Twitter

November 23, 2023

[1]:
```
import pandas as pd
import numpy as np
from pathlib import Path
import re

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import accuracy_score
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertConfig, BertForTokenClassification

torch.manual_seed(1)
```

[1]:
```
<torch._C.Generator at 0x2273291c210>
```

[2]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

[3]:
```
def read_wnut(file_path):
    file_path = Path(file_path)

    raw_text = file_path.read_text().strip()
    raw_docs = re.split(r'\n\t?\n', raw_text)
    token_docs = []
    tag_docs = []
    for doc in raw_docs:
        tokens = []
        tags = []
        for line in doc.split('\n'):
            token, tag = line.split('\t')
            tokens.append(token)
            tags.append(tag)
        token_docs.append(tokens)
        tag_docs.append(tags)

    return token_docs, tag_docs
```

```
def get_dataframe(text,tags):
    words = []
    target = []
    for sentence, tag in zip(text,tags):
        for word in range(len(sentence)):
            words.append(sentence[word])
            target.append(tag[word])
    data = pd.DataFrame(data={"Words":words,"Tags":target})
    return data
```

[4]:
```
text, tag = read_wnut("wnut 16.txt")
```

[5]:
```
for text1, tags in zip(text,tag):
    print(text1,tags)
    break
```

```
['@SammieLynnsMom', '@tg10781', 'they', 'will', 'be', 'all', 'done', 'by',
'Sunday', 'trust', 'me', '*wink*'] ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O']
```

[6]:
```
train_text, train_tags = read_wnut("wnut 16.txt")
test_text, test_tags = read_wnut("wnut 16test.txt")
train_data = get_dataframe(train_text,train_tags)
test_data = get_dataframe(test_text,test_tags)

train_data.to_csv("train.tsv",sep="\t",index=False,header=True)
test_data.to_csv("test.tsv",sep="\t",index=False,header=True)
```

[7]:
```
train_data
```

[7]:
```
                 Words Tags
0       @SammieLynnsMom    O
1             @tg10781    O
2                 they    O
3                 will    O
4                   be    O
…                  …   …
46464          whatchu    O
46465              got    O
46466              for    O
46467               me    O
46468       @kanyewest    O

[46469 rows x 2 columns]
```

[8]:
```
words = list(set(train_data["Words"].values))
n_words = len(words)
print("Number of unique words in the dataset: ", n_words)
```

```
word2idx = {w: i for i, w in enumerate(words)}

tags = list(set(train_data["Tags"].values))
label2idx = {t: i for i, t in enumerate(tags)}
START_TAG = "<START>"
STOP_TAG = "<STOP>"
label2idx[START_TAG] = 21
label2idx[STOP_TAG] = 22
```

```
Number of unique words in the dataset:  10586
```

[13]:
```
def argmax(vec):
    # return the argmax as a python int
    _, idx = torch.max(vec, 1)
    return idx.item()


def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    return torch.tensor(idxs, dtype=torch.long)


# Compute log sum exp in a numerically stable way for the forward algorithm
def log_sum_exp(vec):
    max_score = vec[0, argmax(vec)]
    max_score_broadcast = max_score.view(1, -1).expand(1, vec.size()[1])
    return max_score + \
        torch.log(torch.sum(torch.exp(vec - max_score_broadcast)))
```

[14]:
```
class BiLSTM_CRF(nn.Module):

    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
        super(BiLSTM_CRF, self).__init__()
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tag_to_ix = tag_to_ix
        self.tagset_size = len(tag_to_ix)

        self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                            num_layers=1, bidirectional=True)

        # Maps the output of the LSTM into tag space.
        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size)

        # Matrix of transition parameters.  Entry i,j is the score of
        # transitioning *to* i *from* j.
```

```python
        self.transitions = nn.Parameter(
            torch.randn(self.tagset_size, self.tagset_size))

        # These two statements enforce the constraint that we never transfer
        # to the start tag and we never transfer from the stop tag
        self.transitions.data[tag_to_ix[START_TAG], :] = -10000
        self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000

        self.hidden = self.init_hidden()

    def init_hidden(self):
        return (torch.randn(2, 1, self.hidden_dim // 2),
                torch.randn(2, 1, self.hidden_dim // 2))

    def _forward_alg(self, feats):
        # Do the forward algorithm to compute the partition function
        init_alphas = torch.full((1, self.tagset_size), -10000.)
        # START_TAG has all of the score.
        init_alphas[0][self.tag_to_ix[START_TAG]] = 0.

        # Wrap in a variable so that we will get automatic backprop
        forward_var = init_alphas

        # Iterate through the sentence
        for feat in feats:
            alphas_t = []  # The forward tensors at this timestep
            for next_tag in range(self.tagset_size):
                # broadcast the emission score: it is the same regardless of
                # the previous tag
                emit_score = feat[next_tag].view(
                    1, -1).expand(1, self.tagset_size)
                # the ith entry of trans_score is the score of transitioning to
                # next_tag from i
                trans_score = self.transitions[next_tag].view(1, -1)
                # The ith entry of next_tag_var is the value for the
                # edge (i -> next_tag) before we do log-sum-exp
                next_tag_var = forward_var + trans_score + emit_score
                # The forward variable for this tag is log-sum-exp of all the
                # scores.
                alphas_t.append(log_sum_exp(next_tag_var).view(1))
            forward_var = torch.cat(alphas_t).view(1, -1)
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        alpha = log_sum_exp(terminal_var)
        return alpha

    def _get_lstm_features(self, sentence):
        self.hidden = self.init_hidden()
        embeds = self.word_embeds(sentence).view(len(sentence), 1, -1)
        lstm_out, self.hidden = self.lstm(embeds, self.hidden)
        lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
        lstm_feats = self.hidden2tag(lstm_out)
        return lstm_feats

    def _score_sentence(self, feats, tags):
        # Gives the score of a provided tag sequence
        score = torch.zeros(1)
        tags = torch.cat([torch.tensor([self.tag_to_ix[START_TAG]], dtype=torch.
↪long), tags])
        for i, feat in enumerate(feats):
            score = score + \
                self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]
        score = score + self.transitions[self.tag_to_ix[STOP_TAG], tags[-1]]
        return score

    def _viterbi_decode(self, feats):
        backpointers = []

        # Initialize the viterbi variables in log space
        init_vvars = torch.full((1, self.tagset_size), -10000.)
        init_vvars[0][self.tag_to_ix[START_TAG]] = 0

        # forward_var at step i holds the viterbi variables for step i-1
        forward_var = init_vvars
        for feat in feats:
            bptrs_t = []  # holds the backpointers for this step
            viterbivars_t = []  # holds the viterbi variables for this step

            for next_tag in range(self.tagset_size):
                # next_tag_var[i] holds the viterbi variable for tag i at the
                # previous step, plus the score of transitioning
                # from tag i to next_tag.
                # We don't include the emission scores here because the max
                # does not depend on them (we add them in below)
                next_tag_var = forward_var + self.transitions[next_tag]
                best_tag_id = argmax(next_tag_var)
                bptrs_t.append(best_tag_id)
                viterbivars_t.append(next_tag_var[0][best_tag_id].view(1))
            # Now add in the emission scores, and assign forward_var to the set
            # of viterbi variables we just computed
            forward_var = (torch.cat(viterbivars_t) + feat).view(1, -1)
            backpointers.append(bptrs_t)

        # Transition to STOP_TAG
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        best_tag_id = argmax(terminal_var)
        path_score = terminal_var[0][best_tag_id]

        # Follow the back pointers to decode the best path.
        best_path = [best_tag_id]
        for bptrs_t in reversed(backpointers):
            best_tag_id = bptrs_t[best_tag_id]
            best_path.append(best_tag_id)
        # Pop off the start tag (we dont want to return that to the caller)
        start = best_path.pop()
        assert start == self.tag_to_ix[START_TAG]  # Sanity check
        best_path.reverse()
        return path_score, best_path

    def neg_log_likelihood(self, sentence, tags):
        feats = self._get_lstm_features(sentence)
        forward_score = self._forward_alg(feats)
        gold_score = self._score_sentence(feats, tags)
        return forward_score - gold_score

    def forward(self, sentence):  # dont confuse this with _forward_alg above.
        # Get the emission scores from the BiLSTM
        lstm_feats = self._get_lstm_features(sentence)

        # Find the best path, given the features.
        score, tag_seq = self._viterbi_decode(lstm_feats)
        return score, tag_seq
```

[16]:
```
START_TAG = "<START>"
STOP_TAG = "<STOP>"
EMBEDDING_DIM = 1500
HIDDEN_DIM = 1500



model = BiLSTM_CRF(len(test_word2idx), label2idx, EMBEDDING_DIM, HIDDEN_DIM)
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-4)




# Make sure prepare_sequence from earlier in the LSTM section is loaded
for epoch in range(10):  # again, normally you would NOT do 300 epochs, it is␣
 ↪toy data
    losses = []
    i = 0
    for sentence, tags in zip(train_text, train_tags):
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Get our inputs ready for the network, that is,
        # turn them into Tensors of word indices.
        sentence_in = prepare_sequence(sentence, word2idx)
        targets = torch.tensor([label2idx[t] for t in tags])

        # Step 3. Run our forward pass.
        loss = model.neg_log_likelihood(sentence_in, targets)

        # Step 4. Compute the loss, gradients, and update the parameters by
        # calling optimizer.step()
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
        if i%1000 == 0:
            print(f"Iteration{i}th and the loss {loss}")
        i+=1
    print(torch.tensor(losses).mean())

# Check predictions after training
with torch.no_grad():
    precheck_sent = prepare_sequence(train_text[0], word2idx)
    print(model(precheck_sent))
# We got it!
```

```
Iteration0th and the loss tensor([57.1450], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([2.0627], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.2528], grad_fn=<SubBackward0>)
tensor(5.8269)
Iteration0th and the loss tensor([2.2906], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.2193], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.0815], grad_fn=<SubBackward0>)
tensor(2.0242)
Iteration0th and the loss tensor([0.1412], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.2069], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.0155], grad_fn=<SubBackward0>)
tensor(0.6142)
Iteration0th and the loss tensor([0.2774], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.0683], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.0090], grad_fn=<SubBackward0>)
tensor(0.2749)
Iteration0th and the loss tensor([0.0433], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.0195], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.0034], grad_fn=<SubBackward0>)
tensor(0.1818)
Iteration0th and the loss tensor([0.0272], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.0181], grad_fn=<SubBackward0>)
```

```
Iteration2000th and the loss tensor([0.0030], grad_fn=<SubBackward0>)
tensor(0.1415)
Iteration0th and the loss tensor([0.0131], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.0171], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.0025], grad_fn=<SubBackward0>)
tensor(0.1258)
Iteration0th and the loss tensor([0.0095], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.0187], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.0019], grad_fn=<SubBackward0>)
tensor(0.1129)
Iteration0th and the loss tensor([0.0197], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.0139], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.0014], grad_fn=<SubBackward0>)
tensor(0.1041)
Iteration0th and the loss tensor([0.0137], grad_fn=<SubBackward0>)
Iteration1000th and the loss tensor([0.0103], grad_fn=<SubBackward0>)
Iteration2000th and the loss tensor([0.0016], grad_fn=<SubBackward0>)
tensor(0.0924)
(tensor(154.5079), [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[17]:
```python
with torch.no_grad():
    precheck_sent = prepare_sequence(train_text[0], word2idx)
    print(model(precheck_sent))
```

```
(tensor(154.7276), [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[18]:
```python
print("Model's state_dict:")
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())

# Print optimizer's state_dict
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])
```

```
Model's state_dict:
transitions      torch.Size([23, 23])
word_embeds.weight       torch.Size([18320, 1500])
lstm.weight_ih_l0        torch.Size([3000, 1500])
lstm.weight_hh_l0        torch.Size([3000, 750])
lstm.bias_ih_l0          torch.Size([3000])
lstm.bias_hh_l0          torch.Size([3000])
lstm.weight_ih_l0_reverse        torch.Size([3000, 1500])
lstm.weight_hh_l0_reverse        torch.Size([3000, 750])
lstm.bias_ih_l0_reverse          torch.Size([3000])
lstm.bias_hh_l0_reverse          torch.Size([3000])
hidden2tag.weight        torch.Size([23, 1500])
hidden2tag.bias          torch.Size([23])
Optimizer's state_dict:
```

```
state    {0: {'momentum_buffer': None}, 1: {'momentum_buffer': None}, 2:
{'momentum_buffer': None}, 3: {'momentum_buffer': None}, 4: {'momentum_buffer':
None}, 5: {'momentum_buffer': None}, 6: {'momentum_buffer': None}, 7:
{'momentum_buffer': None}, 8: {'momentum_buffer': None}, 9: {'momentum_buffer':
None}, 10: {'momentum_buffer': None}, 11: {'momentum_buffer': None}}
param_groups     [{'lr': 0.01, 'momentum': 0, 'dampening': 0, 'weight_decay':
0.0001, 'nesterov': False, 'maximize': False, 'foreach': None, 'differentiable':
False, 'params': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]}]
```

[19]:
```
torch.save(model.state_dict(), "LSTM_CRF1.th")
```

[20]:
```
words = list(set(test_data["Words"].values))
n_words = len(words)
print("Number of unique words in the dataset: ", n_words)
test_word2idx = {w: i for i, w in enumerate(words)}

tags = list(set(test_data["Tags"].values))
test_label2idx = {t: i for i, t in enumerate(tags)}
START_TAG = "<START>"
STOP_TAG = "<STOP>"
test_label2idx[START_TAG] = 21
test_label2idx[STOP_TAG] = 22
```

```
Number of unique words in the dataset:  18320
```

[23]:
```
EMBEDDING_DIM = 1500
HIDDEN_DIM = 1500
model = BiLSTM_CRF(len(test_word2idx), label2idx, EMBEDDING_DIM, HIDDEN_DIM)
model.load_state_dict(torch.load("LSTM_CRF1.th"))
model.eval()
```

[23]:
```
BiLSTM_CRF(
  (word_embeds): Embedding(18320, 1500)
  (lstm): LSTM(1500, 750, bidirectional=True)
  (hidden2tag): Linear(in_features=1500, out_features=23, bias=True)
)
```

[42]:
```
with torch.no_grad():
    accuracy = 0
    for text, tags in zip(test_text,test_tags):
        postcheck_sent = torch.tensor([test_word2idx[w] for w in text])
        postheck_tags = torch.tensor([test_label2idx[t] for t in tags],␣
 ↪dtype=torch.long)
        pred = model(postcheck_sent)
        accuracy+=(postheck_tags == torch.tensor(pred[1])).sum()/
 ↪len(postheck_tags)
    print(accuracy/len(test_text))
```

```
tensor(0.8740)
```

```
[99]: data = pd.read_csv("train.csv", encoding='unicode_escape',index_col=[0])
      data.head()
```

```
[99]:            Words Tags
      0  @SammieLynnsMom    O
      1         @tg10781    O
      2             they    O
      3             will    O
      4               be    O
```

```
[100]: print("Number of tags: {}".format(len(data.Tags.unique())))
       frequencies = data.Tags.value_counts()
       frequencies
```

```
Number of tags: 21
```

```
[100]: Tags
       O                44007
       B-person           449
       I-other            320
       B-geo-loc          276
       B-other            225
       I-person           215
       B-company          171
       I-facility         105
       B-facility         104
       B-product           97
       I-product           80
       I-musicartist       61
       B-musicartist       55
       B-sportsteam        51
       I-geo-loc           49
       I-movie             46
       I-company           36
       B-movie             34
       B-tvshow            34
       I-tvshow            31
       I-sportsteam        23
       Name: count, dtype: int64
```

```
[101]: label2id = {k: v for v, k in enumerate(data.Tags.unique())}
       id2label = {v: k for v, k in enumerate(data.Tags.unique())}
       label2id
```

```
[101]: {'O': 0,
        'B-geo-loc': 1,
        'B-facility': 2,
        'I-facility': 3,
        'B-movie': 4,
        'I-movie': 5,
        'B-company': 6,
        'B-product': 7,
        'B-person': 8,
        'B-other': 9,
        'I-other': 10,
        'B-sportsteam': 11,
        'I-sportsteam': 12,
        'I-product': 13,
        'I-company': 14,
        'I-person': 15,
        'I-geo-loc': 16,
        'B-tvshow': 17,
        'B-musicartist': 18,
        'I-musicartist': 19,
        'I-tvshow': 20}
```

[102]:
```
tags = {}
for tag, count in zip(frequencies.index, frequencies):
    if tag != "O":
        if tag not in tags.keys():
            tags[tag] = count
        else:
            tags[tag] += count
    continue

print(sorted(tags.items(), key=lambda x: x[1], reverse=True))
```

```
[('B-person', 449), ('I-other', 320), ('B-geo-loc', 276), ('B-other', 225),
('I-person', 215), ('B-company', 171), ('I-facility', 105), ('B-facility', 104),
('B-product', 97), ('I-product', 80), ('I-musicartist', 61), ('B-musicartist',
55), ('B-sportsteam', 51), ('I-geo-loc', 49), ('I-movie', 46), ('I-company',
36), ('B-movie', 34), ('B-tvshow', 34), ('I-tvshow', 31), ('I-sportsteam', 23)]
```

[103]:
```
sentences = []
for text in train_text:
    sentence = ""
    for i in text:
        sentence+=i+" "
    sentences.append(sentence)
total_tags = []
for tags in train_tags:
    tag = ""
    for i in tags:
        tag+=i + ","
    total_tags.append(tag[:-1])
```

[104]:
```
data["sentence"] = pd.Series(sentences)
data["label"] = pd.Series(total_tags)
```

[105]:
```
data.drop(["Words","Tags"],inplace=True,axis=1)
data.dropna(inplace=True)
```

[106]:
```
data
```

[106]:
```
                                                  sentence  \
0     @SammieLynnsMom @tg10781 they will be all done…
1     Made it back home to GA . It sucks not to be a…
2     ' Breaking Dawn ' Returns to Vancouver on Janu…
3     @ls_n perhaps , but folks may find something i…
4                 @Carr0t aye been tonight - excellent
…                                                  …
2389  RT @MarioBB9 : Pope says atheists pick and cho…
2390  Man I swear I bought 2 new outfits but it 's c…
2391  RT @ArtVanFurniture : Mr . Van sure is busy to…
2392  @PersonalSelena can you follow me pretty pleas…
2393         good friday whatchu got for me @kanyewest

                                                     label
0                                0,0,0,0,0,0,0,0,0,0,0,0
1     0,0,0,0,0,B-geo-loc,0,0,0,0,0,0,0,B-facility,I…
2             0,B-movie,I-movie,0,0,0,B-geo-loc,0,0,0,0
3     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,…
4                                          0,0,0,0,0,0
…                                                  …
2389  0,0,0,B-person,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0…
2390              0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2391  0,0,0,B-person,I-person,I-person,0,0,0,0,0,0,0…
2392                    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2393                                   0,0,0,0,0,0,0

[2394 rows x 2 columns]
```

[124]:
```
MAX_LEN = 128
TRAIN_BATCH_SIZE = 4
VALID_BATCH_SIZE = 2
EPOCHS = 10
LEARNING_RATE = 1e-05
MAX_GRAD_NORM = 10
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

[108]:
```
def tokenize_and_preserve_labels(sentence, text_labels, tokenizer):

    tokenized_sentence = []
    labels = []

    sentence = sentence.strip()

    for word, label in zip(sentence.split(), text_labels.split(",")):

        # Tokenize the word and count # of subwords the word is broken into
        tokenized_word = tokenizer.tokenize(word)
        n_subwords = len(tokenized_word)

        # Add the tokenized word to the final tokenized word list
        tokenized_sentence.extend(tokenized_word)

        # Add the same label to the new list of labels `n_subwords` times
        labels.extend([label] * n_subwords)

    return tokenized_sentence, labels
```

```
[112]: class dataset(Dataset):
    def __init__(self, dataframe, tokenizer, max_len):
        self.len = len(dataframe)
        self.data = dataframe
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __getitem__(self, index):
        # step 1: tokenize (and adapt corresponding labels)
        sentence = self.data.sentence[index]
        word_labels = self.data.label[index]
        tokenized_sentence, labels = tokenize_and_preserve_labels(sentence,␣
↪word_labels, self.tokenizer)

        # step 2: add special tokens (and corresponding labels)
        tokenized_sentence = ["[CLS]"] + tokenized_sentence + ["[SEP]"] # add␣
↪special tokens
        labels.insert(0, "O") # add outside label for [CLS] token
        labels.insert(-1, "O") # add outside label for [SEP] token

        # step 3: truncating/padding
        maxlen = self.max_len

        if (len(tokenized_sentence) > maxlen):
          # truncate
          tokenized_sentence = tokenized_sentence[:maxlen]
          labels = labels[:maxlen]
        else:
          # pad
          tokenized_sentence = tokenized_sentence + ['[PAD]'for _ in␣
↪range(maxlen - len(tokenized_sentence))]
          labels = labels + ["O" for _ in range(maxlen - len(labels))]

        # step 4: obtain the attention mask
        attn_mask = [1 if tok != '[PAD]' else 0 for tok in tokenized_sentence]

        # step 5: convert tokens to input ids
        ids = self.tokenizer.convert_tokens_to_ids(tokenized_sentence)

        label_ids = [label2id[label] for label in labels]
        # the following line is deprecated
        #label_ids = [label if label != 0 else -100 for label in label_ids]

        return {
              'ids': torch.tensor(ids, dtype=torch.long),
              'mask': torch.tensor(attn_mask, dtype=torch.long),
              #'token_type_ids': torch.tensor(token_ids, dtype=torch.long),
              'targets': torch.tensor(label_ids, dtype=torch.long)
        }

    def __len__(self):
        return self.len
```

[113]:
```
train_size = 0.8
train_dataset = data.sample(frac=train_size,random_state=200)
test_dataset = data.drop(train_dataset.index).reset_index(drop=True)
train_dataset = train_dataset.reset_index(drop=True)

print("FULL Dataset: {}".format(data.shape))
print("TRAIN Dataset: {}".format(train_dataset.shape))
print("TEST Dataset: {}".format(test_dataset.shape))

training_set = dataset(train_dataset, tokenizer, MAX_LEN)
testing_set = dataset(test_dataset, tokenizer, MAX_LEN)
```

```
FULL Dataset: (2394, 2)
TRAIN Dataset: (1915, 2)
TEST Dataset: (479, 2)
```

[114]:
```
training_set[0]
```

[114]:
```
{'ids': tensor([  101,  1030,  3565, 20147,  2064,  1045,  2131,  1037, 10474,
4007,
         2000, 11867,  7974,  6721, 14636,  2055, 13586,  1013,  4157,  1013,
         9281,  1029,  3426,  2008,  2052,  2489,  2039,  2070,  2051,  1012,
          102,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
              0,      0,      0,      0,      0,      0,      0,      0,      0,      0,
              0,      0,      0,      0,      0,      0,      0,      0,      0,      0,
              0,      0,      0,      0,      0,      0,      0,      0,      0,      0,
              0,      0,      0,      0,      0,      0,      0,      0,      0,      0,
              0,      0,      0,      0,      0,      0,      0,      0,      0,      0,
              0,      0,      0,      0,      0,      0,      0,      0,      0,      0,
              0,      0,      0,      0,      0,      0,      0,      0,      0,      0,
              0,      0,      0,      0,      0,      0,      0,      0]),
 'mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0]),
 'targets': tensor([0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0])}
```

[115]:
```
training_set[0]["ids"]
```

[115]:
```
tensor([  101,  1030,  3565, 20147,  2064,  1045,  2131,  1037, 10474,  4007,
         2000, 11867,  7974,  6721, 14636,  2055, 13586,  1013,  4157,  1013,
         9281,  1029,  3426,  2008,  2052,  2489,  2039,  2070,  2051,  1012,
          102,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0])
```

[116]:
```
# print the first 30 tokens and corresponding labels
for token, label in zip(tokenizer.convert_ids_to_tokens(training_set[0]["ids"][:
↪30]), training_set[0]["targets"][:30]):
  print('{0:10}  {1}'.format(token, id2label[label.item()]))
```

```
[CLS]       O
@           O
super       O
##anne      O
```

```
can          O
i            O
get          O
a            O
twitter      B-company
software     O
to           O
sp           O
##ew         O
random       O
bullshit     O
about        O
airports     O
/            O
coffee       O
/            O
conferences  O
?            O
cause        O
that         O
would        O
free         O
up           O
some         O
time         O
.            O
```

[117]:
```
train_params = {'batch_size': TRAIN_BATCH_SIZE,
                'shuffle': True,
                'num_workers': 0
                }

test_params = {'batch_size': VALID_BATCH_SIZE,
                'shuffle': True,
                'num_workers': 0
                }

training_loader = DataLoader(training_set, **train_params)
testing_loader = DataLoader(testing_set, **test_params)
```

[118]:
```
model = BertForTokenClassification.from_pretrained('bert-base-uncased',
                                                   num_labels=len(id2label),
                                                   id2label=id2label,
                                                   label2id=label2id)
model.to(device)
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertForTokenClassification:
```

```
['cls.predictions.transform.LayerNorm.weight',
'cls.predictions.transform.dense.bias', 'cls.predictions.bias',
'cls.predictions.transform.dense.weight',
'cls.predictions.transform.LayerNorm.bias', 'cls.seq_relationship.bias',
'cls.seq_relationship.weight', 'cls.predictions.decoder.weight']
- This IS expected if you are initializing BertForTokenClassification from the
checkpoint of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing BertForTokenClassification from
the checkpoint of a model that you expect to be exactly identical (initializing
a BertForSequenceClassification model from a BertForSequenceClassification
model).
Some weights of BertForTokenClassification were not initialized from the model
checkpoint at bert-base-uncased and are newly initialized: ['classifier.weight',
'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

[118]:
```
BertForTokenClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (1): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (2): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
```

```
      (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (3): BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
      (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (4): BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
      (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (5): BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
      (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (6): BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
      (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (7): BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
      (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (8): BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
      (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (9): BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
      (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (10): BertLayer(
    (attention): BertAttention(
      (self): BertSelfAttention(
        (query): Linear(in_features=768, out_features=768, bias=True)
        (key): Linear(in_features=768, out_features=768, bias=True)
        (value): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
```

```
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=21, bias=True)
)
```

[119]:
```
ids = training_set[0]["ids"].unsqueeze(0)
mask = training_set[0]["mask"].unsqueeze(0)
targets = training_set[0]["targets"].unsqueeze(0)
ids = ids.to(device)
```

```
mask = mask.to(device)
targets = targets.to(device)
outputs = model(input_ids=ids, attention_mask=mask, labels=targets)
initial_loss = outputs[0]
initial_loss
```

```
[119]: tensor(2.9067, device='cuda:0', grad_fn=<NllLossBackward0>)
```

```
[120]: tr_logits = outputs[1]
       tr_logits.shape
```

```
[120]: torch.Size([1, 128, 21])
```

```
[121]: optimizer = torch.optim.Adam(params=model.parameters(), lr=LEARNING_RATE)
```

```
[122]: def train(epoch):
           tr_loss, tr_accuracy = 0, 0
           nb_tr_examples, nb_tr_steps = 0, 0
           tr_preds, tr_labels = [], []
           # put model in training mode
           model.train()

           for idx, batch in enumerate(training_loader):

               ids = batch['ids'].to(device, dtype = torch.long)
               mask = batch['mask'].to(device, dtype = torch.long)
               targets = batch['targets'].to(device, dtype = torch.long)

               outputs = model(input_ids=ids, attention_mask=mask, labels=targets)
               loss, tr_logits = outputs.loss, outputs.logits
               tr_loss += loss.item()

               nb_tr_steps += 1
               nb_tr_examples += targets.size(0)

               if idx % 100==0:
                   loss_step = tr_loss/nb_tr_steps
                   print(f"Training loss per 100 training steps: {loss_step}")

               # compute training accuracy
               flattened_targets = targets.view(-1) # shape (batch_size * seq_len,)
               active_logits = tr_logits.view(-1, model.num_labels) # shape␣
        ↪(batch_size * seq_len, num_labels)
               flattened_predictions = torch.argmax(active_logits, axis=1) # shape␣
        ↪(batch_size * seq_len,)
               # now, use mask to determine where we should compare predictions with␣
        ↪targets (includes [CLS] and [SEP] token predictions)
               active_accuracy = mask.view(-1) == 1 # active accuracy is also of shape␣
        ↪(batch_size * seq_len,)
               targets = torch.masked_select(flattened_targets, active_accuracy)
               predictions = torch.masked_select(flattened_predictions,␣
        ↪active_accuracy)

               tr_preds.extend(predictions)
               tr_labels.extend(targets)

               tmp_tr_accuracy = accuracy_score(targets.cpu().numpy(), predictions.
        ↪cpu().numpy())
               tr_accuracy += tmp_tr_accuracy

               # gradient clipping
               torch.nn.utils.clip_grad_norm_(
                   parameters=model.parameters(), max_norm=MAX_GRAD_NORM
               )

               # backward pass
               optimizer.zero_grad()
               loss.backward()
               optimizer.step()

           epoch_loss = tr_loss / nb_tr_steps
           tr_accuracy = tr_accuracy / nb_tr_steps
           print(f"Training loss epoch: {epoch_loss}")
           print(f"Training accuracy epoch: {tr_accuracy}")
```

[125]:
```
for epoch in range(EPOCHS):
    print(f"Training epoch: {epoch + 1}")
    train(epoch)
```

```
Training epoch: 1
Training loss per 100 training steps: 0.06929675489664078
Training loss per 100 training steps: 0.053565228719835975
Training loss per 100 training steps: 0.050586917432979564
Training loss per 100 training steps: 0.04808246714304328
Training loss per 100 training steps: 0.0475751140678622
Training loss epoch: 0.04790579551093996
Training accuracy epoch: 0.9590183286945262
Training epoch: 2
Training loss per 100 training steps: 0.04215997830033302
Training loss per 100 training steps: 0.03712692446914492
Training loss per 100 training steps: 0.03649516081187262
Training loss per 100 training steps: 0.03590065079425582
Training loss per 100 training steps: 0.03553006039181041
Training loss epoch: 0.03396850670755905
Training accuracy epoch: 0.9682017397128238
Training epoch: 3
Training loss per 100 training steps: 0.03562479838728905
Training loss per 100 training steps: 0.025924439903920387
Training loss per 100 training steps: 0.02644293190939213
Training loss per 100 training steps: 0.025053960830824097
Training loss per 100 training steps: 0.024339997132385424
Training loss epoch: 0.024882503507400247
Training accuracy epoch: 0.9753222433888027
Training epoch: 4
Training loss per 100 training steps: 0.0019726348109543324
Training loss per 100 training steps: 0.01626833022161765
Training loss per 100 training steps: 0.018292514582405526
Training loss per 100 training steps: 0.018759091248375322
Training loss per 100 training steps: 0.019167041538272776
Training loss epoch: 0.01842798689761372
Training accuracy epoch: 0.9816473252221113
Training epoch: 5
Training loss per 100 training steps: 0.0008706478402018547
Training loss per 100 training steps: 0.018135244703372147
Training loss per 100 training steps: 0.01588568390076007
Training loss per 100 training steps: 0.014712965404815922
Training loss per 100 training steps: 0.01425880267333583
Training loss epoch: 0.013970131969924176
Training accuracy epoch: 0.9860617081633083
Training epoch: 6
Training loss per 100 training steps: 0.021439671516418457
Training loss per 100 training steps: 0.012446737448260704
Training loss per 100 training steps: 0.010536172018335456
Training loss per 100 training steps: 0.010129752745263997
Training loss per 100 training steps: 0.010078357787369669
Training loss epoch: 0.01022619594497089
Training accuracy epoch: 0.9901233446605672
Training epoch: 7
Training loss per 100 training steps: 0.007052768487483263
Training loss per 100 training steps: 0.007571113483155017
Training loss per 100 training steps: 0.007643858531046316
Training loss per 100 training steps: 0.007538463997610526
Training loss per 100 training steps: 0.007384444169608407
Training loss epoch: 0.007839718331509447
Training accuracy epoch: 0.9932364386489853
Training epoch: 8
Training loss per 100 training steps: 0.0022671876940876245
Training loss per 100 training steps: 0.0073342239847268426
Training loss per 100 training steps: 0.006448446606912544
Training loss per 100 training steps: 0.006033509025432207
Training loss per 100 training steps: 0.005855275300752948
Training loss epoch: 0.005983862560181655
Training accuracy epoch: 0.9950161624421213
Training epoch: 9
Training loss per 100 training steps: 0.001748141017742455
Training loss per 100 training steps: 0.004751914223174261
Training loss per 100 training steps: 0.004747931765559227
Training loss per 100 training steps: 0.005169166667882718
Training loss per 100 training steps: 0.005119441130240343
Training loss epoch: 0.005041133050732937
Training accuracy epoch: 0.9960488598854534
Training epoch: 10
Training loss per 100 training steps: 0.00241595646366477
Training loss per 100 training steps: 0.003791343578941991
Training loss per 100 training steps: 0.004032732243533826
Training loss per 100 training steps: 0.0038039785575050477
Training loss per 100 training steps: 0.003909893949809914
Training loss epoch: 0.004040137354147451
Training accuracy epoch: 0.9962345635927358
```

[126]:
```
def valid(model, testing_loader):
    # put model in evaluation mode
    model.eval()

    eval_loss, eval_accuracy = 0, 0
    nb_eval_examples, nb_eval_steps = 0, 0
    eval_preds, eval_labels = [], []

    with torch.no_grad():
        for idx, batch in enumerate(testing_loader):

            ids = batch['ids'].to(device, dtype = torch.long)
            mask = batch['mask'].to(device, dtype = torch.long)
            targets = batch['targets'].to(device, dtype = torch.long)

            outputs = model(input_ids=ids, attention_mask=mask, labels=targets)
            loss, eval_logits = outputs.loss, outputs.logits

            eval_loss += loss.item()

            nb_eval_steps += 1
            nb_eval_examples += targets.size(0)

            if idx % 100==0:
                loss_step = eval_loss/nb_eval_steps
                print(f"Validation loss per 100 evaluation steps: {loss_step}")

            # compute evaluation accuracy
            flattened_targets = targets.view(-1) # shape (batch_size * seq_len,)
            active_logits = eval_logits.view(-1, model.num_labels) # shape␣
↪(batch_size * seq_len, num_labels)
            flattened_predictions = torch.argmax(active_logits, axis=1) # shape␣
↪(batch_size * seq_len,)
            # now, use mask to determine where we should compare predictions␣
↪with targets (includes [CLS] and [SEP] token predictions)
            active_accuracy = mask.view(-1) == 1 # active accuracy is also of␣
↪shape (batch_size * seq_len,)
            targets = torch.masked_select(flattened_targets, active_accuracy)
            predictions = torch.masked_select(flattened_predictions,␣
↪active_accuracy)

            eval_labels.extend(targets)
            eval_preds.extend(predictions)

            tmp_eval_accuracy = accuracy_score(targets.cpu().numpy(),␣
↪predictions.cpu().numpy())
            eval_accuracy += tmp_eval_accuracy

    #print(eval_labels)
    #print(eval_preds)

    labels = [id2label[id.item()] for id in eval_labels]
    predictions = [id2label[id.item()] for id in eval_preds]

    #print(labels)
    #print(predictions)

    eval_loss = eval_loss / nb_eval_steps
    eval_accuracy = eval_accuracy / nb_eval_steps
    print(f"Validation Loss: {eval_loss}")
    print(f"Validation Accuracy: {eval_accuracy}")

    return labels, predictions
```

[127]:
```
labels, predictions = valid(model, testing_loader)
```

```
Validation loss per 100 evaluation steps: 0.00020672072423622012
Validation loss per 100 evaluation steps: 0.04670936920628814
Validation loss per 100 evaluation steps: 0.04190672623143886
Validation Loss: 0.04038859859538206
Validation Accuracy: 0.9742906349364928
```

[128]:
```
from seqeval.metrics import classification_report

print(classification_report([labels], [predictions]))
```

```
              precision    recall  f1-score   support
```

```
      company       0.61      0.61      0.61        33
     facility       0.71      0.35      0.47        43
      geo-loc       0.56      0.91      0.69        45
        movie       0.43      0.27      0.33        11
  musicartist       0.00      0.00      0.00        12
        other       0.23      0.14      0.18        76
       person       0.82      0.80      0.81       100
      product       0.57      0.33      0.42        24
   sportsteam       0.69      0.69      0.69        13
       tvshow       0.30      0.25      0.27        12

    micro avg       0.60      0.51      0.55       369
    macro avg       0.49      0.44      0.45       369
 weighted avg       0.56      0.51      0.52       369
```

[129]:
```python
test_data = pd.read_csv("test.csv", encoding='unicode_escape',index_col=[0])
test_data.head()
```

[129]:
```
     Words     Tags
0      New  B-other
1  Orleans  I-other
2   Mother  I-other
3       's  I-other
4      Day  I-other
```

[130]:
```python
sentences = []
for text in test_text:
    sentence = ""
    for i in text:
        sentence+=i+" "
    sentences.append(sentence)
total_tags = []
for tags in test_tags:
    tag = ""
    for i in tags:
        tag+=i + ","
    total_tags.append(tag[:-1])


test_data["sentence"] = pd.Series(sentences)
test_data["label"] = pd.Series(total_tags)
test_data.drop(["Words","Tags"],inplace=True,axis=1)
test_data.dropna(inplace=True)
```

[131]:
```python
test_data
```

[131]:
```
                                               sentence  \
0     New Orleans Mother 's Day Parade shooting . On…
1     RT @hxranspizza : Going into school tomorrow l…
2     May e just a smile in your heart EILY Countdow…
3                I could so do Thursday Club right now
4     @therealdaftbear Albert Nobbs ( Glenn Close)is…
…                                                  …
3845  Priest killed , another injured in US shooting…
3846  Michael__Myerz : |LIVE NOW| Yes #meerkat https…
3847  http://t.co/MoMmuSaDKE Daily Fantasy Basketbal…
3848  @Toniakins no man alive has it all . But you c…
3849  RT @NaddictsOfc : She 's living with her famil…

                                                  label
0     B-other,I-other,I-other,I-other,I-other,I-othe…
1                            0,0,0,0,0,0,0,0,0,0,0,0
2                      0,0,0,0,0,0,0,0,B-movie,0,0
3                                  0,0,0,0,0,0,0,0
4     0,B-person,I-person,0,B-person,0,0,0,0,0,0,0,0…
…                                                  …
3845  0,0,0,0,0,0,B-geo-loc,0,0,B-geo-loc,I-geo-loc,…
3846                     0,0,0,0,0,0,0,0,0,0,0,0,0,0
3847      0,0,0,0,0,0,0,B-other,0,B-company,0,0,0,0,0,0
3848  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,…
3849                           0,0,0,0,0,0,0,0,0,0,0,0

[3850 rows x 2 columns]
```

[134]:
```
testing_set = dataset(test_data, tokenizer, MAX_LEN)
```

[135]:
```
testing_set[0]
```

[135]:
```
{'ids': tensor([ 101, 2047, 5979, 2388, 1005, 1055, 2154, 7700, 5008, 1012,
2028, 1997,
         1996, 2111, 3480, 2001, 1037, 2184, 1011, 2095, 1011, 2214, 2611, 1012,
         2054, 1996, 3109, 2003, 3308, 2007, 2111, 1029,  102,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0]),
 'mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0]),
 'targets': tensor([ 0,  9, 10, 10, 10, 10, 10, 10,  0,  0,  0,  0,  0,  0,  0,
0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0])}
```

[136]:
```
test_params = {'batch_size': 4,
                'shuffle': False,
                'num_workers': 0
                }
testing_loader = DataLoader(testing_set, **test_params)
```

[137]:
```
labels, predictions = valid(model, testing_loader)
```

```
Validation loss per 100 evaluation steps: 0.08591140806674957
Validation loss per 100 evaluation steps: 0.103067406943635
Validation loss per 100 evaluation steps: 0.10063853276601299
Validation loss per 100 evaluation steps: 0.10158272669984196
Validation loss per 100 evaluation steps: 0.09894667951045547
Validation loss per 100 evaluation steps: 0.09846239740300497
Validation loss per 100 evaluation steps: 0.0936237592795866
Validation loss per 100 evaluation steps: 0.09177889410474357
Validation loss per 100 evaluation steps: 0.09134200254667983
Validation loss per 100 evaluation steps: 0.09140491785799615
Validation Loss: 0.09194363826059972
Validation Accuracy: 0.9513486634783093
```

[138]:
```
from seqeval.metrics import classification_report

print(classification_report([labels], [predictions]))
```

```
              precision    recall  f1-score   support

     company       0.64      0.41      0.50      1337
    facility       0.21      0.14      0.17       390
     geo-loc       0.47      0.60      0.53      1244
       movie       0.16      0.05      0.07        66
 musicartist       0.47      0.02      0.04       326
       other       0.28      0.18      0.22      1004
      person       0.54      0.58      0.56       726
     product       0.22      0.08      0.12       428
  sportsteam       0.56      0.22      0.32       247
      tvshow       0.14      0.06      0.08        71

   micro avg       0.46      0.35      0.40      5839
   macro avg       0.37      0.23      0.26      5839
weighted avg       0.44      0.35      0.37      5839
```

[140]:
```
sentence = "India has a capital called Mumbai. On wednesday, the president will␣
 ↪give a presentation"

inputs = tokenizer(sentence, padding='max_length', truncation=True,␣
 ↪max_length=MAX_LEN, return_tensors="pt")

# move to gpu
ids = inputs["input_ids"].to(device)
mask = inputs["attention_mask"].to(device)
# forward pass
outputs = model(ids, mask)
logits = outputs[0]

active_logits = logits.view(-1, model.num_labels) # shape (batch_size *␣
 ↪seq_len, num_labels)
flattened_predictions = torch.argmax(active_logits, axis=1) # shape␣
 ↪(batch_size*seq_len,) - predictions at the token level

tokens = tokenizer.convert_ids_to_tokens(ids.squeeze().tolist())
token_predictions = [id2label[i] for i in flattened_predictions.cpu().numpy()]
wp_preds = list(zip(tokens, token_predictions)) # list of tuples. Each tuple =␣
 ↪(wordpiece, prediction)
```

[141]:
```
wp_preds
```

[141]:
```
[('[CLS]', 'O'),
 ('india', 'B-geo-loc'),
 ('has', 'O'),
 ('a', 'O'),
 ('capital', 'O'),
 ('called', 'O'),
 ('mumbai', 'B-geo-loc'),
 ('.', 'O'),
 ('on', 'O'),
 ('wednesday', 'O'),
 (',', 'O'),
 ('the', 'O'),
 ('president', 'O'),
 ('will', 'O'),
 ('give', 'O'),
 ('a', 'O'),
 ('presentation', 'O'),
 ('[SEP]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
 ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O'),
  ('[PAD]', 'O')]
```

[139]:
```
model.save_pretrained("Bert_Ner.th")
tokenizer.save_pretrained("Tokenizer_Bert_Ner.th")
```

[139]:
```
('Tokenizer_Bert_Ner.th\\tokenizer_config.json',
 'Tokenizer_Bert_Ner.th\\special_tokens_map.json',
 'Tokenizer_Bert_Ner.th\\vocab.txt',
 'Tokenizer_Bert_Ner.th\\added_tokens.json')
```