Mining and Evaluating Verb tags and Other Important POS tags inside Software Documentation

Design Document

Team 17

Client: Arushi Sharma, & Hiep Vo

Advisor: Ali Jannesari

Team Members/Roles:

William Sengstock - Team Leader

Kelly Jacobson - Team Organization

Jacob Kinser - Component Design

Zachary Witte - Component Design

Samuel Moore - Component Design

Dan Vasudevan - Component Design

Austin Buller - Component Design

Team email: sdmay22-17@iastate.edu

Team website: https://sdmay22-17.sd.ece.iastate.edu/

Table of Contents

1 Team

1.1 Team Members

- Austin Buller
- Kelly Jacobson
- Jacob Kinser
- Samuel Moore
- William Sengstock
- Dan Vasudevan
- Zachary Witte

1.2 Required Skill Sets for Your Project

- Apply knowledge of mathematics, science, and engineering.
- Should identify, formulate, and solve engineering problems.
- Optimal communication skills.
- Knowledge of coding languages and practices.
- Experience with Python, Jupyter Notebooks, and GitHub

1.3 Skill Sets covered by the Team

- Austin Buller - Covers all skills
- Kelly Jacobson - Covers all skills
- Jacob Kinser - Covers all skills
- Samuel Moore - Covers all skills
- William Sengstock - Covers all skills
- Dan Vasudevan - Covers all skills
- Zachary Witte - Covers all skills

1.4 Project Management Style Adopted by the team

We used a waterfall project management style. This was the best style for the project because each week we built on our knowledge and the feedback from our client.

1.5 Initial Project Management Roles

William Sengstock - Team Leader

Kelly Jacobson - Team Organization

Jacob Kinser - Component Design

Zachary Witte - Component Design

Samuel Moore - Component Design

Dan Vasudevan - Component Design

Austin Buller - Component Design

2 Introduction

2.1 Problem Statement

Our project was focused on researching the application of Natural Language Processing techniques to software documentation. The objective was to improve Part of Speech (POS) tagging to make information mining more effective when applied to software languages. Current NLP models are used for English and other natural languages. We wanted to apply these models to software languages like Java, Python, C, etc. and combine both strategies to effectively analyze software documentation containing both natural language and software language.

2.2 Requirements & Constraints

Research Requirements/Constraints:

- Data pre-processing techniques
    - Stemming
    - Lemmatization
    - Tokenization
- Word vectorization
    - Skip-Gram
    - Continuous Bag of Words (CBOW)
- Word embedding and word clustering
- Algorithms
    - Gradient Boosting

  - Random Forests
- Supervised /unsupervised Learning
  - Clustering algorithms
  - Neural Networks
- Existing POS tagging models
  - NLTK
  - SpaCy
  - BERT
  - StanfordNLP
  - Term Frequency - Inverse Document Frequency (TF-IDF)
- Existing Code-Understanding Models
  - CuBERT
  - CodeBERT
- Code Tokenizers
  - CuBERT Tokenizer
  - HuggingFace WordPiece and BPE Tokenizers
  - Python Tokenizer
- Each member of the project is expected to research on their own and cite credible sources

Programming Requirements/Constraints:

- Models will be written in Python using Jupyter Notebook (constraint)
- Finetune CodeBERT with Software Documentation
  - Preprocess Software Documentation dataset to fit CodeBERT's structure
  - Fine-tune CodeBERT model with given Software Documentation dataset
  - Create visual metrics that display the accuracy of the newly fine-tuned model.
- Data used for research will come from approved sources, such as www.kaggle.com, GitHub pages, and those given by the client.
- Dataset must be processable by CodeBERT
- Fine Tuned model should be transferable to other users.
- Evaluate the performance of each model compared to a manual review.

2.3 Engineering Standards

- ISO-IEC 9001: Quality Management
  - Helps ensure quality throughout the project, as well as continuous improvement
- ISO-IEC 830: Software Requirements Specifications
  - Influences the structure of the project, along with communication between users
- ISO-IEC 12207: Software Life Cycle Processes

- Influences project design, maintains product throughout stages

2.4 Intended Users and Uses

Our end goal for this project is to improve natural language processes on software documentation. Our project will directly benefit people researching NLP, including our client, and will be most useful as a reference for other projects. Our research will be the basis of other programs and projects that use POS tagging for software documentation in the long run. For example, a search engine for software documents would utilize POS tagging to identify similar documents. This could help a researcher looking for Python documentation find relevant information. Another example is searching for similar code blocks across various documents. It would also be beneficial to someone inheriting a project who would use the same NLP processes we research to better understand their project. There are many possible uses of NLP and POS as applied to software documentation. Our results will become a basis for future progress by experimenting with different word embeddings, algorithms, etc.

3 Project Design

3.1 Project Evolution

Our project has evolved immensely since CPRE 491. Throughout 491, our team conducted many different experiments regarding Natural Language Processing and Part of Speech tagging with software documentation. These experiments consisted of using and analyzing the performance of existing models like spaCy, Natural Language Toolkit (NLTK), and StanfordNLP.; we also attempted to train models from scratch, which we decided to stray away from. Our team ultimately found those models to be very helpful regarding POS tagging natural language text, but not so much for software documentation. For this semester, instead of disregarding the information we learned and the results we came to in 491, we decided to use it to our advantage. Our team discovered CodeBERT, a pre-trained model for programming languages and natural language. After experimenting and reading about codeBert, we decided that to take the pre-trained CodeBERT model and fine tune it with data of our own to achieve a model that reaches an accuracy higher than either the natural language processors or CodeBERT itself. This is where the research and experimentation conducted in 491 became very helpful. To finetune CodeBERT, we needed a large amount of software data that already had accurate POS tags. We were introduced to the Python Tokenizer that could tag most of the text in software, but needed to be customized, which we did manually and programmatically. This allowed us to finetune our codeBert model on data, ultimately making the model better understand how to POS tag software documentation. Overall, our project evolved from a group of experiments and abundant research to a real, fine-tuned model that utilized the results and information gathered in CPRE 491.

3.2 Security Concerns

Our project is research-focused and does not include any sensitive data or create opportunities for malicious use. Our final deliverable is a python program. The only variable input is the dataset, and the dataset we used was open source. There are no security or safety concerns with our project.

3.3 Implementation Details

Since our project relied heavily on analyzing code, our group decided to use the BERT model, CodeBERT. This is because CodeBERT is already pre-trained on code and would provide us a good foundation to fine tune for our specific purpose. For the CodeBERT model implementation, we used BertForSequenceClassification. This is because we were training the model to predict the tag for each input. We then trained and tested the model on software documentation improve model accuracy.

For the data-processing steps of tokenizing the raw data file, we used the standard Python Tokenizer that is part of the python tokenize library. This tokenizer is made to find tokens in python source code, which makes it perfect for finding tokens in software documentation. We chose a code-understanding tokenizer over the natural language tokenizers (NLTK, SpaCy, etc.) because the challenge is not understanding natural language; the challenge is understanding both natural language and source code. Acode-understanding tokenizer is the best choice. In Appendix I, under the Different Tokenizers section, we talk about why we chose the Python Tokenizer over other code-understanding tokenizers.

4 Testing

4.1 Integration Testing

Our individual modules included pre-processing data, tokenization, model training, and evaluation. Since different modules depend on other modules' outputs as their input, these modules were expected to all run together and produce the desired result. As we were testing different implementations for each module, integration testing was important to ensure our system still produced the correct results. The tool used for integration testing was Python. This is demonstrated in our final deliverable as it is a combination of multiple modules created throughout the semester.

4.2 System Testing

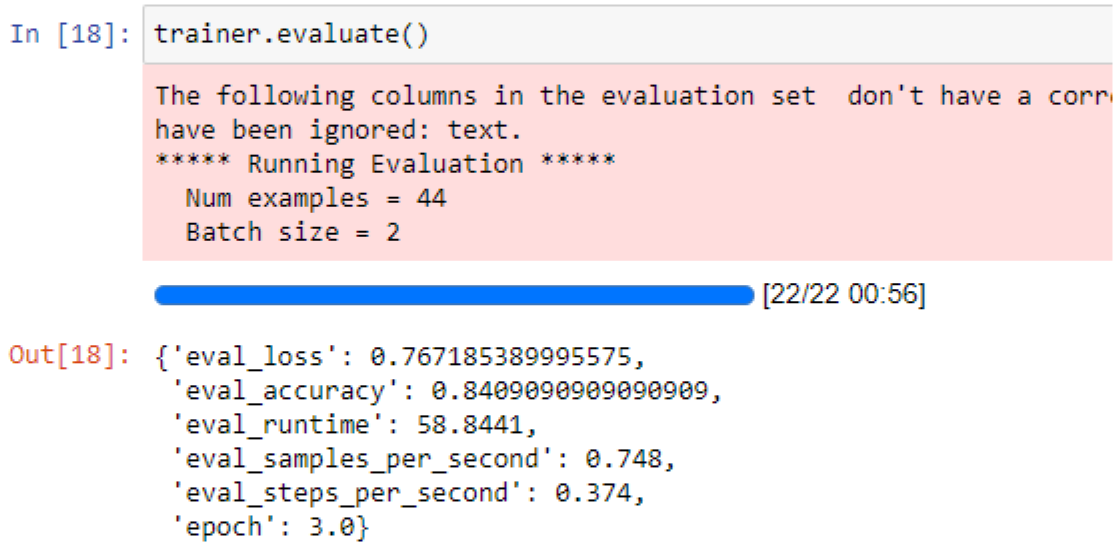
Our system-level testing strategy was to test the end-to-end code of the model. This means that we checked if each step from importing data to outputting accuracy of the NLP model worked as expected. The integration tests validated that data was being fed into the model. With system testing, we had to assess the accuracy of POS tagging and our predictive model altogether, and how it fairs against software documentation. This process will also ensure that our code is written efficiently in terms of runtime and that each part of the code is commented out and well explained. Another aspect of this testing strategy was to ensure that the IDE works as expected. We quickly learned that Google Colab does not allocate enough memory for our model so we did all our training and evaluation on Jupyter Notebook.

4.3 Acceptance Testing

Team members demonstrated that the design requirements were met by showing our client how we adapted software documentation to the various word embeddings. Through communicating with one another and meeting at our weekly times, our clients gave us feedback on whether we were on track or not. We involved our client in the acceptance testing by showing them our progress with training our models. Each week our client reviewed our work and gave us the next set of goals to work on for the next time we met.

4.4 Results

We were able to achieve an 84% accuracy with Part of Speech tagging software documentation.



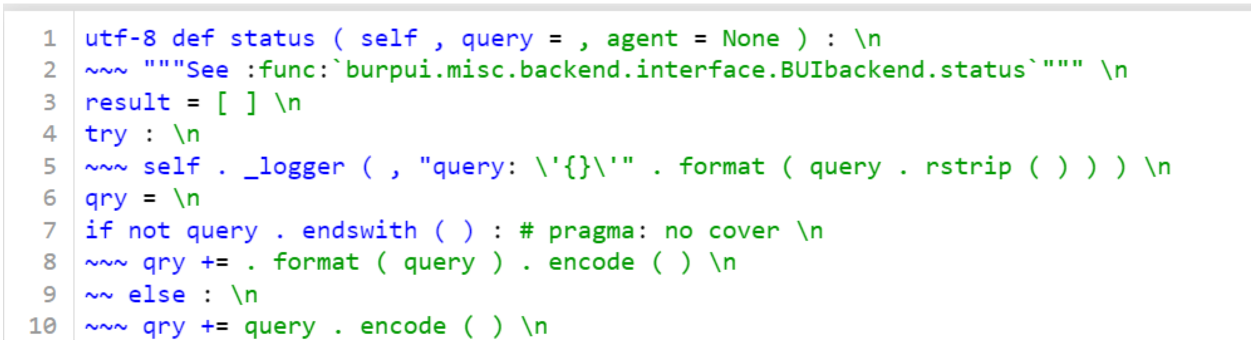
```
In [18]: trainer.evaluate()

The following columns in the evaluation set  don't have a corr
have been ignored: text.
***** Running Evaluation *****
  Num examples = 44
  Batch size = 2

[22/22 00:56]

Out[18]: {'eval_loss': 0.767185389995575,
          'eval_accuracy': 0.8409090909090909,
          'eval_runtime': 58.8441,
          'eval_samples_per_second': 0.748,
          'eval_steps_per_second': 0.374,
          'epoch': 3.0}
```

Appendix

5 Appendix Ⅰ: Operation Manual

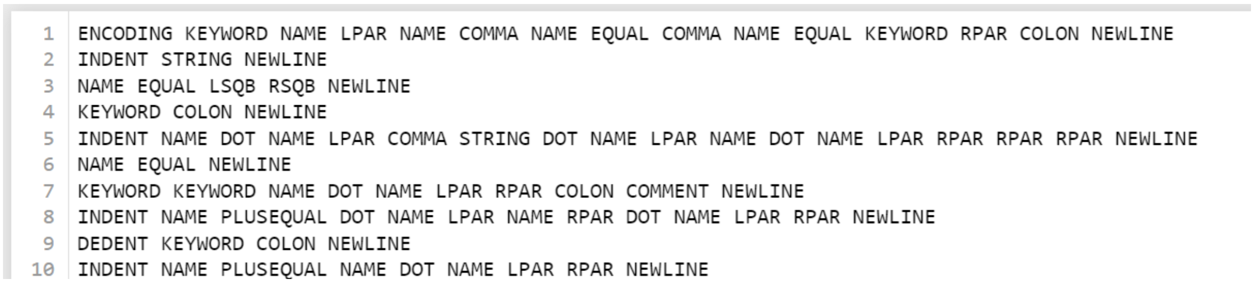
5.1 Step 1: Dataset Processing

First, we must obtain a software documentation dataset that meets the model requirements. Software documentation is text that contains source code, natural language, and code comments. The model is trained on this type of dataset, so it is vital that it is tested on data of the same type. CodeBERT requires the dataset to be formatted so that it has the original “text” of the dataset and the corresponding “label”. Here is an example of the “text” file:



```
utf-8 def status ( self , query = , agent = None ) : \n
~~~ """See :func:`burpui.misc.backend.interface.BUIbackend.status`""" \n
result = [ ] \n
try : \n
~~~ self . _logger ( , "query: \'{}\'" . format ( query . rstrip ( ) ) ) \n
qry = \n
if not query . endswith ( ) : # pragma: no cover \n
~~~ qry += . format ( query ) . encode ( ) \n
~~ else : \n
~~~ qry += query . encode ( ) \n
```

And here is an example of a “label” file:

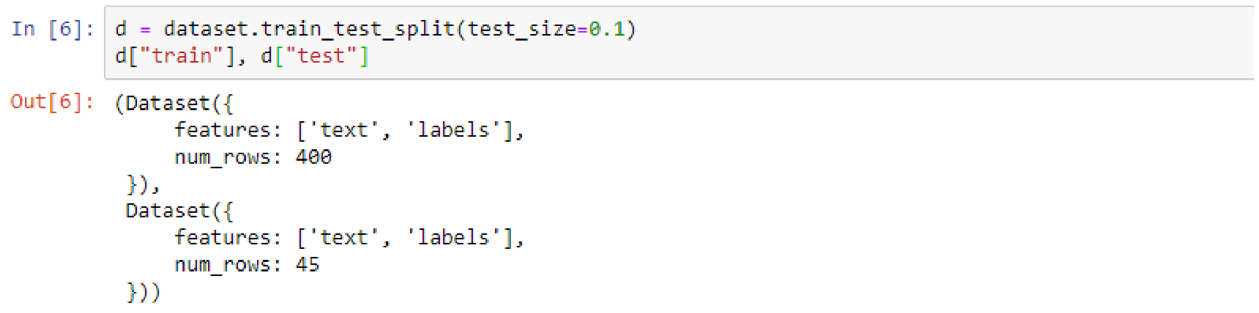


```
ENCODING KEYWORD NAME LPAR NAME COMMA NAME EQUAL COMMA NAME EQUAL KEYWORD RPAR COLON NEWLINE
INDENT STRING NEWLINE
NAME EQUAL LSQB RSQB NEWLINE
KEYWORD COLON NEWLINE
INDENT NAME DOT NAME LPAR COMMA STRING DOT NAME LPAR NAME DOT NAME LPAR RPAR RPAR RPAR NEWLINE
NAME EQUAL NEWLINE
KEYWORD KEYWORD NAME DOT NAME LPAR RPAR COLON COMMENT NEWLINE
INDENT NAME PLUSEQUAL DOT NAME LPAR NAME RPAR DOT NAME LPAR RPAR NEWLINE
DEDENT KEYWORD COLON NEWLINE
INDENT NAME PLUSEQUAL NAME DOT NAME LPAR RPAR NEWLINE
```

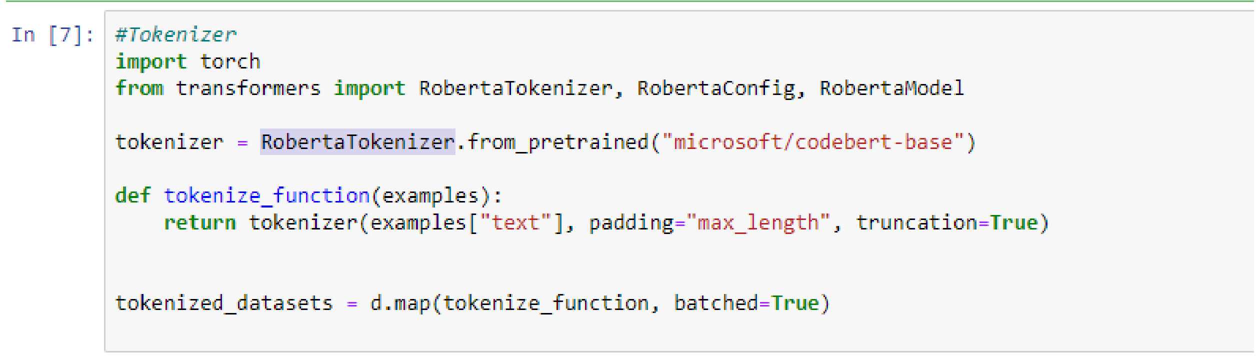
The labels corresponding to the text were found using our modified Python Tokenizer. The Final POS Tags list identifies all possible POS labels for the tokens and the most common pieces of text that will be assigned these labels. Part of data-processing involves taking the software documentation text and using the modified Python Tokenizer to find these labels. Then, the original dataset is split into the text file and the label file for the CodeBERT model to read.

5.2 Step 2: Tokenization

Next we have to tokenize the data set prior to training the model with it. Right now the dataset is split into a test and train dataset. You will need to put this split dataset into the RobertaTokenizer so the CodeBERT model understands it for training. Here is how that is done in our final deliverable:



```
In [6]: d = dataset.train_test_split(test_size=0.1)
        d["train"], d["test"]

Out[6]: (Dataset({
             features: ['text', 'labels'],
             num_rows: 400
         }),
         Dataset({
             features: ['text', 'labels'],
             num_rows: 45
         }))
```



```
In [7]: #Tokenizer
        import torch
        from transformers import RobertaTokenizer, RobertaConfig, RobertaModel

        tokenizer = RobertaTokenizer.from_pretrained("microsoft/codebert-base")

        def tokenize_function(examples):
            return tokenizer(examples["text"], padding="max_length", truncation=True)


        tokenized_datasets = d.map(tokenize_function, batched=True)
```

5.3 Step 3: Training and Evaluation

The next step is to take the tokenized data and train the model on it. To do this you must create a Trainer object that you need to import from the transformers library. In the trainer object there are a few parameters you must pass in. First, you need to pass in the training and testing datasets that you split up and tokenized in the previous step. The testing dataset will later be used for evaluation. Second, you must create a model object and pass in an instance of the RobertaModel. Next, you need to pass in a compute_metrics method which is what the model will use to evaluate the accuracy during the evaluation phase. And finally you need to pass in arguments you would like to tweak for training. To train the data simply run Trainer.train(). And to evaluate the accuracy of your model run Trainer.evaluate(). The evaluate() method should return metrics on how accurate the model is based on the compute_metrics method you created.

5.4 Step 4: Load/Save Deep Learning Model

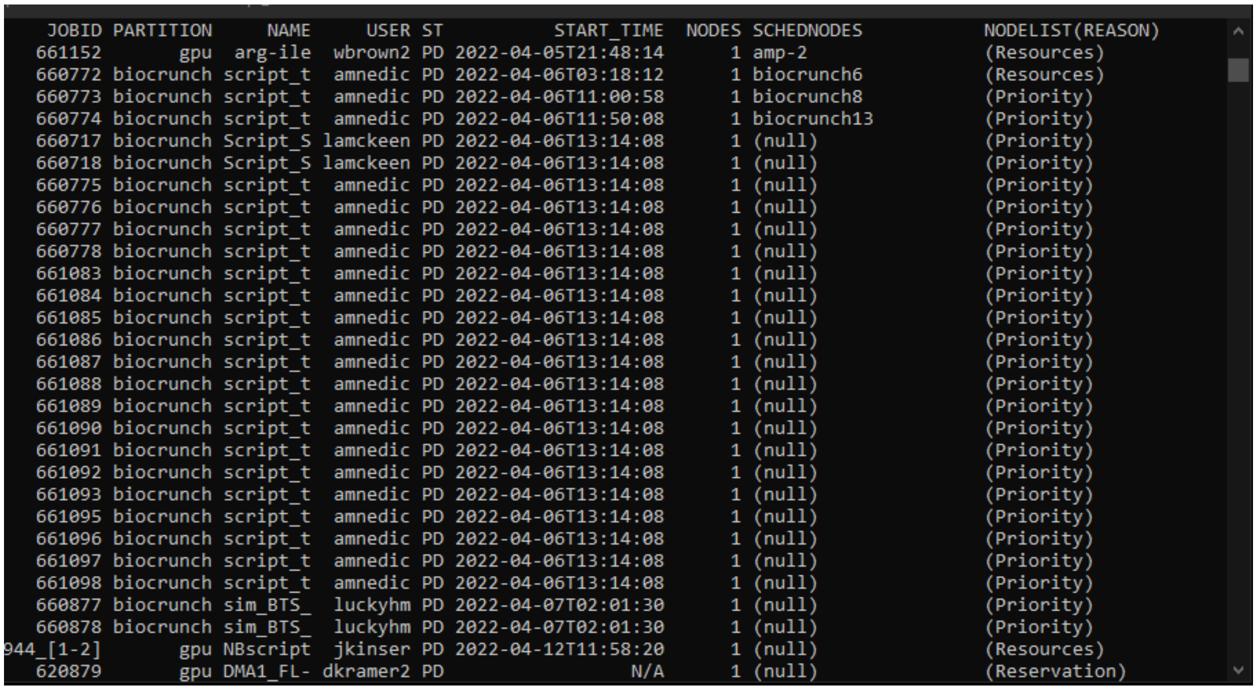
Once you have trained and evaluated your model, you will want to save it so that you can use it in the future on different data. To save the model you need to run the method torch.save(). And if you want to load that saved model you can use the torch.load() method.

6 Appendix Ⅱ: Alternative/Other Initial Versions

6.1 Training CodeBERT from scratch:

In the early stages of creating our model, we were going to attempt to train CodeBERT from scratch. After getting the model created we determined that we were not able to train it on our personal computers and we were then given access to run our model on pronto. Pronto allowed us to allocate the necessary resources that our model would require to train. We quickly noticed that this would not be a viable option as the script to run our model was given low priority (noted in the picture below, netid: jkinser). Our estimated time for the script to run was one week. This is significant because we would be training the model on multiple sets of data over the course of the semester. From here, our group, along with our advisors felt we should finetune the existing model of CodeBERT instead of training from scratch. We felt this would still be beneficial to our project because CodeBERT is already pre-trained on code. The notebook for this model is located in our Github repository under the folder CodeBERTfromScratch

Screenshot of queue (One week wait):



```
   JOBID PARTITION     NAME     USER ST          START_TIME  NODES SCHEDNODES           NODELIST(REASON)
  661152       gpu  arg-ile  wbrown2 PD 2022-04-05T21:48:14      1 amp-2                (Resources)
  660772 biocrunch script_t  amnedic PD 2022-04-06T03:18:12      1 biocrunch6           (Resources)
  660773 biocrunch script_t  amnedic PD 2022-04-06T11:00:58      1 biocrunch8           (Priority)
  660774 biocrunch script_t  amnedic PD 2022-04-06T11:50:08      1 biocrunch13          (Priority)
  660717 biocrunch Script_S lamckeen PD 2022-04-06T13:14:08      1 (null)               (Priority)
  660718 biocrunch Script_S lamckeen PD 2022-04-06T13:14:08      1 (null)               (Priority)
  660775 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  660776 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  660777 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  660778 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661083 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661084 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661085 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661086 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661087 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661088 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661089 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661090 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661091 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661092 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661093 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661095 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661096 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661097 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  661098 biocrunch script_t  amnedic PD 2022-04-06T13:14:08      1 (null)               (Priority)
  660877 biocrunch sim_BTS_  luckyhm PD 2022-04-07T02:01:30      1 (null)               (Priority)
  660878 biocrunch sim_BTS_  luckyhm PD 2022-04-07T02:01:30      1 (null)               (Priority)
944_[1-2]       gpu NBscript  jkinser PD 2022-04-12T11:58:20      1 (null)               (Resources)
  620879       gpu DMA1_FL- dkramer2 PD                 N/A      1 (null)               (Reservation)
```

6.2 Different tokenizers:

In the process of developing this project, we considered several different tokenizers to use with our model. We eventually settled on using the python tokenizer and making some modifications to tailor it to our needs. Some tokenizers we looked at before include CuBERT and a couple of HuggingFace tokenizers.

CuBERT is a Google research project, based on the BERT model. BERT is a language model for natural language processing, using machine learning. CuBERT is a language model intended to be used for understanding code. The concept is very similar to our project so we thought it would be a good idea to include in our model. Unfortunately, CuBERT is extremely hard to use. While it is available to download from GitHub, CuBERT is extremely complex and outside the bounds of our team's knowledge. A few members of our team spent multiple weeks just trying to get a CuBERT example model to run and were unsuccessful. We decided it was not worth our time and moved on to other tokenizers.

There were two HuggingFace tokenizers we looked at. One was a WordPiece tokenizer, and the other was based on Byte Pair Encoding. These are the same types of tokenizers used by the BERT model. We found that when we tried tokenizing python source code with these tokenizers, they did not produce meaningful results.

We finally decided to use the standard Python Tokenizer. This tokenizer is made specifically for identifying tokens in python source code. Our project was meant to be applied to any software documentation containing any code language, but we decided Python was the best starting point. The tokenizer was also easy to use and easy to modify to fit our needs.

7 Appendix Ⅲ: Other Considerations

This senior design project was a great learning experience for each of our team members. Prior to last semester, none of us had ever worked with any machine learning concepts. However, after we started this project we felt like we picked up a new skill each week. Here are a few of the main technical skills we obtained from this project:

1. **Python Data Processing Tools**
    a. Numpy
    b. Matplotlib
    c. Pandas
    d. scikit-learn
2. **Machine Learning Libraries**
    a. Pytorch
    b. Hugging Face
    c. Keras
    d. Codebert
3. **Tokenizers**
    a. Python Tokenizer
    b. AutoTokenizer
4. **NLP Software**

a. NLTK
b. Spacy
c. Stanford NLP

8 Appendix IV: Code

8.1 Github Repository:

Our complete notebook, along with models and other modifications can be found on our teams github repository

Link: https://git.ece.iastate.edu/sd/sdmay22-17/

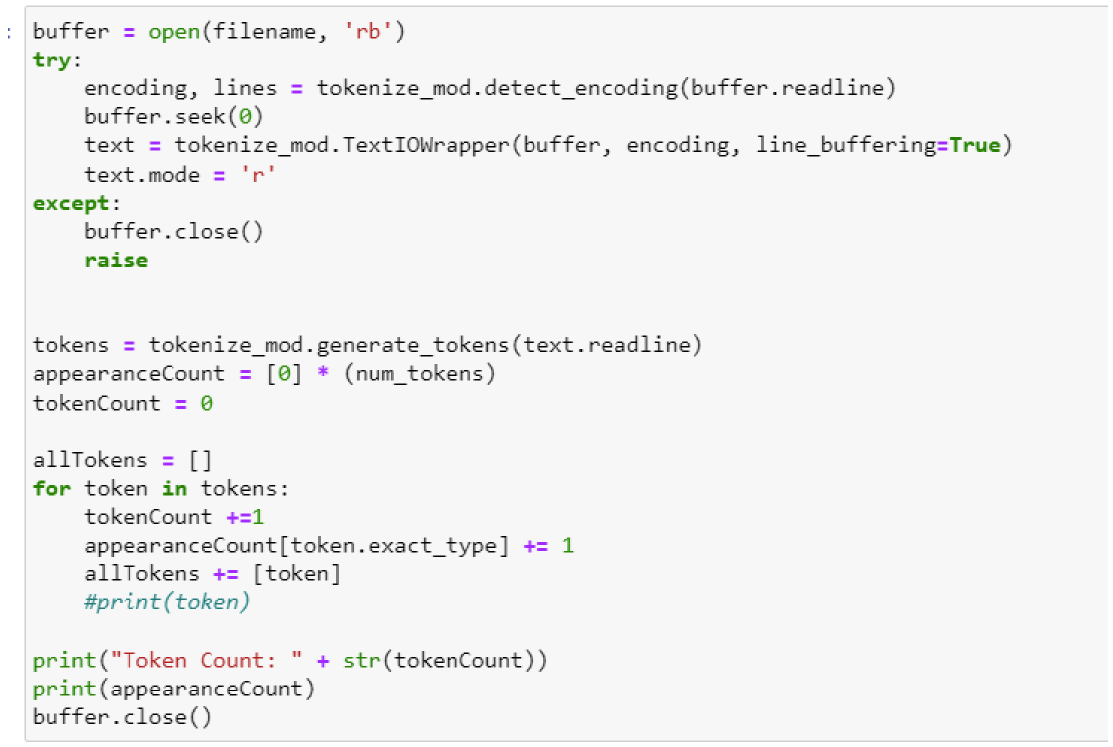
8.2 Python tokenizer:

The original Python Tokenizer already does a great job of tokenizing python source code and assigning labels, but we needed to modify it. We added our own labels, and altered the code to apply those new labels. Our added labels include: KEYWORD, PRINT, WHITESPACE, COND, COND_BLOCK, WHILE, WHILE_BLOCK, FOR, FOR_BLOCK.

KEYWORD identifies the reserved python keywords. PRINT identifies the common print statements. For example `print("Hello, World!")`. WHITESPACE is used as a catchall for any whitespace characters that did not already have a specific label. COND is used to identify `if`, `elif`, and `else` statements. COND_BLOCK is used to identify entire conditional blocks, including everything from the conditional (`if`, `elif`, `else`) to the ending bracket. WHILE identifies the `while` keyword, and WHILE_BLOCK identifies while-loop blocks, similar to the conditional blocks. FOR and FOR_BLOCK behave in a similar way for the `for` keyword.

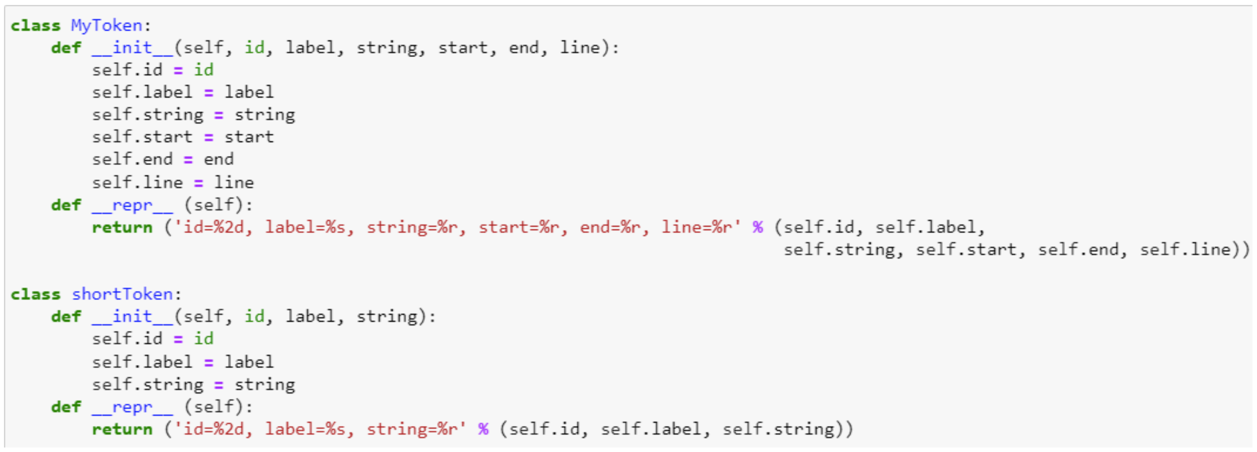
The Python Tokenizer was altered so that tokens that fit these new labels will receive the new labels instead of receiving the NAME label. The NAME token is a catch-all token for most variable, class, and function names. If we wanted to add even more specific labels, the best way to do so is to modify how the tokenizer applies the NAME label.

Now that we have a tokenizer to generate our labels, we need to actually use it on our raw data. Our raw data was a software documentation file that contained both python source code and natural language in the form of comment blocks. Running the Python Tokenizer on this file is a simple task.



```
buffer = open(filename, 'rb')
try:
    encoding, lines = tokenize_mod.detect_encoding(buffer.readline)
    buffer.seek(0)
    text = tokenize_mod.TextIOWrapper(buffer, encoding, line_buffering=True)
    text.mode = 'r'
except:
    buffer.close()
    raise


tokens = tokenize_mod.generate_tokens(text.readline)
appearanceCount = [0] * (num_tokens)
tokenCount = 0

allTokens = []
for token in tokens:
    tokenCount +=1
    appearanceCount[token.exact_type] += 1
    allTokens += [token]
    #print(token)

print("Token Count: " + str(tokenCount))
print(appearanceCount)
buffer.close()
```

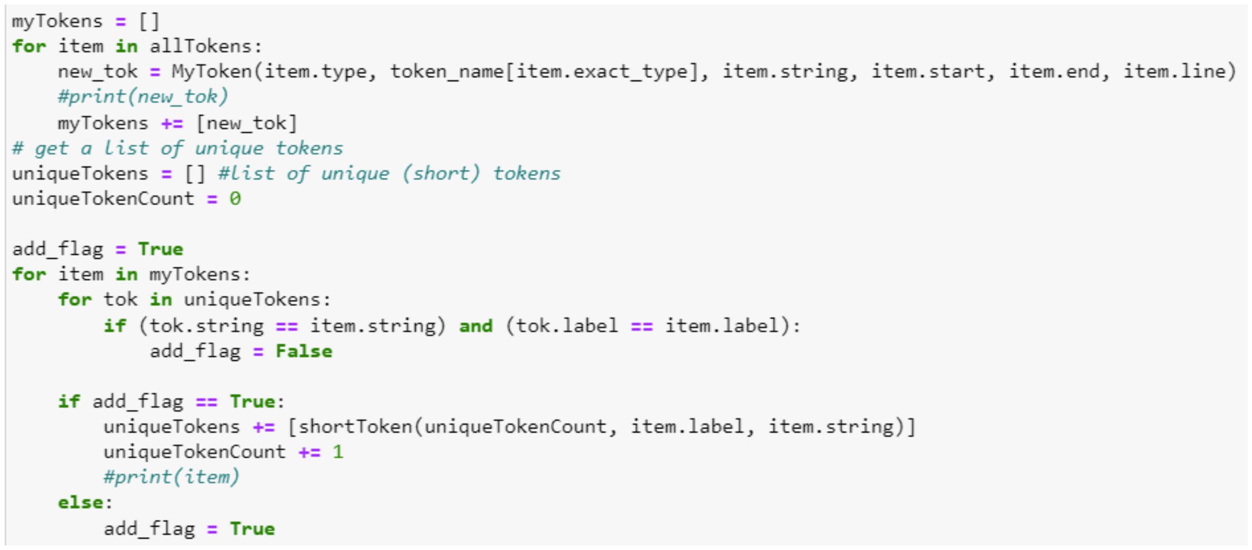
The file must be fed into a buffer, which will allow the tokenizer to generate tokens. We used the array allTokens to remember. Using those tokens, however, is not straightforward. The tokenizer saves them as a custom class with all identifying information including the token label, id, the line it appears on, and the start and end locations. While this is all very nice to know, tuples are difficult to work with. To combat this issue, we created two classes that define our own token types.



```
class MyToken:
    def __init__(self, id, label, string, start, end, line):
        self.id = id
        self.label = label
        self.string = string
        self.start = start
        self.end = end
        self.line = line
    def __repr__ (self):
        return ('id=%2d, label=%s, string=%r, start=%r, end=%r, line=%r' % (self.id, self.label,
                                                                              self.string, self.start, self.end, self.line))

class shortToken:
    def __init__(self, id, label, string):
        self.id = id
        self.label = label
        self.string = string
    def __repr__ (self):
        return ('id=%2d, label=%s, string=%r' % (self.id, self.label, self.string))
```

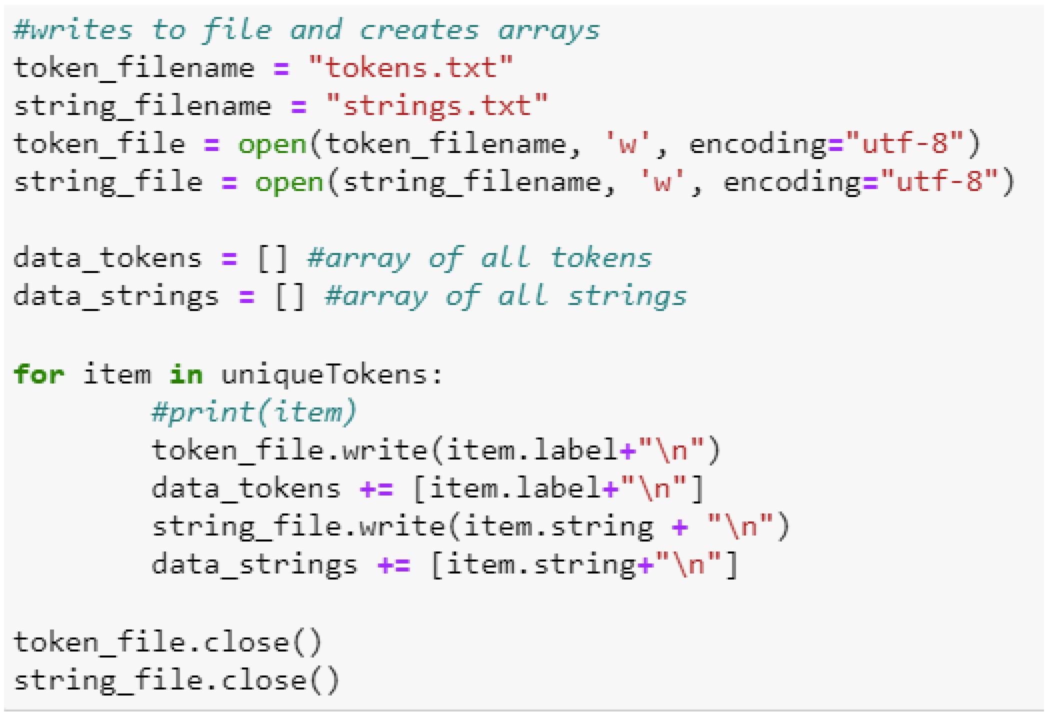
The MyToken class contains all the identifying information in easy to manipulate strings. The shortToken class contains only the id, label, and text-string which is all we need for the CodeBERT model.

The next part of this process was to remove any duplicate label and text pairs. To train the model, we only needed each token to appear once. For example, we did not need to include every single “\n” text with the associated NEWLINE label. We only needed the unique tokens.



```python
myTokens = []
for item in allTokens:
    new_tok = MyToken(item.type, token_name[item.exact_type], item.string, item.start, item.end, item.line)
    #print(new_tok)
    myTokens += [new_tok]
# get a list of unique tokens
uniqueTokens = [] #list of unique (short) tokens
uniqueTokenCount = 0

add_flag = True
for item in myTokens:
    for tok in uniqueTokens:
        if (tok.string == item.string) and (tok.label == item.label):
            add_flag = False

    if add_flag == True:
        uniqueTokens += [shortToken(uniqueTokenCount, item.label, item.string)]
        uniqueTokenCount += 1
        #print(item)
    else:
        add_flag = True
```

Next, split each token into its label and text. Write these tokens to arrays and files that can be loaded into CodeBERT.



```python
#writes to file and creates arrays
token_filename = "tokens.txt"
string_filename = "strings.txt"
token_file = open(token_filename, 'w', encoding="utf-8")
string_file = open(string_filename, 'w', encoding="utf-8")

data_tokens = [] #array of all tokens
data_strings = [] #array of all strings

for item in uniqueTokens:
        #print(item)
        token_file.write(item.label+"\n")
        data_tokens += [item.label+"\n"]
        string_file.write(item.string + "\n")
        data_strings += [item.string+"\n"]

token_file.close()
string_file.close()
```

```
IN = []
for item in data_strings:
    IN.append(item)

LABEL = []
for item in data_tokens:
    all_labels = ["ENDMARKER", "NAME", "NUMBER", "STRING", "NEWL

    i = 0
    j = 0
    for x in all_labels:
        if item == x+"\n":
            j = 1
            LABEL.append(i)
        i = i + 1
```

Finally, create the datasets for use with CodeBERT. It is required that there is a column for strings, the labels, and a unique id for each item.

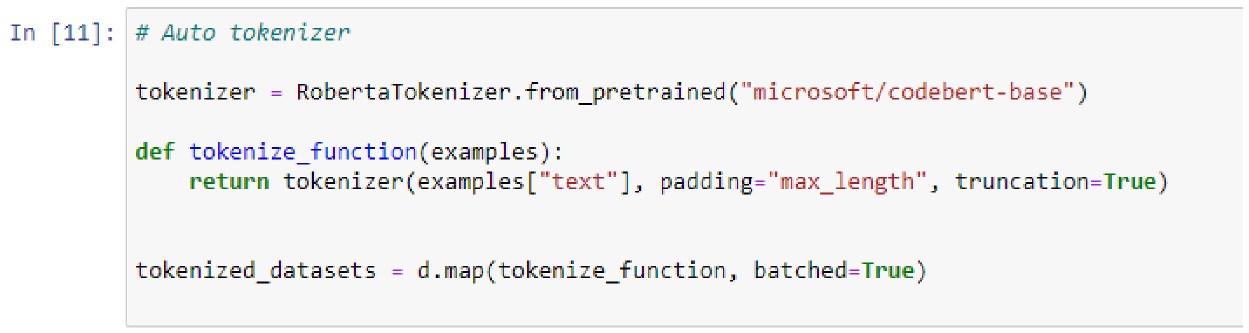
```
# Create the dataset
files = ["strings.txt"]
dataset = load_dataset("text", data_files=files, split="train")
dataset = dataset.add_column("label", LABEL)
```

Then split the dataset into testing and training datasets.

```
# Split the dataset and allocate 10% of inputs to testing
d = dataset.train_test_split(test_size=0.1)
d["train"], d["test"]
```

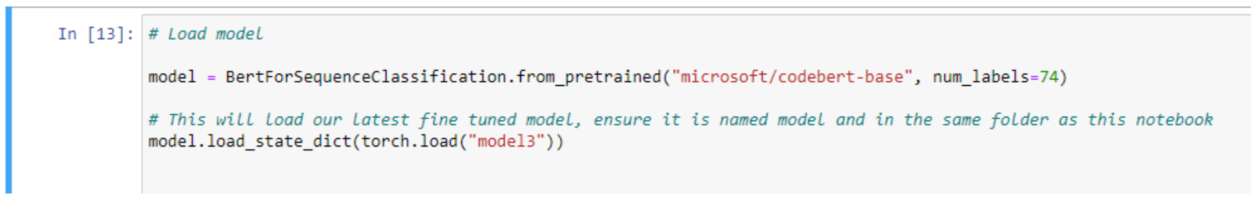
8.3 CodeBERT:

RobertaTokenizer: The RobertaTokenizer is the tokenizer designed for use with CodeBERT. This tokenizer will generate the necessary inputs that the trainer will need to train the model. This includes the input_ids and attention_mask attributes.



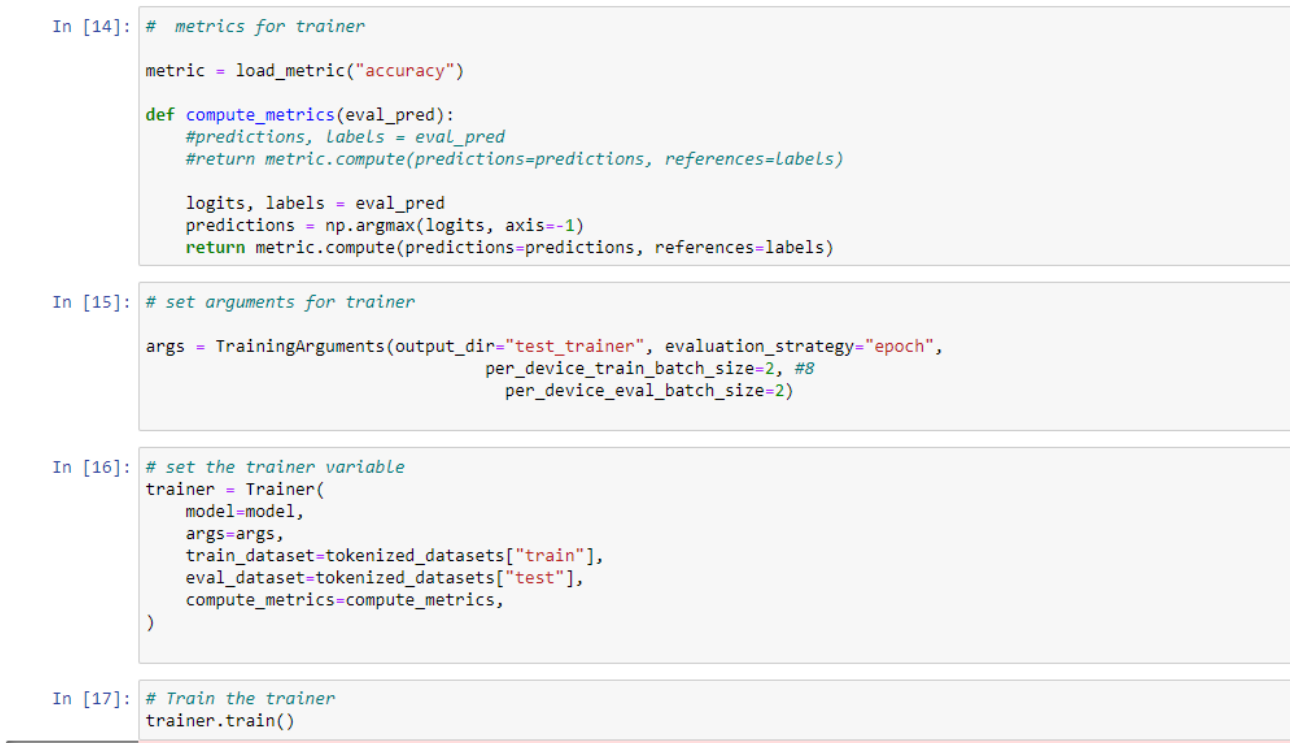
```
In [11]: # Auto tokenizer

         tokenizer = RobertaTokenizer.from_pretrained("microsoft/codebert-base")

         def tokenize_function(examples):
             return tokenizer(examples["text"], padding="max_length", truncation=True)


         tokenized_datasets = d.map(tokenize_function, batched=True)
```

Model: This is the code used for loading in your most recent model. When creating the model object we need to specify the number of labels we have. We have 74 tags, as shown below in our Final POS tags table.



```
In [13]: # Load model

         model = BertForSequenceClassification.from_pretrained("microsoft/codebert-base", num_labels=74)

         # This will load our latest fine tuned model, ensure it is named model and in the same folder as this notebook
         model.load_state_dict(torch.load("model3"))
```

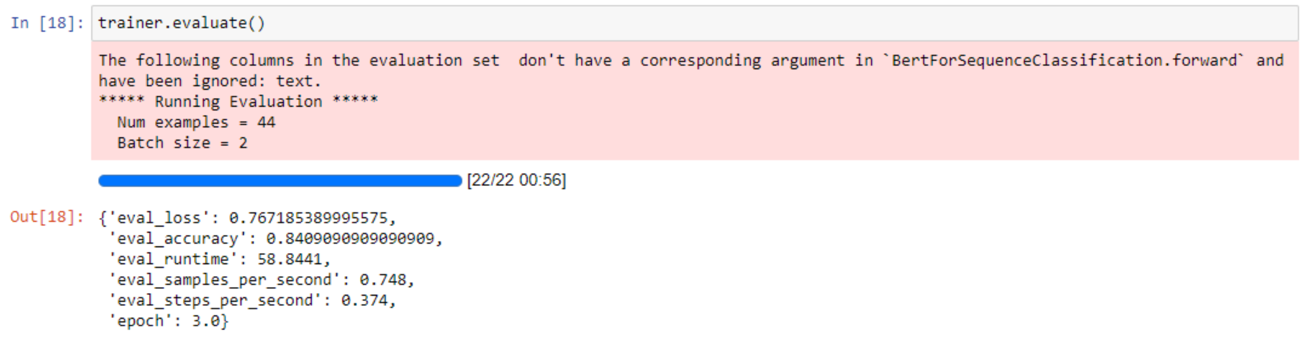
Trainer: Here you set up the trainer object and pass in the dataset and model created above.



```
In [14]: #  metrics for trainer

         metric = load_metric("accuracy")

         def compute_metrics(eval_pred):
             #predictions, labels = eval_pred
             #return metric.compute(predictions=predictions, references=labels)

             logits, labels = eval_pred
             predictions = np.argmax(logits, axis=-1)
             return metric.compute(predictions=predictions, references=labels)

In [15]: # set arguments for trainer

         args = TrainingArguments(output_dir="test_trainer", evaluation_strategy="epoch",
                                           per_device_train_batch_size=2, #8
                                             per_device_eval_batch_size=2)

In [16]: # set the trainer variable
         trainer = Trainer(
             model=model,
             args=args,
             train_dataset=tokenized_datasets["train"],
             eval_dataset=tokenized_datasets["test"],
             compute_metrics=compute_metrics,
         )

In [17]: # Train the trainer
         trainer.train()
```

8.4 Evaluation:

The trainer.evaluate() will give you the metrics for the test data that you passed into the trainer. We were able to get an evaluation accuracy of 84%



```
In [18]: trainer.evaluate()
The following columns in the evaluation set  don't have a corresponding argument in `BertForSequenceClassification.forward` and
have been ignored: text.
***** Running Evaluation *****
  Num examples = 44
  Batch size = 2
[22/22 00:56]
Out[18]: {'eval_loss': 0.767185389995575,
 'eval_accuracy': 0.8409090909090909,
 'eval_runtime': 58.8441,
 'eval_samples_per_second': 0.748,
 'eval_steps_per_second': 0.374,
 'epoch': 3.0}
```

8.5 Final POS tags

Tag	**ID**	**Example**
ENDMARKER	0	(EOF marker)
NAME	1	var_name
NUMBER	2	54
STRING	3	"your-string-here"
NEWLINE	4	\n
INDENT	5	\t (tab space)
DEDENT	6	(identifies when a new line has removed an indent compared to the previous line)
LPAR	7	(
RPAR	8	)
LSQB	9	[
RSQB	10	]
COLON	11	:

COMMA	12	,
SEMI	13	;
PLUS	14	+
MINUS	15	-
STAR	16	*
SLASH	17	/
VBAR	18	|
AMPER	19	&
LESS	20	<
GREATER	21	>
EQUAL	22	=
DOT	23	.
PERCENT	24	%
LBRACE	25	{
RBRACE	26	}
EQEQUAL	27	==
NOTEQUAL	28	!=
LESSEQUAL	29	<=
GREATEREQUAL	30	>=
TILDE	31	~
CIRCUMFLEX	32	^
LEFTSHIFT	33	<<
RIGHTSHIFT	34	>>
DOUBLESTAR	35	**
PLUSEQUAL	36	+=

MINEQUAL	37	-=
STAREQUAL	38	*=
SLASHEQUAL	39	/=
PERCENTEQUAL	40	%=
AMPEREQUAL	41	&=
VBAREQUAL	42	|=
CIRCUMFLEXEQUAL	43	^=
LEFTSHIFTEQUAL	44	<<=
RIGHTSHIFTEQUAL	45	>>=
DOUBLESTAREQUAL	46	**=
DOUBLESLASH	47	//
DOUBLESLASHEQUAL	48	//=
AT	49	@
ATEQUAL	50	@=
RARROW	51	->
ELLIPSIS	52	…
COLONEQUAL	53	:=
OP	54	>>
AWAIT	55	await
ASYNC	56	async
TYPE_IGNORE	57	type: ignore
TYPE_COMMENT	58	(identify type comment)
SOFT_KEYWORD	59	case
ERRORTOKEN	60	‘Single quote is not closed
COMMENT	61	# python comment line

NL	62	(used when a logical line of code is continued over multiple lines)
ENCODING	63	(indicates the encoding used to decode the source bytes) utf-8
N_TOKENS	64	Returns number of tokens
KEYWORD	65	(python keywords) and, break, class, True, etc.
PRINT	66	print(“Hello, World!”)
WHITESPACE	67	(catch-all for miscellaneous whitespace characters)
COND	68	if, elif, else
COND_BLOCK	69	if (){ … }
WHILE	70	while(...)
WHILE_BLOCK	71	while(...) {...}
FOR	72	for()
FOR_BLOCK	73	for(...){...}