Task 1: Simple Bidirectional LSTM model

In [1]:
```
from collections import defaultdict
import operator
import pandas as pd
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torch.utils.data.sampler import SubsetRandomSampler
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence, pad_packed_sequence
import torch
import torch.nn as nn
import torch.nn.functional as F
import pandas as pd
from collections import defaultdict
import operator
```

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

In [3]:
```
def read_and_preprocess_data(file_path):
    f = open(file_path, "r")
    count_dict = defaultdict(int)
    label_set = []
    for line in f:
        get_words = line.split()
        if len(get_words) != 0:
            count_dict[get_words[1]] += 1
            if get_words[2] not in label_set:
                label_set.append(get_words[2])
    f.close()

    unkw = 0
    for key, val in count_dict.items():
        if val < 2:
            unkw += val

    sorted_count_list = sorted(count_dict.items(), key=operator.itemgetter(1), reverse=True)

    word_index = {}
    word_index['<PAD>'] = 0
    word_index['<UNK>'] = 1

    i = 2
    for word, count in sorted_count_list:
        if count >= 2:
            word_index[word] = i
            i += 1

    f_train = open(file_path, "r")
    sentences = []
    tags = []
    curr_sent = ""
    curr_tags = ""

    for line in f_train:
        get_line = line.split()
        if len(get_line) > 0:
            curr_sent += get_line[1]
            curr_sent += " "
            curr_tags += get_line[2]
```

```
            curr_tags += " "
        else:
            curr_sent = curr_sent[:-1]
            curr_tags = curr_tags[:-1]
            sentences.append(curr_sent)
            tags.append(curr_tags)
            curr_sent = ""
            curr_tags = ""
    f_train.close()

    curr_sent = curr_sent[:-1]
    curr_tags = curr_tags[:-1]
    sentences.append(curr_sent)
    tags.append(curr_tags)

    return sentences, tags, word_index, label_set
```

What I did above was develop a function named read_data that pulls phrases and their matching tags from a file using a file path as input. I divided each line in the file into words by looping over iterations within this programme. I took out the second element as a word and, if it was available, the third element as a tag for every line that wasn't empty. I removed trailing spaces before adding the collected sentences and tags to the appropriate categories whenever I came across an empty line or the end of a sentence. Lastly, I added the final sentence to the lists if it wasn't followed by an empty line. The lists of sentences and tags that were taken out of the file are then returned by the function.

The preprocess_data function takes sentences and tags as input, counts the occurrences of words in the sentences, and assigns indices to words based on their frequency. It also creates a mapping of unique labels to indices. The function returns dictionaries for word indices and label indices.

In [4]:
```
def create_data_frame(sentences, tags):
    return pd.DataFrame({'sentences': sentences, 'tags': tags})
```

In [5]:
```
def create_test_data(file_path):
    f_test = open(file_path, "r")
    sentences = []
    curr_sent = ""

    for line in f_test:
        get_line = line.split()
        if len(get_line) > 0:
            curr_sent += get_line[1]
            curr_sent += " "
        else:
            curr_sent = curr_sent[:-1]
            sentences.append(curr_sent)
            curr_sent = ""
    f_test.close()

    curr_sent = curr_sent[:-1]
    sentences.append(curr_sent)

    return pd.DataFrame({'sentences': sentences})
```

In [6]:
```
def create_label_index(label_set):
    label_index = {}
    i = 0
    for label in label_set:
        label_index[label] = i
        i += 1
    label_index['pad_label'] = -1
    return label_index
```

In [7]:
```
def create_index_word(word_index):
    return {v: k for k, v in word_index.items()}

def create_index_label(label_index):
    return {v: k for k, v in label_index.items()}
```

In [9]:
```
train_sentences, train_tags, word_index, label_set = read_and_preprocess_data("./data/train")
train_data = create_data_frame(train_sentences, train_tags)
test_data = create_test_data("./data/test")
label_index = create_label_index(label_set)
index_word = create_index_word(word_index)
index_label = create_index_label(label_index)
```

In [10]:
```
f_dev = open("./data/dev","r")
sentences = []
tags = []
curr_sent = ""
curr_tags = ""

for line in f_dev:
    get_line = line.split()
    if len(get_line)>0:
        curr_sent += get_line[1]
        curr_sent += " "
        curr_tags += get_line[2]
        curr_tags += " "
    else:
        curr_sent = curr_sent[:-1]
        curr_tags = curr_tags[:-1]
        sentences.append(curr_sent)
        tags.append(curr_tags)
        curr_sent = ""
        curr_tags = ""
f_dev.close()

curr_sent = curr_sent[:-1]
curr_tags = curr_tags[:-1]
sentences.append(curr_sent)
tags.append(curr_tags)
curr_sent = ""
curr_tags = ""

dev_data = pd.DataFrame({'sentences':sentences, 'tags':tags})
```

In [11]:
```
train_data
```

Out[11]:

	sentences	tags
0	EU rejects German call to boycott British lamb .	B-ORG O B-MISC O O O B-MISC O O
1	Peter Blackburn	B-PER I-PER
2	BRUSSELS 1996-08-22	B-LOC O
3	The European Commission said on Thursday it di...	O B-ORG I-ORG O O O O O O B-MISC O O O O O B-M...
4	Germany 's representative to the European Unio...	B-LOC O O O O B-ORG I-ORG O O O B-PER I-PER O ...
...	...	...
14982	Division two	O O
14983	Plymouth 2 Preston 1	B-ORG O B-ORG O
14984	Division three	O O
14985	Swansea 1 Lincoln 2	B-ORG O B-ORG O
14986	-DOCSTART-	O

14987 rows × 2 columns

In [12]:
```
class TrainDataBiLSTM:
    def __init__(self, sentences, tags, word_index, label_index):
        self.sentences = sentences
        self.tags = tags
        self.word_index = word_index
        self.label_index = label_index

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

    def __getitem__(self, i):
        sentence = self.sentences.iloc[i].split()
        ner_tag = self.tags.iloc[i].split()

        sentence = [self.word_index.get(word, self.word_index['<UNK>']) for word in sentence]
        ner_tag = [self.label_index[tag] for tag in ner_tag]

        sentence = torch.tensor(sentence)
        ner_tag = torch.tensor(ner_tag)

        return sentence, ner_tag
```

In [13]:
```
class DevDataBiLSTM:
    def __init__(self, sentences, tags, word_index, label_index):
        self.sentences = sentences
        self.tags = tags
        self.word_index = word_index
        self.label_index = label_index

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

    def __getitem__(self, i):
        sentence = self.sentences.iloc[i].split()
        ner_tag = self.tags.iloc[i].split()

        sentence = [self.word_index.get(word, self.word_index['<UNK>']) for word in sentence]
        ner_tag = [self.label_index[tag] for tag in ner_tag]

        sentence = torch.tensor(sentence)
        ner_tag = torch.tensor(ner_tag)

        return sentence, ner_tag
```

In [14]:
```
class TestDataBiLSTM:
    def __init__(self, sentences, word_index):
        self.sentences = sentences
        self.word_index = word_index

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

    def __getitem__(self, i):
        sentence = self.sentences.iloc[i].split()

        sentence = [self.word_index.get(word, self.word_index['<UNK>']) for word in sentence]

        sentence = torch.tensor(sentence)

        return sentence
```

In [15]:
```
def pad_collate(batch):
    sentences, ner_tags = zip(*batch)

    lengths = torch.tensor([len(sentence) for sentence in sentences])
    sentences = pad_sequence(sentences, batch_first=True, padding_value=0)
    ner_tags = pad_sequence(ner_tags, batch_first=True, padding_value=-1)

    return sentences, lengths, ner_tags

def pad_collate_test(batch):
    sentences = batch

    lengths = torch.tensor([len(sentence) for sentence in sentences])
    sentences = pad_sequence(sentences, batch_first=True, padding_value=0)

    return sentences, lengths
```

In [16]:
```
batch_size=16

train_dataset = TrainDataBiLSTM(train_data['sentences'], train_data['tags'], word_index, label_index)
train_loader = DataLoader(train_dataset, batch_size=batch_size, collate_fn=pad_collate)

dev_dataset = DevDataBiLSTM(dev_data['sentences'], dev_data['tags'], word_index, label_index)
dev_loader = DataLoader(dev_dataset, batch_size=batch_size, collate_fn=pad_collate)

test_dataset = TestDataBiLSTM(test_data['sentences'], word_index)
test_loader = DataLoader(test_dataset, batch_size=batch_size, collate_fn=pad_collate_test)
```

In [17]:
```
class BiLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, lstm_hidden_dim, lstm_dropout, linear_output_dim, num_tags)
        super(BiLSTM, self).__init__()

        self.embedding = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embedding_dim, padding_idx=0)
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=lstm_hidden_dim, num_layers=1,
                            batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(0.33)
        self.linear = nn.Linear(lstm_hidden_dim*2, linear_output_dim)
        self.elu = nn.ELU(0.35)
        self.classifier = nn.Linear(linear_output_dim, num_tags)

    def forward(self, inputs, lengths):
        embedded = self.embedding(inputs)
        packed_embedded = pack_padded_sequence(embedded, lengths.cpu(), batch_first=True, enforce_sorted=Fals
        packed_output, (hidden, cell) = self.lstm(packed_embedded)
        output, _ = pad_packed_sequence(packed_output, batch_first=True)
        output = self.dropout(output)
        linear_output = self.linear(output)
        elu_output = self.elu(linear_output)
        logits = self.classifier(elu_output)

        return logits
```

In [18]:
```
bilstm_model = BiLSTM(len(word_index.keys()), 100, 256, 0.33, 128, 9).to(device)
print(bilstm_model)

criterion = nn.CrossEntropyLoss(ignore_index=-1)
optimizer = torch.optim.SGD(bilstm_model.parameters(), lr=0.33)

epochs = 75
```

```
BiLSTM(
  (embedding): Embedding(11985, 100, padding_idx=0)
  (lstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=0.35)
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)
```

In [19]:
```
for epoch in range(epochs):
    train_loss = 0.0

    bilstm_model.train()
    for sentences, lengths, labels in train_loader:
        # Move input data and labels to GPU
        sentences, lengths, labels = sentences.to(device), lengths.to(device), labels.to(device)

        optimizer.zero_grad()
        output = bilstm_model(sentences, lengths)
        output = output.permute(0, 2, 1)
        loss = criterion(output, labels)

        loss.backward()
        optimizer.step()

        train_loss += loss.item() * sentences.size(0)

    train_loss = train_loss / (len(train_loader.dataset))

    print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
```

```
Epoch: 1        Training Loss: 0.681628
Epoch: 2        Training Loss: 0.465434
Epoch: 3        Training Loss: 0.349854
Epoch: 4        Training Loss: 0.271496
Epoch: 5        Training Loss: 0.218479
Epoch: 6        Training Loss: 0.182319
Epoch: 7        Training Loss: 0.155231
Epoch: 8        Training Loss: 0.134901
Epoch: 9        Training Loss: 0.117522
Epoch: 10       Training Loss: 0.102957
Epoch: 11       Training Loss: 0.092544
Epoch: 12       Training Loss: 0.083159
Epoch: 13       Training Loss: 0.076716
Epoch: 14       Training Loss: 0.067547
Epoch: 15       Training Loss: 0.062104
Epoch: 16       Training Loss: 0.056442
Epoch: 17       Training Loss: 0.051509
Epoch: 18       Training Loss: 0.048947
Epoch: 19       Training Loss: 0.043988
Epoch: 20       Training Loss: 0.041682
Epoch: 21       Training Loss: 0.037791
Epoch: 22       Training Loss: 0.035401
Epoch: 23       Training Loss: 0.032947
Epoch: 24       Training Loss: 0.029896
Epoch: 25       Training Loss: 0.029225
Epoch: 26       Training Loss: 0.027386
Epoch: 27       Training Loss: 0.026035
Epoch: 28       Training Loss: 0.023837
Epoch: 29       Training Loss: 0.022181
Epoch: 30       Training Loss: 0.022469
Epoch: 31       Training Loss: 0.020508
Epoch: 32       Training Loss: 0.019236
Epoch: 33       Training Loss: 0.017796
Epoch: 34       Training Loss: 0.017783
Epoch: 35       Training Loss: 0.016240
Epoch: 36       Training Loss: 0.015785
Epoch: 37       Training Loss: 0.015519
Epoch: 38       Training Loss: 0.014012
Epoch: 39       Training Loss: 0.014126
Epoch: 40       Training Loss: 0.013775
Epoch: 41       Training Loss: 0.013121
```

```
Epoch: 42       Training Loss: 0.011916
Epoch: 43       Training Loss: 0.012552
Epoch: 44       Training Loss: 0.011261
Epoch: 45       Training Loss: 0.010600
Epoch: 46       Training Loss: 0.009980
Epoch: 47       Training Loss: 0.009884
Epoch: 48       Training Loss: 0.010606
Epoch: 49       Training Loss: 0.009497
Epoch: 50       Training Loss: 0.008721
Epoch: 51       Training Loss: 0.009603
Epoch: 52       Training Loss: 0.008502
Epoch: 53       Training Loss: 0.008443
Epoch: 54       Training Loss: 0.008063
Epoch: 55       Training Loss: 0.007976
Epoch: 56       Training Loss: 0.007517
Epoch: 57       Training Loss: 0.008738
Epoch: 58       Training Loss: 0.007534
Epoch: 59       Training Loss: 0.007532
Epoch: 60       Training Loss: 0.007051
Epoch: 61       Training Loss: 0.006904
Epoch: 62       Training Loss: 0.006665
Epoch: 63       Training Loss: 0.006741
Epoch: 64       Training Loss: 0.006841
Epoch: 65       Training Loss: 0.006180
Epoch: 66       Training Loss: 0.006263
Epoch: 67       Training Loss: 0.005753
Epoch: 68       Training Loss: 0.005625
Epoch: 69       Training Loss: 0.005909
Epoch: 70       Training Loss: 0.005612
Epoch: 71       Training Loss: 0.005612
Epoch: 72       Training Loss: 0.005335
Epoch: 73       Training Loss: 0.005125
Epoch: 74       Training Loss: 0.005026
Epoch: 75       Training Loss: 0.005310
```

In [20]:
```
torch.save(bilstm_model.state_dict(), 'blstm1.pt')
```

In [21]:
```
ef getDevResults(model, dataloader):
    model.eval()

    f_read = open("./data/dev", "r")
    f_write = open("dev1.out", "w")
    for sentences, lengths, labels in dataloader:
        # Move input data to GPU
        sentences, lengths = sentences.to(device), lengths.to(device)

        output = model(sentences, lengths)
        max_values, max_indices = torch.max(output, dim=2)
        y = max_indices

        for i in range(len(sentences)):
            for j in range(len(sentences[i])):
                read_line = f_read.readline().split()
                if len(read_line) > 0:
                    f_write.write(str(read_line[0]) + " " + str(read_line[1]) + " " + index_label[labels[i][j].
                else:
                    break
                if j + 1 >= len(sentences[i]):
                    f_read.readline()
            if len(sentences) == batch_size or i < len(sentences) - 1:
                f_write.write("\n")
    f_read.close()
    f_write.close()
```

In [22]:
```
def getTestResults(model, dataloader):
    model.eval()

    f_read = open("./data/test", "r")
    f_write = open("test1.out", "w")
    for sentences, lengths in dataloader:
        # Move input data to GPU
        sentences, lengths = sentences.to(device), lengths.to(device)

        output = model(sentences, lengths)
        max_values, max_indices = torch.max(output, dim=2)
        y = max_indices

        for i in range(len(sentences)):
            for j in range(len(sentences[i])):
                read_line = f_read.readline().split()
                if len(read_line) > 0:
                    f_write.write(str(read_line[0]) + " " + str(read_line[1]) + " " + index_label[y[i][j].ite
                else:
                    break
                if j + 1 >= len(sentences[i]):
                    f_read.readline()
            if len(sentences) == batch_size or i < len(sentences) - 1:
                f_write.write("\n")
    f_read.close()
    f_write.close()
```

In [23]:
```
getDevResults(bilstm_model, dev_loader)
getTestResults(bilstm_model, test_loader)
```

In [24]:
```
!python eval.py -p dev1.out -g data/dev
```

```
processed 51578 tokens with 5942 phrases; found: 5293 phrases; correct: 4407.
accuracy:  95.60%; precision:  83.26%; recall:  74.17%; FB1:  78.45
              LOC: precision:  90.26%; recall:  82.69%; FB1:  86.31  1683
             MISC: precision:  84.37%; recall:  74.95%; FB1:  79.38  819
              ORG: precision:  73.74%; recall:  66.37%; FB1:  69.86  1207
              PER: precision:  82.51%; recall:  70.96%; FB1:  76.30  1584
```

As we can see the precision, recall and F1 scores are printed above for the dev set.

Task 2: Using GloVe word embeddings

In [38]:
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence, pad_packed_sequence
import numpy as np
```

In [39]:
```
embed_vectors = []
embed_vocab = []
file_embed = open("glove.6B.100d","r")
for line in file_embed:
    line = line.split()
    embed_vocab.append(line[0])
    embed_vectors.append(line[1:])
embed_vocab = np.array(embed_vocab)
embed_vectors = np.array(embed_vectors, dtype=np.float64)
```

In [40]:
```
pad_vector = np.zeros((1,embed_vectors.shape[1]))
unk_vector = np.mean(embed_vectors,axis=0,keepdims=True)

embed_vocab = np.insert(embed_vocab, 0, '<PAD>')
embed_vocab = np.insert(embed_vocab, 1, '<UNK>')

embed_vectors = np.vstack((pad_vector,unk_vector,embed_vectors))
```

In [41]:
```
glove_word_index = {k: v for v, k in enumerate(embed_vocab)}
```

The purpose of the load_embeddings is that I'm trying to read embeddings from a given file directory. The vocabulary words and the associated embedding vectors are stored in two lists that I initialise within this function, embed_vocab and embed_vectors, respectively. Next, I use a for loop to iterate over each line in the file, breaking it up into words and appending the word to embed_vocab and the vector that goes with

it to embed_vectors. I create NumPy arrays from embed_vocab and embed_vectos after processing every embedding. I then append the special tokens '' and '' to {embed_vocab} after that. I compute a padding vector and an unknown token vector and insert them at the start of embed_vectors to make sure they are compatible with the embeddings. In order to link every word in the vocabulary to its matching index in the embeddings, I lastly construct a dictionary called glove_word_index. Next, glove_word_index, embed vocab, and embed vectors are returned by the function.

In [42]:
```
class TrainDataBiLSTMGlove(Dataset):
    def __init__(self, sentences, tags, glove_word_index, label_index):
        self.sentences = sentences
        self.tags = tags
        self.glove_word_index = glove_word_index
        self.label_index = label_index

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

    def __getitem__(self, idx):
        sentence = self.sentences[idx].split()
        ner_tag = self.tags[idx].split()
        is_capital = [1 if (word.isupper() or word.istitle()) else 0 for word in sentence]
        sentence = [self.glove_word_index.get(word.lower(), self.glove_word_index['<UNK>']) for word in sente
        ner_tag = [self.label_index[tag] for tag in ner_tag]

        return torch.tensor(sentence), torch.tensor(is_capital), torch.tensor(ner_tag)
```

In [43]:
```
class DevDataBiLSTMGlove:
    def __init__(self, sentences, tags, glove_word_index, label_index):
        self.sentences = sentences
        self.tags = tags
        self.glove_word_index = glove_word_index
        self.label_index = label_index

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

    def __getitem__(self, i):
        sentence = self.sentences.iloc[i].split()
        ner_tag = self.tags.iloc[i].split()

        is_capital = [1 if (word.isupper() or word.istitle()) else 0 for word in sentence]
        sentence = [self.glove_word_index.get(word.lower(), self.glove_word_index['<UNK>']) for word in sente
        ner_tag = [self.label_index[tag] for tag in ner_tag]

        sentence = torch.tensor(sentence)
        is_capital = torch.tensor(is_capital)
        ner_tag = torch.tensor(ner_tag)

        return sentence, is_capital, ner_tag
```

In [44]:
```
def pad_collate_glove(batch):
    sentences, is_capitals, ner_tags = zip(*batch)
    lengths = torch.tensor([len(sentence) for sentence in sentences])
    sentences = pad_sequence(sentences, batch_first=True, padding_value=0)
    is_capitals = pad_sequence(is_capitals, batch_first=True, padding_value=-1)
    ner_tags = pad_sequence(ner_tags, batch_first=True, padding_value=-1)

    return sentences, is_capitals, lengths, ner_tags
```

In [45]:
```
class TestDataBiLSTMGlove:
    def __init__(self, sentences, glove_word_index):
        self.sentences = sentences
        self.glove_word_index = glove_word_index

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

    def __getitem__(self, i):
        sentence = self.sentences.iloc[i].split()

        is_capital = [1 if (word.isupper() or word.istitle()) else 0 for word in sentence]
        sentence = [self.glove_word_index.get(word.lower(), self.glove_word_index['<UNK>']) for word in sente

        sentence = torch.tensor(sentence)
        is_capital = torch.tensor(is_capital)

        return sentence, is_capital

def pad_collate_glove_test(batch):

    sentences, is_capitals = zip(*batch)

    lengths = torch.tensor([len(sentence) for sentence in sentences])
    sentences = pad_sequence(sentences, batch_first=True, padding_value=0)
    is_capitals = pad_sequence(is_capitals, batch_first=True, padding_value=-1)

    return sentences, is_capitals, lengths
```

In [46]:
```
train_dataset = TrainDataBiLSTMGlove(train_data['sentences'], train_data['tags'], glove_word_index, label_inc
train_loader = DataLoader(train_dataset, batch_size=batch_size, collate_fn=pad_collate_glove)

dev_dataset = DevDataBiLSTMGlove(dev_data['sentences'], dev_data['tags'], glove_word_index, label_index)
dev_loader = DataLoader(dev_dataset, batch_size=batch_size, collate_fn=pad_collate_glove)

test_dataset = TestDataBiLSTMGlove(test_data['sentences'], glove_word_index)
test_loader = DataLoader(test_dataset, batch_size=batch_size, collate_fn=pad_collate_glove_test)
```

In [47]:
```
s BiLSTMGlove(nn.Module):
def __init__(self, embedding_dim, lstm_hidden_dim, lstm_dropout, linear_output_dim, num_tags):
    super(BiLSTMGlove, self).__init__()

    self.embedding = nn.Embedding.from_pretrained(torch.from_numpy(embed_vectors),padding_idx=0)
    self.lstm = nn.LSTM(input_size=embedding_dim+1, hidden_size=lstm_hidden_dim, num_layers=1,
                        batch_first=True, bidirectional=True)
    self.dropout = nn.Dropout(lstm_dropout)
    self.linear = nn.Linear(lstm_hidden_dim*2, linear_output_dim)
    self.elu = nn.ELU()
    self.classifier = nn.Linear(linear_output_dim, num_tags)

def forward(self, inputs, is_capitals, lengths):
    embedded = self.embedding(inputs)
    concatenated_tensor = torch.cat((embedded, is_capitals.unsqueeze(-1)), dim=-1)
    packed_embedded = pack_padded_sequence(concatenated_tensor, lengths.cpu(), batch_first=True, enforce_sorte
    packed_embedded = packed_embedded.float()
    packed_output, (hidden, cell) = self.lstm(packed_embedded)
    output, _ = pad_packed_sequence(packed_output, batch_first=True)
    linear_output = self.linear(output)
    elu_output = self.elu(linear_output)
    logits = self.classifier(elu_output)

    return logits
```

In [48]:
```
get_results_glove(model, dataloader, index_label):
model.eval()
with open("./data/dev", "r") as f_read, open("dev2.out", "w") as f_write:
    for sentences, is_capitals, lengths, labels in dataloader:
        sentences = sentences.to(device)
        is_capitals = is_capitals.to(device)
        lengths = lengths.to(device)
        output = model(sentences, is_capitals, lengths)
        max_values, max_indices = torch.max(output, dim=2)
        y = max_indices

        for i in range(len(sentences)):
            for j in range(len(sentences[i])):
                read_line = f_read.readline().split()
                if len(read_line) > 0:
                    f_write.write(f"{read_line[0]} {read_line[1]} {index_label[labels[i][j].item()]} {index_l
                else:
                    break
                if j + 1 >= len(sentences[i]):
                    f_read.readline()
            if len(sentences) == batch_size or i < len(sentences) - 1:
                f_write.write("\n")
```

In [49]:
```
embedding_dim = 100
lstm_hidden_dim = 256
lstm_dropout = 0.33
linear_output_dim = 128
num_tags = len(label_index.keys())

bilstm_glove_model = BiLSTMGlove(embedding_dim, lstm_hidden_dim, lstm_dropout, linear_output_dim, num_tags).t
print(bilstm_glove_model)

criterion = nn.CrossEntropyLoss(ignore_index=-1)
optimizer = torch.optim.SGD(bilstm_glove_model.parameters(), lr=0.33)
```

```
BiLSTMGlove(
  (embedding): Embedding(400002, 100, padding_idx=0)
  (lstm): LSTM(101, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=10, bias=True)
)
```

In [*]:
```
epochs = 50
for epoch in range(epochs):
    train_loss = 0.0
    bilstm_glove_model.train()
    for sentences, is_capitals, lengths, labels in train_loader:
        optimizer.zero_grad()
        sentences = sentences.to(device)
        is_capitals = is_capitals.to(device)
        lengths = lengths.to(device)
        labels = labels.to(device)

        output = bilstm_glove_model(sentences, is_capitals, lengths)
        output = output.permute(0, 2, 1)
        loss = criterion(output, labels)

        loss.backward()
        optimizer.step()

        train_loss += loss.item() * sentences.size(0)

    train_loss = train_loss / len(train_dataset)

    print(f'Epoch: {epoch + 1}\tTraining Loss: {train_loss:.6f}')

torch.save(bilstm_glove_model.state_dict(), 'blstm2.pt')

get_results_glove(bilstm_glove_model, dev_loader, index_label)
```

```
Epoch: 1        Training Loss: 0.322834
```

In [*]:
```
get_results_glove(bilstm_glove_model, test_loader, index_label)
```

In [*]:
```
!python eval.py -p dev2.out -g data/dev
```

As we can see the precision, recall and F1 scores are printed above for the dev set.