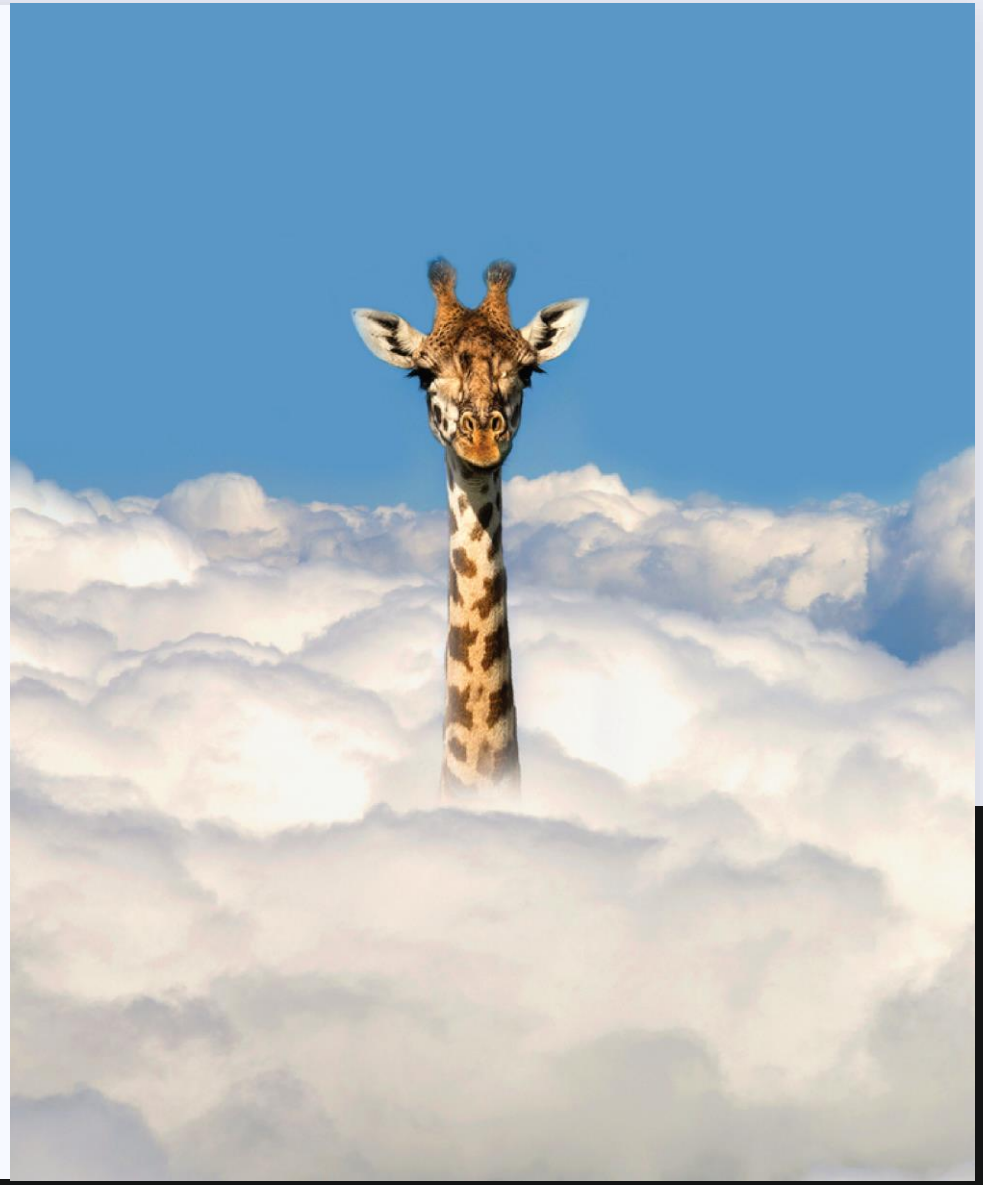
MULTILINGUAL NER

ASH GROUP



PROJECT STRUCTURE

MULTI LINGUAL NER

"Fine-tune XLM Roberta model to support English and Persian language named entity recognition.

MULTI LINGUAL NER

- *Given a sequence of tokens (words, and possibly punctuation marks), provide a tag from a predefined tag set for each token in the sequence.*

Tags :

```
['I-CW', 'I-CORP', 'B-PROD', 'I-PROD',
'I-GRP', 'O', 'B-LOC', 'B-GRP',
'B-CORP', 'B-CW', 'I-PER', 'B-PER',
'I-LOC']
```

PATH AND PARAMETERS

- If Debug == True then fine-tuning will be executed with limited partition of data specified with MAX_EXAMPLES.
- Other global parameters are well commented.

```
1 DEBUG = True # Debug mode: If True then training set will be limited to MAX_EXAMPLES
2 MAX_EXAMPLES = 1000 # Size of Debug Set
3 MAX_EPOCHS = 15 # Max Number of epoochs
4 CHECKPOINT_PATH = "./drive/MyDrive" # Root directory of checkpoints
5 MODEL_PATH = "./model" # Root directory of pytorch model
6 MODEL_NAME = "model.pt" # name of final pytorch model
```

DOWNLOAD DATASET

- Create data folder and download datasets

```
# Create data folder and donwload required datasets
! mkdir data
! wget https://github.com/language-ml/4-token-classification/raw/main/Multilingual-NER/en_test.csv -P ./data
! wget https://github.com/language-ml/4-token-classification/raw/main/Multilingual-NER/en_train.csv -P ./data
! wget https://github.com/language-ml/4-token-classification/raw/main/Multilingual-NER/fa_test.csv -P ./data
! wget https://github.com/language-ml/4-token-classification/raw/main/Multilingual-NER/fa_train.csv -P ./data
```

LIBRARY

List of necessary libraries.

```python
# import primitive libraries
import os
import pandas as pd
from tqdm import tqdm
import numpy as np
import json

# import seqval to report classifier performance metrics
from seqeval.metrics import accuracy_score, precision_score, recall_score, f1_score
from seqeval.scheme import IOB2

# import torch related modules
import torch
from torch.utils.data import DataLoader
from torch.utils.data import Dataset
from torch.nn.utils.rnn import pad_sequence
import torch.nn as nn

# import pytorch lightning library
import pytorch_lightning as pl
from torchcrf import CRF as SUPERCRF

# import NLTK to create better tokenizer
import nltk
from nltk.tokenize import RegexpTokenizer

# Transformers : Roberta Model
from transformers import XLMRobertaTokenizerFast
from transformers import XLMRobertaModel, XLMRobertaConfig

# import sklearn inorder to split data into train-evaluation-test
from sklearn.model_selection import GroupShuffleSplit

# import Typings
from typing import Union,Dict,List,Tuple,Any,Optional
```

Load Data

```python
# A function to load dataset into dataframe
def load_data(name, path='./data', test=False):
    print(f'Processing {name}')
    # Read CSV
    df = pd.read_csv(path + '/' + name)
    # Define columns
    df.columns = ['index', 'Token', 'Tag']
    df.set_index('index', drop=True, inplace=True)
    # Remove rows which starts with # id
    mask = df.Token.str.startswith('# id')
    indices = list(map(lambda x:x+1, df.index[mask].tolist())) + [df.index[-1]]
    # Assign Sent number to each row
    sentence = 0
    df['Sent'] = None
    for index in tqdm(range(len(indices)-1)):
        df.loc[indices[index]:indices[index+1], 'Sent'] = sentence
        sentence += 1
    df.drop(df.index[mask], inplace=True)
    # Drop Tag column if Dataframe is for test
    if test:
      df.drop(columns='Tag', inplace=True)
    return df

```

Load Data

```
# EN Data
en_data = load_data('en_train.csv')
# clean tag column
en_data['Tag'] = en_data['Tag'].apply(lambda tag: tag.strip() if tag!=' _ O' else 'O')
# clean Token column
en_data['Token'] = en_data['Token'].apply(lambda token: token.strip())

# EN Test (Deploy)
en_deploy_test = load_data('en_test.csv', test=True)

# FA Data
fa_data = load_data('fa_train.csv')
# clean tag column
fa_data['Tag'] = fa_data['Tag'].apply(lambda tag: tag.strip() if tag!=' _ O' else 'O')
# clean Token column
fa_data['Token'] = fa_data['Token'].apply(lambda token: token.strip())

# FA TEST (Desploy)
fa_deploy_test = load_data('fa_test.csv', test=True)
```

Preparing Data

- Preparing data
  - Train
  - Test
  - Evaluate

GroupShuffleSplit would proportionally split data based on ***VAL_TEST_SIZE*** and ***TEST_SIZE*** which are ratio of Val set + Test set to training set and Test set to Val set.

```
# Split Dataframe into Train and Val+Test
splitter = GroupShuffleSplit(test_size=VAL_TEST_SIZE, n_splits=1,  random_state = 42)
split = splitter.split(en_data, groups=en_data['Sent'])
en_train_inds, en_val_test_inds = next(split)

# En Train and Val+Test
en_train = en_data.iloc[en_train_inds]
en_val_test = en_data.iloc[en_val_test_inds]

# Split Dataframe into Val and Test
splitter = GroupShuffleSplit(test_size=TEST_SIZE, n_splits=1,  random_state = 42)
split = splitter.split(en_val_test, groups=en_val_test['Sent'])
en_val_inds, en_test_inds = next(split)

# En Val and Test
en_val = en_val_test.iloc[en_val_inds]
en_test = en_val_test.iloc[en_test_inds]
```

Preparing Data

- We need to create suitable training and test input files in order to feed into model.

```python
# Create proper input files
def write_file(en_df, fa_df, name, path='./data'):
    # Get the first sentence. The parameter will be updated during execution of function and show the last processed sentence number.
    last_sent = en_df.iloc[0].Sent

    with open(f'{path}/{name}.txt', 'a') as writer:
      # Write English (Token, Tag) pairs
        for index, row in tqdm(en_df.iterrows(), desc = f'Writing English {name} file'):
            token, tag, current_sent = row

            if (current_sent == last_sent):
                writer.write(f"{token}\t{tag}\n")
            else:
                last_sent = current_sent
                writer.write(f"\n")
                writer.write(f"{token}\t{tag}\n")

        # Write Farsi (Token, Tag) pairs
        last_row_df = fa_df.iloc[-1].Sent

        for index, row in tqdm(fa_df.iterrows(), desc = f'Writing Farsi {name} file'):
            token, tag, current_sent = row

            if current_sent!=last_row_df:
                if (current_sent == last_sent):
                    writer.write(f"{token}\t{tag}\n")

                else:
                    last_sent = current_sent
                    writer.write(f"\n")
                    writer.write(f"{token}\t{tag}\n")
            else:
                # Last Sentence of file without newline char
                writer.write(f"\n")
                writer.write(f"{token}\t{tag}")
```

Create Test File

- Write Individual Persian and English test files to test final model.

```python
def write_individual_test(df, name, path='./data'):
    # Get the first sentence. The parameter will be updated during execution c
    last_sent = df.iloc[0].Sent

    with open(f'{path}/{name}.txt', 'a') as writer:
        last_row_df = df.iloc[-1].Sent

        for index, row in tqdm(df.iterrows(), desc = f'Writing {name} file'):
            token, tag, current_sent = row

            if current_sent!=last_row_df:
                if (current_sent == last_sent):
                    writer.write(f"{token}\t{tag}\n")

                else:
                    last_sent = current_sent
                    writer.write(f"\n")
                    writer.write(f"{token}\t{tag}\n")
            else:
                # Last Sentence of file without newline char
                writer.write(f"\n")
                writer.write(f"{token}\t{tag}")
```

XLM Roberta

Following subsections show different parts of out XLM-Roberta model which is implemented with help of Pytorch and Pytorch Lightening.

Token From Subtoken

Pad, Reshape and finally convert to tensor each units and masks.

```python
class TokenFromSubtoken(torch.nn.Module):

    def forward(self, units: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:

        device = units.device
        nf_int = units.size()[-1]
        batch_size = units.size()[0]

        # number of TOKENS in each sentence
        token_seq_lengths = torch.sum(mask, 1).to(torch.int64)
        # number of words
        n_words = torch.sum(token_seq_lengths)
        # max token seq len
        max_token_seq_len = torch.max(token_seq_lengths)

        idxs = torch.stack(torch.nonzero(mask, as_tuple=True), dim=1)
        # padding is for computing change from one sample to another in the batch
        sample_ids_in_batch = torch.nn.functional.pad(input=idxs[:, 0], pad=[1, 0])

        a = (~torch.eq(sample_ids_in_batch[1:], sample_ids_in_batch[:-1])).to(torch.int64)

        # transforming sample start masks to the sample starts themselves
        q = a * torch.arange(n_words, device=device).to(torch.int64)
        count_to_substract = torch.nn.functional.pad(torch.masked_select(q, q.to(torch.bool)), [1, 0])

        new_word_indices = torch.arange(n_words, device=device).to(torch.int64) - count_to_substract[torch.cumsum(a, 0)]

        n_total_word_elements = max_token_seq_len*torch.ones_like(token_seq_lengths, device=device).sum()
        word_indices_flat = (idxs[:, 0] * max_token_seq_len + new_word_indices).to(torch.int64)
        #x_mask = torch.sum(torch.nn.functional.one_hot(word_indices_flat, n_total_word_elements), 0)
        #x_mask = x_mask.to(torch.bool)
        x_mask = torch.zeros(n_total_word_elements, dtype=torch.bool, device=device)
        x_mask[word_indices_flat] = torch.ones_like(word_indices_flat, device=device, dtype=torch.bool)
        # to get absolute indices we add max_token_seq_len:

        nonword_indices_flat = (~x_mask).nonzero().squeeze(-1)

        elements = units[mask.bool()]

        paddings = torch.zeros_like(nonword_indices_flat, dtype=elements.dtype).unsqueeze(-1).repeat(1,nf_int).to(device)

        # tensor_flat -> [x, x, x, x, x, 0, x, 0, 0]
        tensor_flat_unordered = torch.cat([elements, paddings])
        _, order_idx = torch.sort(torch.cat([word_indices_flat, nonword_indices_flat]))
        tensor_flat = tensor_flat_unordered[order_idx]

        tensor = torch.reshape(tensor_flat, (-1, max_token_seq_len, nf_int))

        return tensor
```

CRF

- CRF Layer implemented with help of **TorchCRF** library, but we need to override **_Viterbi_decode** to make it more compatible with our model.

```python
class CRF(SUPERCRF):

    # override veiterbi decoder in order to make it compatible with our code
    def _viterbi_decode(self, emissions: torch.FloatTensor,
                        mask: torch.ByteTensor) -> List[List[int]]:
        # emissions: (seq_length, batch_size, num_tags)
        # mask: (seq_length, batch_size)
        assert emissions.dim() == 3 and mask.dim() == 2
        assert emissions.shape[:2] == mask.shape
        assert emissions.size(2) == self.num_tags
        assert mask[0].all()

        seq_length, batch_size = mask.shape

        # Start transition and first emission
        # shape: (batch_size, num_tags)
        score = self.start_transitions + emissions[0]
        history = []


        for i in range(1, seq_length):
            # Broadcast viterbi score for every possible next tag
            # shape: (batch_size, num_tags, 1)
            broadcast_score = score.unsqueeze(2)

            # Broadcast emission score for every possible current tag
            # shape: (batch_size, 1, num_tags)
            broadcast_emission = emissions[i].unsqueeze(1)

            # shape: (batch_size, num_tags, num_tags)
            next_score = broadcast_score + self.transitions + broadcast_emission

            # shape: (batch_size, num_tags)
            next_score, indices = next_score.max(dim=1)


            # shape: (batch_size, num_tags)
            score = torch.where(mask[i].unsqueeze(1), next_score, score)
            history.append(indices)
```

CRF Layer

- Forward
  - Decide output logits based on backbone network
- Backward
  - decode based on CRF weig -hts

```python
class CRFLayer(nn.Module):
    def __init__(self, embedding_size, n_labels):

        super(CRFLayer, self).__init__()
        self.dropout = nn.Dropout(0.1)
        self.output_dense = nn.Linear(embedding_size,n_labels)
        self.crf = CRF(n_labels, batch_first=True)
        self.token_from_subtoken = TokenFromSubtoken()

    # Forward: decide output logits basaed on backbone network
    def forward(self, embedding, mask):
        logits = self.output_dense(self.dropout(embedding))
        logits = self.token_from_subtoken(logits, mask)
        pad_mask = self.token_from_subtoken(mask.unsqueeze(-1), mask).squeeze(-1).bool()
        return logits, pad_mask

    # Decode: decode based on CRF weights
    def decode(self, logits, pad_mask):
        return self.crf.decode(logits, pad_mask)

    # Evaluation Loss: calculate mean log likelihood of CRF layer
    def eval_loss(self, logits, targets, pad_mask):
        mean_log_likelihood = self.crf(logits, targets, pad_mask, reduction='sum').mean()
        return -mean_log_likelihood

```

NER Model

Following class include pretrained ***XLMRobertaModel*** and implements basic necessary methods like predict, decode, loss calculation and freezing specific number of layers.

```python
class NERModel(nn.Module):

    def __init__(self, n_labels:int, roberta_path:str):
        super(NERModel,self).__init__()
        self.roberta = XLMRobertaModel.from_pretrained(roberta_path)
        self.crf = CRFLayer(self.roberta.config.hidden_size, n_labels)

    # Forward: pass embedings to CRF layer in order to evaluate logits from suboword sequence
    def forward(self,
                input_ids:torch.Tensor,
                attention_mask:torch.Tensor,
                token_type_ids:torch.Tensor,
                mask:torch.Tensor) -> torch.Tensor:

        embedding = self.roberta(input_ids=input_ids,
                                 attention_mask=attention_mask,
                                 token_type_ids=token_type_ids)[0]
        logits, pad_mask = self.crf(embedding, mask)
        return logits, pad_mask

    # Disable Gradient and Predict with model
    @torch.no_grad()
    def predict(self, inputs:Tuple[torch.Tensor]) -> torch.Tensor:
        input_ids, attention_mask, token_type_ids, mask = inputs
        logits, pad_mask = self(input_ids, attention_mask, token_type_ids, mask)
        decoded = self.crf.decode(logits, pad_mask)
        return decoded, pad_mask

    # Decode: pass to crf decoder and decode based on CRF weights
    def decode(self, logits, pad_mask):
        """Decode logits using CRF weights
        """
        return self.crf.decode(logits, pad_mask)

    # Evaluation Loss: pass to crf eval_loss and calculate mean log likelihood of CRF layer
    def eval_loss(self, logits, targets, pad_mask):
        return self.crf.eval_loss(logits, targets, pad_mask)

    # Determine number of layers to be fine-tuned (!freeze)
    def freeze_roberta(self, n_freeze:int=6):
        for param in self.roberta.parameters():
            param.requires_grad = False

        for param in self.roberta.encoder.layer[n_freeze:].parameters():
            param.requires_grad = True
```

NER Tokenizer

Following class inherits benefits of ***NLTK*** tokenizer and ***XLMRobertaTokenizer*** in order to create better tokenizer which is responsible to tokenize input batches.

```python
class NERTokenizer(object):

    MAX_LEN=512
    BATCH_LENGTH_LIMT = 380 # Max number of roberta tokens in one sentence.

    # Modified version of http://stackoverflow.com/questions/36353125/nltk-regular-expression-tokenizer
    PATTERN = r'''(?x)          # set flag to allow verbose regexps
        (?:[A-Z]\.)+        # abbreviations, e.g. U.S.A. or U.S.A #
      | (?:\d+\.)           # numbers
      | \w+(?:[-.]\w+)*     # words with optional internal hyphens
      | \$?\d+(?:.\d+)?%?   # currency and percentages, e.g. $12.40, 82%
      | \.\.\.              # ellipsis, and special chars below, includes ], [
      | [-\]\[.,;"'?():_`“”/°º‘’"…#$%()*+<>=@\\^_{}|~❑&§]
    '''

    def __init__(self, base_model:str, to_device:str='cpu'):
        super(NERTokenizer,self).__init__()
        self.roberta_tokenizer = XLMRobertaTokenizerFast.from_pretrained(base_model, do_lower_case=False)
        self.to_device = to_device

        self.word_tokenizer = RegexpTokenizer(self.PATTERN)
        self.sent_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')

    # tokenize batch of tokens
    def tokenize_batch(self, inputs, pad_to = None) -> torch.Tensor:
        batch = [inputs] if isinstance(inputs[0], str) else inputs

        input_ids, attention_mask, token_type_ids, mask = [], [], [], []
        for tokens in batch:
            input_ids_tmp, attention_mask_tmp, token_type_ids_tmp, mask_tmp = self._tokenize_words(tokens)
            input_ids.append(input_ids_tmp)
            attention_mask.append(attention_mask_tmp)
            token_type_ids.append(token_type_ids_tmp)
            mask.append(mask_tmp)

        input_ids = pad_sequence(input_ids, batch_first=True, padding_value=self.roberta_tokenizer.pad_token_id)
        attention_mask = pad_sequence(attention_mask, batch_first=True, padding_value=0)
        token_type_ids = pad_sequence(token_type_ids, batch_first=True, padding_value=0)
        mask = pad_sequence(mask, batch_first=True, padding_value=0)
```

NER Dataset

Any Pytorch Model Fine-tuning needs a good Pytorch dataset with ***__len__*** and ***__getitem__*** conventional methods.

```python
# Pytorch Dataset
class NERDataset(Dataset):

    def __init__(self, data_path:str, label_tags: List[str], base_model:str="xlm-roberta-base",
            default_label=0, max_length:int=512, to_device="cpu"):
        self.tokenizer = NERTokenizer(base_model=base_model, to_device=to_device)
        self.label_tags = label_tags
        self.name_to_label = {x: i for i, x in enumerate(self.label_tags)}
        self.default_label = default_label
        self.max_length = max_length


        # open file (train, test or val)
        with open(data_path,'r') as f:
            data_text = f.read()
        self.data = []

        # the loop notices the change of sentence with double newline
        for sentence in filter(lambda x: len(x)>2, data_text.split('\n\n')):
            sample = []
            # each word laid in sepertaed lines
            for wordline in sentence.split('\n'):
                if wordline=='':
                    continue
                # the word and label are seperated from each other with tab
                word, label = wordline.split('\t')
                sample.append((word, label))
            self.data.append(sample)

    # len of dataset
    def __len__(self):
        return len(self.data)


    def __getitem__(self, idx):
        item = self.data[idx]
        words, labels = list(zip(*item))

        labels_idx = [self.name_to_label.get(x, self.default_label) for x in labels]
        y = torch.tensor(labels_idx, dtype=torch.long)
        diff = self.max_length - y.shape[-1]
        y = torch.nn.functional.pad(y, (0, diff), value=self.default_label)
        X = self.tokenizer.tokenize_batch(list(words), pad_to=self.max_length)

        return X, y
```

NER Wrapper

Following Class inherits from Pytorch Lightening module class which will help us to pass this class to the **trainer** object and train model with predefined methods in this class.

```python
# Lightining Wrapper for NER Model

class NERWrapper(pl.LightningModule):
    def __init__(self,
        learning_rate = 2e-5,
        weight_decay = 0.0,
        batch_size = 16,
        freeze_layers = 8,
        tags = tags_table,
        train_path = "./data/train.txt",
        val_path = "./data/val.txt",
        test_path ="./data/test.txt",
        pretrained_path = None,
        *args, **kwargs
    ):

        super(NERWrapper,self).__init__()
        self.save_hyperparameters('learning_rate', 'weight_decay', 'batch_size')
        self.tags, self.train_path, self.val_path, self.test_path = tags, train_path, val_path, test_path
        self.model = NERModel(n_labels=len(self.tags), roberta_path="xlm-roberta-base")

        if pretrained_path is not None:
            self.model.load_state_dict(torch.load(pretrained_path))
        self.model.freeze_roberta(freeze_layers)

    def forward(self, *args, **kwargs):
        return self.model.forward(*args, **kwargs)

    def _step(self, batch, batch_idx):
        (input_ids, attention_mask, token_type_ids, mask), labels = batch
        logits, pad_mask = self.model(input_ids, attention_mask, token_type_ids, mask)
        labels = labels[:, :logits.shape[1]]
        loss = self.model.eval_loss(logits, labels, pad_mask)
        preds_tag_idx = self.model.decode(logits, pad_mask)
        preds_tag = [[self.tags[start.item()] for m, start in zip(mask, sample) if m] for mask, sample in zip(pad_mask, preds_tag_idx)]
        labels_tag = [[self.tags[start.item()] for m, start in zip(mask, sample) if m] for mask, sample in zip(pad_mask, labels)]
        tensorboard_logs = {'batch_loss': loss}
        for metric, value in tensorboard_logs.items():
            self.log(metric, value, prog_bar=True)
        return {'loss': loss, "preds": preds_tag, "labels": labels_tag}
```

NER Interface

You can use this interface and pass a raw string and receive NER tags in response.

```python
class NER(object):

    def __init__(self, model_path, model_name = MODEL_NAME, tags = tags_table):

        self.tags = tags
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
        roberta_path = "xlm-roberta-base"
        self.model = NERModel(n_labels=len(self.tags), roberta_path=roberta_path).to(self.device)
        state_dict = torch.load(os.path.join(model_path, model_name))
        self.model.load_state_dict(state_dict, strict=False)
        self.model.eval()
        self.tokenizer = NERTokenizer(base_model=roberta_path, to_device=self.device)

    @torch.no_grad()
    def __call__(self, raw_text):

        outputs_flat, spans_flat, entities = [], [], []
        for batch, words, spans in self.tokenizer.prepare_row_text(raw_text):
            output, pad_mask = self.model.predict(batch)
            outputs_flat.extend(output[pad_mask.bool()].reshape(-1).tolist())
            spans_flat += sum(spans, [])

        for tag_idx,(start,end) in zip(outputs_flat,spans_flat):
            tag = self.tags[tag_idx]
            if tag != 'O':
                entities.append({'Text': raw_text[start:end], 'Tag': tag})

        return entities
```

Train and Test Model

We train model with training dataset and we use validation test on predefined steps and finally output model will test with test datasets and result will be displayed

Train + Test

Following method will train and test over whole data.

```
  | Name  | Type     | Params
-----------------------------------
0 | model | NERModel | 278 M
-----------------------------------
28.4 M    Trainable params
249 M     Non-trainable params
278 M     Total params
```

```python
# train function
def train_test(test_path):
    plmodel = NERWrapper(test_path = test_path)
    trainer = pl.Trainer(default_root_dir=CHECKPOINT_PATH, gpus=1, max_epochs=MAX_EPOCHS)
    trainer.fit(plmodel)
    result = trainer.test(plmodel)
    print(result)
```

NER Interface

Here is example of how you can use NER interface which has been explained before.

```
# del ner
ner = NER(model_path = MODEL_PATH, model_name = MODEL_NAME, tags = tags_table) # Load pretrained model.
```

```
text = """his playlist includes sonny sharrock , gza , country teasers and the notorious b.i.g."""
result = ner(text)
result
```

EVALUATION

Here are final evaluation result of our model after hours and hours of sitting behind the monitor proving google colab that we are not robots.

```
DATALOADER:0 TEST RESULTS
{'batch_loss': 24.60813919067383,
 'test_F1': 0.7578543,
 'test_accuracy': 0.9484375,
 'test_loss': 60.608143005371094,
 'test_precision': 0.772465,
 'test_recall': 0.7423492}
```