChen		0.94071	1	2d		
2	**Humza1729**		0.93570	11	1s		

Leaderboard at time of writing report (Humza1729)

Error Analysis

Misclassified examples:

- Example 1:

```
--------------------------------------------
entity-level f1-score:  nan
text:   ['newsroom', ',', '+361', '266', '2410']
index:  [144243, 144244, 144245, 144246, 144247]
NER:    ['O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
--------------------------------------------
```

  - While not "technically" a misclassified sample, samples such as this one caused a lot of headaches as our entity-level f1-score is always going to be NAN whenever our predicted and true label dictionaries are empty. This happens because of how we pre-process our data to split it up into sequences of maximum length *limit*. To remedy this, we simply sought out these empty dictionaries and did not take the entity-level f1-score into account.

3 implementations 15 / 15

- **- 0 pts** Correct

✓ **- 0 pts Explain how you implemented the model (e.g. which **algorithms/data structures** you used). What existing **packages** are used. When describing your implementation details (e.g. **decoding**), we suggest that you include corresponding but only **important code pieces **(e.g. `classes`, `data structures`, `algorithms`, maybe in the form of **screenshot**).**

- **- 5 pts** no proper code files in GitHub repo
- **- 5 pts** no link to GitHub repo

- **Sklearn**
  - **Train_test_split -** used to split training data into training and validation sets
- **Torch**
  - **DataLoader -** used to load data for processing validation data
- **Pandas -** used for data preprocessing purposes
- **MatPlotLib -** used for exploratory data analysis

Experiments and Results

Development Results

Recall that we used an AdamW optimizer with values of β_1 = 0.9, β_2 = 0.9, and a weight decay of 0.00001. We kept these constant and decided to alter our epochs and learning rates as hyperparameters. To develop our results we focused on the validation f1-score that we computed for each model. We chose our learning rates from values of: 1e-4, 5e-5, and 5e-4. For each learning rate we ran a different number of epochs: 3, 6, and 10. An interesting note of observation is that a learning rate of 1e-4 yielded the best results consistently. Given that there is a degree of randomness, we see this holds consistent throughout different epoch lengths. While 3 epochs is generally considered sufficient for training a classification model, we went all the way up to 10 epochs. Interestingly enough, we note that our best model was produced with 10 epochs and a learning rate of 1e-4 and a lower validation f1-score compared to its predecessors. Something else to note is that a high learning rate of 5e-4 yielded poor results as the model converted to suboptimal solutions. We also note that training this learning rate for 10 epochs had no positive results as our f1-score and kaggle score were both 0.

Epoch	Learning Rate	Validation F1-Score	Kaggle Score
3	1.00E-04	0.9009	0.92915
6	1.00E-04	0.9163	0.93109
10	1.00E-04	0.8891	0.9357
3	5.00E-05	0.9292	0.9262
6	5.00E-05	0.9012	0.92839
10	5.00E-05	0.9149	0.935
3	5.00E-04	0.7269	0.77281
6	5.00E-04	0.8194	0.83528
10	5.00E-04	0	0

Test Results Presentation

The best test results from the leaderboard were given by our model which was trained for 10 epochs at a learning rate of 1e-4. It achieved a validation f1-score of 0.8891 and a Kaggle Score of 0.9375.

#	Team	Members	Score	Entries	Last	Code	Join
1	SmChen		0.94071	1	2d		
2	**Humza1729**		0.93570	11	1s		

Leaderboard at time of writing report (Humza1729)

Error Analysis

Misclassified examples:

```
--------------------------------------------
entity-level f1-score:  nan
text:   ['newsroom', ',', '+361', '266', '2410']
index:  [144243, 144244, 144245, 144246, 144247]
NER:    ['O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
--------------------------------------------
```

- Example 1:
  - While not "technically" a misclassified sample, samples such as this one caused a lot of headaches as our entity-level f1-score is always going to be NAN whenever our predicted and true label dictionaries are empty. This happens because of how we pre-process our data to split it up into sequences of maximum length *limit*. To remedy this, we simply sought out these empty dictionaries and did not take the entity-level f1-score into account.

4.1 dev results 12 / 15

- **- 0 pts** Correct

✓ **- 0 pts** **Briefly explain here how you **evaluated** the models and **metrics** used. Put the results into clear **tables** or **diagrams** and include your **observations** and **analysis**.**

- **- 3 pts** no explanations of metrics

✓ **- 3 pts** **no precision and recall**

- **- 3 pts** no tables
- **- 2 pts** table/diagrams not clearly visible
- **- 3 pts** no observations (xx model achieves ..% F1)

- **Sklearn**
  - **Train_test_split -** used to split training data into training and validation sets
- **Torch**
  - **DataLoader -** used to load data for processing validation data
- **Pandas -** used for data preprocessing purposes
- **MatPlotLib -** used for exploratory data analysis

Experiments and Results

<u>Development Results</u>

Recall that we used an AdamW optimizer with values of $\beta_1 = 0.9$, $\beta_2 = 0.9$, and a weight decay of 0.00001. We kept these constant and decided to alter our epochs and learning rates as hyperparameters. To develop our results we focused on the validation f1-score that we computed for each model. We chose our learning rates from values of: 1e-4, 5e-5, and 5e-4. For each learning rate we ran a different number of epochs: 3, 6, and 10. An interesting note of observation is that a learning rate of 1e-4 yielded the best results consistently. Given that there is a degree of randomness, we see this holds consistent throughout different epoch lengths. While 3 epochs is generally considered sufficient for training a classification model, we went all the way up to 10 epochs. Interestingly enough, we note that our best model was produced with 10 epochs and a learning rate of 1e-4 and a lower validation f1-score compared to its predecessors. Something else to note is that a high learning rate of 5e-4 yielded poor results as the model converted to suboptimal solutions. We also note that training this learning rate for 10 epochs had no positive results as our f1-score and kaggle score were both 0.

Epoch	Learning Rate	Validation F1-Score	Kaggle Score
3	1.00E-04	0.9009	0.92915
6	1.00E-04	0.9163	0.93109
10	1.00E-04	0.8891	0.9357
3	5.00E-05	0.9292	0.9262
6	5.00E-05	0.9012	0.92839
10	5.00E-05	0.9149	0.935
3	5.00E-04	0.7269	0.77281
6	5.00E-04	0.8194	0.83528
10	5.00E-04	0	0

<u>Test Results Presentation</u>

The best test results from the leaderboard were given by our model which was trained for 10 epochs at a learning rate of 1e-4. It achieved a validation f1-score of 0.8891 and a Kaggle Score of 0.9375.

#	Team	Members	Score	Entries	Last	Code	Join
1	SmChen		0.94071	1	2d		
2	**Humza1729**		0.93570	11	1s		

Leaderboard at time of writing report (Humza1729)

<u>Error Analysis</u>

Misclassified examples:

- Example 1:

```
--------------------------------------------
entity-level f1-score:  nan
text:   ['newsroom', ',', '+361', '266', '2410']
index:  [144243, 144244, 144245, 144246, 144247]
NER:    ['O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
--------------------------------------------
```

  - While not "technically" a misclassified sample, samples such as this one caused a lot of headaches as our entity-level f1-score is always going to be NAN whenever our predicted and true label dictionaries are empty. This happens because of how we pre-process our data to split it up into sequences of maximum length *limit*. To remedy this, we simply sought out these empty dictionaries and did not take the entity-level f1-score into account.

4.2 describing test results on leaderboard 5 / 5

- **- 0 pts** Correct
- ✓ **- 0 pts Briefly describe your test results from the **leaderboard**. This must match the number on the leaderboard. Put a **screen shot** from the leaderboard or put your "`team name (your name)`" here.**
- **- 3 pts** no kaggle name or screenshot

- **Sklearn**
  - **Train_test_split -** used to split training data into training and validation sets
- **Torch**
  - **DataLoader -** used to load data for processing validation data
- **Pandas -** used for data preprocessing purposes
- **MatPlotLib -** used for exploratory data analysis

Experiments and Results

Development Results

Recall that we used an AdamW optimizer with values of $\beta_1 = 0.9$, $\beta_2 = 0.9$, and a weight decay of 0.00001. We kept these constant and decided to alter our epochs and learning rates as hyperparameters. To develop our results we focused on the validation f1-score that we computed for each model. We chose our learning rates from values of: 1e-4, 5e-5, and 5e-4. For each learning rate we ran a different number of epochs: 3, 6, and 10. An interesting note of observation is that a learning rate of 1e-4 yielded the best results consistently. Given that there is a degree of randomness, we see this holds consistent throughout different epoch lengths. While 3 epochs is generally considered sufficient for training a classification model, we went all the way up to 10 epochs. Interestingly enough, we note that our best model was produced with 10 epochs and a learning rate of 1e-4 and a lower validation f1-score compared to its predecessors. Something else to note is that a high learning rate of 5e-4 yielded poor results as the model converted to suboptimal solutions. We also note that training this learning rate for 10 epochs had no positive results as our f1-score and kaggle score were both 0.

Epoch	Learning Rate	Validation F1-Score	Kaggle Score
3	1.00E-04	0.9009	0.92915
6	1.00E-04	0.9163	0.93109
10	1.00E-04	0.8891	0.9357
3	5.00E-05	0.9292	0.9262
6	5.00E-05	0.9012	0.92839
10	5.00E-05	0.9149	0.935
3	5.00E-04	0.7269	0.77281
6	5.00E-04	0.8194	0.83528
10	5.00E-04	0	0

Test Results Presentation

The best test results from the leaderboard were given by our model which was trained for 10 epochs at a learning rate of 1e-4. It achieved a validation f1-score of 0.8891 and a Kaggle Score of 0.9375.

#	Team	Members	Score	Entries	Last	Code	Join
1	SmChen		0.94071	1	2d		
2	**Humza1729**		0.93570	11	1s		

Leaderboard at time of writing report (Humza1729)

Error Analysis

Misclassified examples:

- Example 1:

```
--------------------------------------------
entity-level f1-score:  nan
text:   ['newsroom', ',', '+361', '266', '2410']
index:  [144243, 144244, 144245, 144246, 144247]
NER:    ['O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
--------------------------------------------
```

  - While not "technically" a misclassified sample, samples such as this one caused a lot of headaches as our entity-level f1-score is always going to be NAN whenever our predicted and true label dictionaries are empty. This happens because of how we pre-process our data to split it up into sequences of maximum length *limit*. To remedy this, we simply sought out these empty dictionaries and did not take the entity-level f1-score into account.

4.3 test performance 15 / 15

+ **0 pts** Correct

+ **15 pts** 15 if F1 score ≥ 82,

14, elif F1≥ 80,

13, elif F1≥ 75,

12, elif F1≥ 70,

11, elif F1≥ 65,

10, elif F1≥ 59,

8, elif F1≥ 55,

6, elif F1≥ 45,

4, elif F1≥ 35,

2 elif F1≥ 25,

0, otherwise.

+ **15** **Point adjustment**

- **Sklearn**
    - **Train_test_split -** used to split training data into training and validation sets
- **Torch**
    - **DataLoader -** used to load data for processing validation data
- **Pandas -** used for data preprocessing purposes
- **MatPlotLib -** used for exploratory data analysis

Experiments and Results

Development Results

Recall that we used an AdamW optimizer with values of $\beta_1 = 0.9$, $\beta_2 = 0.9$, and a weight decay of 0.00001. We kept these constant and decided to alter our epochs and learning rates as hyperparameters. To develop our results we focused on the validation f1-score that we computed for each model. We chose our learning rates from values of: 1e-4, 5e-5, and 5e-4. For each learning rate we ran a different number of epochs: 3, 6, and 10. An interesting note of observation is that a learning rate of 1e-4 yielded the best results consistently. Given that there is a degree of randomness, we see this holds consistent throughout different epoch lengths. While 3 epochs is generally considered sufficient for training a classification model, we went all the way up to 10 epochs. Interestingly enough, we note that our best model was produced with 10 epochs and a learning rate of 1e-4 and a lower validation f1-score compared to its predecessors. Something else to note is that a high learning rate of 5e-4 yielded poor results as the model converted to suboptimal solutions. We also note that training this learning rate for 10 epochs had no positive results as our f1-score and kaggle score were both 0.

Epoch	Learning Rate	Validation F1-Score	Kaggle Score
3	1.00E-04	0.9009	0.92915
6	1.00E-04	0.9163	0.93109
10	1.00E-04	0.8891	0.9357
3	5.00E-05	0.9292	0.9262
6	5.00E-05	0.9012	0.92839
10	5.00E-05	0.9149	0.935
3	5.00E-04	0.7269	0.77281
6	5.00E-04	0.8194	0.83528
10	5.00E-04	0	0

Test Results Presentation

The best test results from the leaderboard were given by our model which was trained for 10 epochs at a learning rate of 1e-4. It achieved a validation f1-score of 0.8891 and a Kaggle Score of 0.9375.

#	Team	Members	Score	Entries	Last	Code	Join
1	SmChen		0.94071	1	2d		
2	**Humza1729**		0.93570	11	1s		

Leaderboard at time of writing report (Humza1729)

Error Analysis

Misclassified examples:

- Example 1:

```
--------------------------------------------
entity-level f1-score:  nan
text:   ['newsroom', ',', '+361', '266', '2410']
index:  [144243, 144244, 144245, 144246, 144247]
NER:    ['O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [], 'ORG': [], 'PER': []}
--------------------------------------------
```

    - While not “technically” a misclassified sample, samples such as this one caused a lot of headaches as our entity-level f1-score is always going to be NAN whenever our predicted and true label dictionaries are empty. This happens because of how we pre-process our data to split it up into sequences of maximum length *limit*. To remedy this, we simply sought out these empty dictionaries and did not take the entity-level f1-score into account.

- Example 2:

```
---------------------------------------------
entity-level f1-score:  0.5
text:   ['of', 'Nations', '...', 'as', 'a', 'result', 'of', 'the', 'absence', 'of', 'this', 'team', 'from', 'the', 'match', ',', '"', 'CAF', 'said', 'in', 'a', 'statement', '.']
index:  [15616, 15617, 15618, 15619, 15620, 15621, 15622, 15623, 15624, 15625, 15626, 15627, 15628, 15629, 15630, 15631, 15632, 15633, 15634, 15635, 15636, 15637, 15638]
NER:    ['MISC', 'MISC', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'ORG', 'O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [], 'MISC': [(15617, 15617)], 'ORG': [(15633, 15633)], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [(15616, 15617)], 'ORG': [(15633, 15633)], 'PER': []}
---------------------------------------------
```

  - This misclassified sample is another consequence of our pre-processing our data into sequences of maximum length *limit*. In this instance we see [‘of’, ‘Nations’, …] is clearly preceded by something, however it was split off during our data preparation steps. Since context is missing the model is unable to assign the proper tag of “MISC” to “of”.

- Example 3:

```
---------------------------------------------
entity-level f1-score:  0.0
text:   ['delivery', '32,700-32,800', 'Soyoil', 'refined', 'market', 'delivery', '32,900-33,000', '-------------------------------', 'Soymeal', '(', 'in', 'rupees', 'per', 'tonne', ',', 'free', 'on', 'rail-FOR', ')', 'Yellow', 'Black', 'FOR', 'Bombay', '9,800', '8,800', 'FOR', 'Bedi', 'Bunder', '9,800', '8,800', '(', '$', '1=35.73', 'rupees', ')']
index:  [139575, 139576, 139577, 139578, 139579, 139580, 139581, 139582, 139583, 139584, 139585, 139586, 139587, 139588, 139589, 139590, 139591, 139592, 139593, 139594, 139595, 139596, 139597, 139598, 139599, 139600, 139601, 139602, 139603, 139604, 139605, 139606, 139607, 139608, 139609]
NER:    ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'MISC', 'MISC', 'O', 'MISC', 'O', 'O', 'O', 'MISC', 'MISC', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [(139583, 139583)], 'MISC': [], 'ORG': [(139597, 139597), (139601, 139602)], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [(139594, 139595), (139597, 139597), (139601, 139602)], 'ORG': [], 'PER': []}
---------------------------------------------
```

  - In this sample we see nothing is labeled correctly. Rather, the model has a tendency to not label tokens as “MISC” - likely due to missing context or too much context. We see that “Soymeal” is considered a location by the model, whereas its true label is “O”. This is likely due to the model being case-sensitive. We also see “Bombay”, “Bedi”, and “Bunder” are being misclassified as “ORG” when they are actually “MISC”. Again, this could be as a result of the model being case-sensitive and considering these entities to be organizations.

Conclusion

Ultimately, we saw that a pre-trained BERT model fine-tuned to Named Entity Recognition proved to be quite powerful as it gave us an accuracy score of 93.57% on Kaggle. We also saw the use of some very powerful APIs and libraries such as HuggingFace which provided us with the “bert-base-cased” pre-trained BERT model for our task. Our findings demonstrated we have room for improvement in our models by determining a better way to split up long sequences that exceed the pre-trained model’s capabilities. For future work, we could also use a combination of case vs uncased pretrained models to determine the best tag for a token.

4.4 error analysis 10 / 10

- 0 pts Correct

✓ **- 0 pts** **Qualitative** analysis of **selected failure examples**. Show and discuss error examples from your **development** set. Identify certain classes of errors and use the examples to illustrate them. Please analyze with respect to **unknown** words as well.

- 6 pts no error cases (showing both sentence plus its predicted tags)

- 3 pts tag level analysis but not instance (sentence/doc) level

- 5 pts no unk analysis

- 3 pts incomplete unk analysis

- 3 pts no general/common identifications (context, unk, features, etc.) of errors (only ad hoc analysis), supported by possibility multiple examples of the same errors

+ 3 pts Speed Analysis (bonus 3pt) (final maximum 100) Speed analysis and computation needs are a major issue of the task. Makes sure to report the costs of inference per example.

Speed analysis and **computation** needs are a major issue of the task. Makes sure to report the costs of **inference** `per example`.

- Example 2:

```
-------------------------------------------
entity-level f1-score:  0.5
text:   ['of', 'Nations', '...', 'as', 'a', 'result', 'of', 'the', 'absence', 'of', 'this', 'team', 'from', 'the', 'match', ',', '"', 'CAF', 'said', 'in', 'a', 'statement', '.']
index:  [15616, 15617, 15618, 15619, 15620, 15621, 15622, 15623, 15624, 15625, 15626, 15627, 15628, 15629, 15630, 15631, 15632, 15633, 15634, 15635, 15636, 15637, 15638]
NER:    ['MISC', 'MISC', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'ORG', 'O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [], 'MISC': [(15617, 15617)], 'ORG': [(15633, 15633)], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [(15616, 15617)], 'ORG': [(15633, 15633)], 'PER': []}
-------------------------------------------
```

  - This misclassified sample is another consequence of our pre-processing our data into sequences of maximum length *limit*. In this instance we see [‘of’, ‘Nations’, …] is clearly preceded by something, however it was split off during our data preparation steps. Since context is missing the model is unable to assign the proper tag of “MISC” to “of”.
- Example 3:

```
-------------------------------------------
entity-level f1-score:  0.0
text:   ['delivery', '32,700-32,800', 'Soyoil', 'refined', 'market', 'delivery', '32,900-33,000', '--------------------------------', 'Soymeal', '(', 'in', 'rupees', 'per', 'tonne', ',', 'free', 'on', 'rail-FOR', ')', 'Yellow'
index:  [139575, 139576, 139577, 139578, 139579, 139580, 139581, 139582, 139583, 139584, 139585, 139586, 139587, 139588, 139589, 139590, 139591, 139592, 139593, 139594, 139595, 139596, 139597, 139598, 139599, 139600, 139601, 1
NER:    ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'MISC', 'MISC', 'O', 'MISC', 'O', 'O', 'O', 'MISC', 'MISC', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
PRED:   {'LOC': [(139583, 139583)], 'MISC': [], 'ORG': [(139597, 139597), (139601, 139602)], 'PER': []}
TRUE:   {'LOC': [], 'MISC': [(139594, 139595), (139597, 139597), (139601, 139602)], 'ORG': [], 'PER': []}
-------------------------------------------
```

```
, 'Black', 'FOR', 'Bombay', '9,800', '8,800', 'FOR', 'Bedi', 'Bunder', '9,800', '8,800', '(', '$', '1=35.73', 'rupees', ')']
39602, 139603, 139604, 139605, 139606, 139607, 139608, 139609]
```

  - In this sample we see nothing is labeled correctly. Rather, the model has a tendency to not label tokens as “MISC” - likely due to missing context or too much context. We see that “Soymeal” is considered a location by the model, whereas its true label is “O”. This is likely due to the model being case-sensitive. We also see “Bombay”, “Bedi”, and “Bunder” are being misclassified as “ORG” when they are actually “MISC”. Again, this could be as a result of the model being case-sensitive and considering these entities to be organizations.

Conclusion

Ultimately, we saw that a pre-trained BERT model fine-tuned to Named Entity Recognition proved to be quite powerful as it gave us an accuracy score of 93.57% on Kaggle. We also saw the use of some very powerful APIs and libraries such as HuggingFace which provided us with the “bert-base-cased” pre-trained BERT model for our task. Our findings demonstrated we have room for improvement in our models by determining a better way to split up long sequences that exceed the pre-trained model’s capabilities. For future work, we could also use a combination of case vs uncased pretrained models to determine the best tag for a token.

5 conclusion 5 / 5

- **- 0 pts** Correct

✓ **- 0 pts Brief conclusion summarizing findings (from both **numerical** results and **qualitative** analysis). Your brief **feedback** for the final (2-3 sentences would suffice).**

6 late penalty and invited talk 0 / 0

- ✓ **- 0 pts** **not attending invited talk (no bonus points)**
- **+ 1 pts** bonus points: attending invited talk `(final maximum 100)`
- **+ 2 pts** bonus points: asking questions (you need to specify the question and the answer in final report) `(final maximum 100)`