Python version: 3.11.6

I imported the required libraries.I set the device variable to "cuda" if a GPU is available; otherwise, I sets it to "cpu".

- I loaded the CoNLL 2003 dataset, and to understand the distribution of words in the training data (dataset_train['tokens']), I counted the frequency of each word. Afterward, I created a filtered dictionary called word_freq, containing only those words with a frequency of 3 or more.
- I generated a word-to-index mapping (word2idx). Each word was assigned a unique index, starting from 2. Additionally, I added special tokens, namely '[PAD]' and '[UNK]', to the mapping with indices 0 and 1, respectively.
- I renamed the 'ner_tags' column to 'labels' in each split of the CoNLL 2003 dataset. This adjustment aligns the dataset with the naming convention commonly used in natural language processing tasks.
- For the training split, I implemented a function called get_padded_input_ids_and_labels_train. This function takes a DataFrame as input and processes the 'tokens' and 'labels' columns. It converts the words in 'tokens' to their corresponding indices using the word2idx mapping, creating a tensor for each sample. The input tensors are then padded to the maximum sequence length with zeros, and the labels are similarly processed, ensuring consistent lengths by padding with the label '9'. The resulting padded input IDs and labels are organized into a TensorDataset, and a DataLoader is created for batch processing during training.
- To accommodate the testing and validation splits, I designed a more general function named get_padded_input_ids_and_labels, which accepts a precomputed maximum sequence length as an argument. This function follows a similar process, converting words to indices, padding the input IDs and labels, and creating a DataLoader for batched processing. The dimensions of the padded input IDs and labels are printed for verification.
- I designed a Bidirectional Long Short-Term Memory (BiLSTM) model for natural language processing. The architecture includes an Embedding Layer (embedding_dim=100, vocab_size determined dynamically), a BiLSTM Layer (hidden_dim=256, num_layers=1), a Dropout Layer (dropout=0.33), a Linear Layer (output_dim=128, ELU activation), and a Classifier Layer (num_labels=9). Hyperparameters include num_epochs=20 and initial_lr=0.01. I used AdamW optimizer and linear scheduler with a step size of 5. This setup balances complexity and efficiency for effective sequence processing and classification.
- In each epoch, I loop through the training dataset, clear the GPU cache, and iterate over batches. Within each batch, I perform a forward pass, compute the loss using CrossEntropyLoss, and execute backpropagation to update the model's parameters with the optimizer. The learning rate is adjusted at the start of each epoch using a scheduler. I print the loss at the end of each epoch for monitoring. Clearing the GPU cache helps manage memory.
- In the remove_padding function, I'm creating a list of unpadded predictions by truncating each sequence to the length of its corresponding unpadded label. This ensures that padding doesn't affect the evaluation metrics.
- The evaluate_model function evaluates the model on a dataset. I set the model to evaluation mode, make predictions on the padded input tokens, and convert the predictions and labels to human-readable tag sequences. Then, I use these sequences to calculate precision, recall, and F1-score using the evaluate function.
- These functions are crucial for assessing the performance of a Named Entity Recognition model, particularly when dealing with padded input sequences.

Task 2

Task 2 has similar archiechture and dataset as Task1 with minor modifications

- I created a word-to-index mapping (glove_word2idx) for GloVe embeddings, initializing it with special tokens '[PAD]' and '[UNK]'. I then loaded GloVe embeddings from the 'glove.6B.100d.txt' file, extracting words and their corresponding embeddings. I assigned indices to words, including the special tokens, and built a vocabulary list and an embeddings list. To integrate these embeddings into PyTorch, I converted the lists to NumPy arrays (glove_vocab_npa and glove_embeddings_npa). I added embeddings for '[PAD]' and '[UNK]' at the beginning and calculated a zero vector for '[PAD]' and the mean vector for '[UNK]'. Finally, I created a PyTorch embedding layer (my_embedding_layer) using the GloVe embeddings, ensuring it's frozen to retain pre-trained weights and set a padding index. This layer can be seamlessly incorporated into a neural network for natural language processing tasks.
- To make my Task 2 predictions case sensitive, I implemented a function find_additional_features that takes a dataset (data) and the maximum sequence length (max_sequence_length). For each sequence in the dataset, I extracted additional features for each token, including whether it's title-cased, upper-cased, or lower-cased. I created a list of these features for each sequence, ensuring it matches the maximum sequence length by appending zeros as needed. Finally, I converted the list into a PyTorch tensor (torch.tensor(features_data)) for further use in my NLP model.
- I extended my BiLSTM model to incorporate additional features by concatenating them with the word embeddings. These features include whether a token is title-cased, upper-cased, or lower-cased. I utilized a dedicated embedding layer (my_embedding_layer) trained on pre-trained GloVe embeddings. During the forward pass, I combined the input embeddings with the additional features and passed them through a bidirectional LSTM layer. The resulting sequence was then processed through a linear layer, followed by ELU activation, and finally, a classifier layer to obtain logits for each label. This enhancement allows the model to capture more nuanced information beyond word embeddings alone.

```
import warnings
warnings.filterwarnings('ignore')


# import os
# colab_directory = "/content/gdrive/MyDrive/NLP/Assignments/HW4"
# os.chdir(colab_directory)


!pip install torch
!pip install datasets


!wget https://raw.githubusercontent.com/sighsmile/conlleval/master/conlleval.py


import datasets
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

from conlleval import evaluate
from collections import Counter
import itertools


# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

creating dataset to be trained in batch

```
dataset_train = datasets.load_dataset("conll2003", split = 'train')
dataset_test = datasets.load_dataset("conll2003", split = 'test')
dataset_dev = datasets.load_dataset("conll2003", split = 'validation')
```

```
Downloading builder script: 9.57k/9.57k [00:00<00:00, 176kB/s]
Downloading metadata: 3.73k/3.73k [00:00<00:00, 78.0kB/s]
Downloading readme: 12.3k/12.3k [00:00<00:00, 323kB/s]
Downloading data: 983k/983k [00:00<00:00, 1.81MB/s]
```

```
print(dataset_train)
print(dataset_dev)
print(dataset_test)
```

```
Dataset({
    features: ['id', 'tokens', 'pos_tags', 'chunk_tags', 'ner_tags'],
    num_rows: 14041
})
Dataset({
    features: ['id', 'tokens', 'pos_tags', 'chunk_tags', 'ner_tags'],
    num_rows: 3250
})
Dataset({
    features: ['id', 'tokens', 'pos_tags', 'chunk_tags', 'ner_tags'],
    num_rows: 3453
})
```

Vocab creation to get word id for each token on all samples

```
word_freq = (Counter(itertools.chain(*dataset_train['tokens'])))

#word in tokens for each row in dataset becomes the key and value is the freq
word_freq= {
    word : freq
    for word, freq in word_freq.items() if freq >= 3
}
```

```
word2idx = {
    word: index
    for index, word in enumerate(word_freq.keys(), start = 2)
}

word2idx['[PAD]'] = 0
word2idx['[UNK]'] = 1


from torch.utils.data import TensorDataset, DataLoader


# Rename 'ner_tags' to 'labels' in each split
dataset_train = dataset_train.rename_column('ner_tags', 'labels')
dataset_test = dataset_test.rename_column('ner_tags', 'labels')
dataset_dev = dataset_dev.rename_column('ner_tags', 'labels')


def get_padded_input_ids_and_labels_train(df):
  input_ids = [[word2idx.get(token, word2idx['[UNK]']) for token in sample] for sample in df['tokens']]
  input_ids = [torch.tensor(input_ids_sample) for input_ids_sample in input_ids]
  padded_input_ids = pad_sequence(input_ids, batch_first=True, padding_value=0)

  max_seq_len = max(len(seq) for seq in input_ids)
  label_tensors = [torch.tensor(seq + [9]*(max_seq_len-len(seq))) for seq in df['labels']]
  padded_labels = pad_sequence(label_tensors, batch_first=True, padding_value=9)

  data = TensorDataset(padded_input_ids, padded_labels)
  batch_size = 32
  data_loader = DataLoader(data, batch_size=batch_size, shuffle=True)
  return data_loader, padded_input_ids, max_seq_len



def get_padded_input_ids_and_labels(df, max_seq_len):
  input_ids = [[word2idx.get(token, word2idx['[UNK]']) for token in sample] for sample in df['tokens']]
  input_ids = [torch.tensor(input_ids_sample) for input_ids_sample in input_ids]
  padded_input_ids = pad_sequence(input_ids, batch_first=True, padding_value=0)

  label_tensors = [torch.tensor(seq + [9]*(max_seq_len-len(seq))) for seq in df['labels']]
  padded_labels = pad_sequence(label_tensors, batch_first=True, padding_value=9)

  print(padded_input_ids.shape, padded_labels.shape)
  data = TensorDataset(padded_input_ids, padded_labels)
  batch_size = 32
  data_loader = DataLoader(data, batch_size=batch_size, shuffle=True)
  return data_loader, padded_input_ids



padded_dataset_train, train_input_ids, max_seq_len_train  = get_padded_input_ids_and_labels_train(dataset_train)


padded_dataset_dev, dev_input_ids  = get_padded_input_ids_and_labels(dataset_dev, max_seq_len_train)
padded_dataset_test, test_input_ids  = get_padded_input_ids_and_labels(dataset_test, max_seq_len_train)

     torch.Size([3250, 109]) torch.Size([3250, 113])
     torch.Size([3453, 124]) torch.Size([3453, 124])
```

Task 1 Bi-directional LSTM

```
#Embedding → BiLSTM → Linear → ELU → classifier
class BiLSTM(nn.Module):
  def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, num_layers, dropout, num_labels):
    super(BiLSTM, self).__init__()
    self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
    self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=num_layers, bidirectional=True, batch_first=True)
    self.dropout = nn.Dropout()
    self.linear = nn.Linear(2*hidden_dim, output_dim)
    self.elu = nn.ELU()
    self.classifier = nn.Linear(output_dim, num_labels)

  def forward(self, input):
    embeddings = (self.embedding(input))
    x, _ = self.lstm(embeddings)
    x = self.dropout(x)
    x = self.linear(x)
    x = self.elu(x)
    logits = self.classifier(x)
    return logits
```

```
# Define hyperparameters
embedding_dim = 100
hidden_dim = 256
output_dim = 128
num_layers = 1
dropout = 0.33
vocab_size = len(word2idx) # Adjust based on your vocabulary size
num_epochs = 20
initial_lr = 0.01
num_labels = 9


model = BiLSTM(vocab_size, embedding_dim, hidden_dim, output_dim, num_layers, dropout, num_labels)
model.to(device)

criterion = nn.CrossEntropyLoss(ignore_index=9)
optimizer = optim.AdamW(model.parameters(), lr=initial_lr)


step_size = 5
scheduler = StepLR(optimizer=optimizer, step_size=step_size, gamma = 0.1)


#Set the model in training mode
model.train()
```

```
BiLSTM(
  (embedding): Embedding(8128, 100, padding_idx=0)
  (lstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)
```

```
for epoch in range(num_epochs):

    torch.cuda.empty_cache()
    for batch in padded_dataset_train:
        optimizer.zero_grad()
        # Extract input_ids and ner_tags from the batch
        input_ids , labels = batch

        input_ids = input_ids.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(input_ids)
        outputs = outputs.permute(0, 2, 1)
        # Compute the loss
        loss = criterion(outputs, labels)
        # Backpropagation
        loss.backward()
        optimizer.step()

    # Update the learning rate at the beginning of each epoch
    scheduler.step()

    # Optionally clear GPU cache at the end of each epoch to manage memory
    print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss}")
    torch.cuda.empty_cache()
```

```
Epoch 1/20, Loss: 0.12485282868146896
Epoch 2/20, Loss: 0.09035669267177582
Epoch 3/20, Loss: 0.08609665930271149
Epoch 4/20, Loss: 0.07817414402961731
Epoch 5/20, Loss: 0.07878076285123825
Epoch 6/20, Loss: 0.025946995243430138
Epoch 7/20, Loss: 0.025206446647644043
Epoch 8/20, Loss: 0.012350094504654408
Epoch 9/20, Loss: 0.021025974303483963
Epoch 10/20, Loss: 0.009655218571424484
Epoch 11/20, Loss: 0.019309507682919502
Epoch 12/20, Loss: 0.0007442826754413545
Epoch 13/20, Loss: 0.0038777277804911137
Epoch 14/20, Loss: 0.023904113098978996
Epoch 15/20, Loss: 0.002769579878076911
Epoch 16/20, Loss: 0.020701922476291656
Epoch 17/20, Loss: 0.012106637470424175
Epoch 18/20, Loss: 0.03672022372484207
```

```
Epoch 19/20, Loss: 0.00011164149327669293
Epoch 20/20, Loss: 0.017551179975271225
```

```
torch.save(model.state_dict(), 'bilstm1_state_dict.pt')
```

```
def remove_padding(preds, unpadded_labels):
    unpadded_preds = []
    for i in range(len(unpadded_labels)):
        unpadded_preds.append(preds[i][:len(unpadded_labels[i])])
    unpadded_preds = [pred.tolist() for pred in unpadded_preds]

    return unpadded_preds
```

```
#input tokens are padded
#dataset_dev
def evaluate_model(model, padded_input_tokens, dataset):
    model.eval()

    with torch.no_grad():
        padded_input_tokens = padded_input_tokens.to(device)
        predictions = model(padded_input_tokens)
        predictions = torch.argmax(predictions, dim=-1)

    unpadded_preds = remove_padding(predictions, dataset['labels'])


    idx2tag = {0: 'O', 1: 'B-PER', 2: 'I-PER', 3: 'B-ORG', 4: 'I-ORG', 5: 'B-LOC', 6: 'I-LOC', 7: 'B-MISC', 8: 'I-MISC'}
    labels = [list(map(idx2tag.get, labels)) for labels in dataset['labels']]
    preds_string = [list(map(idx2tag.get, labels)) for labels in unpadded_preds]

    precision, recall, f1 = evaluate(itertools.chain(*labels), itertools.chain(*preds_string))

    return precision, recall, f1
```

```
precision, recall, f1 = evaluate_model(model, dev_input_ids, dataset_dev)
```

```
processed 51362 tokens with 5942 phrases; found: 5780 phrases; correct: 4761.
accuracy:  82.49%; (non-O)
accuracy:  96.27%; precision:  82.37%; recall:  80.12%; FB1:  81.23
              LOC: precision:  89.68%; recall:  86.12%; FB1:  87.86  1764
             MISC: precision:  78.88%; recall:  76.57%; FB1:  77.71  895
              ORG: precision:  73.85%; recall:  74.35%; FB1:  74.10  1350
              PER: precision:  83.34%; recall:  80.13%; FB1:  81.70  1771
Precision for vaolidation: 82.37%
Recall: 80.12%
F1-Score: 81.23%
```

What are the precision, recall, and F1 score on the validation data?

Precision for validation data: 82.37% Recall for validation data: 80.12% F1-Score for validation data: 81.23%

Task 1 test dataset

```
precision, recall, f1 = evaluate_model(model, test_input_ids, dataset_test)
```

```
processed 46435 tokens with 5648 phrases; found: 5427 phrases; correct: 4027.
accuracy:  75.54%; (non-O)
accuracy:  94.21%; precision:  74.20%; recall:  71.30%; FB1:  72.72
              LOC: precision:  83.29%; recall:  78.60%; FB1:  80.88  1574
             MISC: precision:  64.05%; recall:  67.52%; FB1:  65.74  740
              ORG: precision:  68.46%; recall:  67.55%; FB1:  68.00  1639
              PER: precision:  75.98%; recall:  69.26%; FB1:  72.47  1474
Precision for vaolidation: 74.20%
Recall: 71.30%
F1-Score: 72.72%
```

What are the precision, recall, and F1 score on the test data?

Precision for test data: 74.20% Recall for test data: 71.30% F1-Score for test data: 72.72%

Task 2 with GloVe Embeddings

```
glove_word2idx = {}
glove_word2idx['[PAD]'] = 0
glove_word2idx['[UNK]'] = 1

glove_vocab, glove_embeddings = [], []
with open('glove.6B.100d/glove.6B.100d.txt', 'rt') as f:
  all_file_embeddings = f.read().strip().split('\n')

for i in range(len(all_file_embeddings)):
  glove_word = all_file_embeddings[i].split(' ')[0] #tokenizing
  glove_embed = [float(x) for x in all_file_embeddings[i].split(' ')[1:]] #read or store each embedding as a float
  glove_word2idx[glove_word] = i+2
  glove_vocab.append(glove_word)
  glove_embeddings.append(glove_embed)

import numpy as np
glove_vocab_npa = np.array(glove_vocab)
glove_embeddings_npa = np.array(glove_embeddings)

#insert '<pad>' and '<unk>' tokens at start of vocab_npa.
glove_vocab_npa = np.insert(glove_vocab_npa, 0, '[PAD]')
glove_vocab_npa = np.insert(glove_vocab_npa, 1, '[UNK]')


pad_emb_npa = np.zeros((1,glove_embeddings_npa.shape[1]))   #embedding for '<pad>' token.
unk_emb_npa = np.mean(glove_embeddings_npa,axis=0,keepdims=True)    #embedding for '<unk>' token

#insert embeddings for pad and unk tokens at top of embs_npa.
glove_embeddings_npa = np.vstack((pad_emb_npa,unk_emb_npa,glove_embeddings_npa))

my_embedding_layer = torch.nn.Embedding.from_pretrained(torch.from_numpy(glove_embeddings_npa).float(), freeze = True, padding_idx=0)


max_seq_length_train_glove = max(len(sequence) for sequence in dataset_train['tokens'])
max_seq_length_test_glove = max(len(sequence) for sequence in dataset_test['tokens'])
max_seq_length_dev_glove = max(len(sequence) for sequence in dataset_dev['tokens'])


def find_additional_features(data, max_sequence_length):
    features_data = []
    for sequence in data:
        add_features = [[float(token.istitle()), float(token.isupper()), float(token.islower())] for token in sequence]
        add_features += [[0.0, 0.0, 0.0]] * (max_sequence_length - len(add_features))
        features_data.append(add_features)
    return torch.tensor(features_data)


addn_features = find_additional_features(dataset_train['tokens'],max_seq_length_train_glove)

def get_padded_input_ids_and_labels_glove(df, add_feats, max_seq_len):
  input_ids = [[glove_word2idx.get(token.lower(), glove_word2idx['[UNK]']) for token in sample] for sample in df['tokens']]
  input_ids = [torch.tensor(input_ids_sample) for input_ids_sample in input_ids]
  padded_input_ids = pad_sequence(input_ids, batch_first=True, padding_value=0)

  label_tensors = [torch.tensor(seq + [9]*(max_seq_len-len(seq))) for seq in df['labels']]
  padded_labels = pad_sequence(label_tensors, batch_first=True, padding_value=9)

  #print(padded_input_ids.shape, padded_labels.shape)
  data = TensorDataset(padded_input_ids, add_feats, padded_labels)
  data_loader = DataLoader(data, batch_size=32, shuffle=True)
  return data_loader, padded_input_ids


train_data_loader_glove, train_data_padded_input_ids = get_padded_input_ids_and_labels_glove(dataset_train, addn_features, max_seq_length
```

BiLSTM_Glove

```
#Embedding -> BiLSTM -> Linear -> ELU -> classifier
class BiLSTM_glove(nn.Module):
  def __init__(self, embedding_dim, hidden_dim,
               output_dim, num_layers, dropout, num_labels):
    super(BiLSTM_glove, self).__init__()
    self.embedding = my_embedding_layer
    self.lstm = nn.LSTM(embedding_dim+3, hidden_dim, num_layers=num_layers, bidirectional=True, batch_first=True)
    self.dropout = nn.Dropout(0.33)
    self.linear = nn.Linear(2*hidden_dim, output_dim)
    self.elu = nn.ELU()
    self.classifier = nn.Linear(output_dim, num_labels)
```

```
  def forward(self, input, add_features):
    embeddings = (self.embedding(input))
    add_features = add_features.to(embeddings.device)
    embeddings_with_add_features = torch.cat((embeddings, add_features), dim= 2)
    x, _ = self.lstm(embeddings_with_add_features)
    x = self.dropout(x)
    x = self.linear(x)
    x = self.elu(x)
    logits = self.classifier(x)
    return logits


# Define hyperparameters
embedding_dim = 100
hidden_dim = 256
output_dim = 128
num_layers = 1
dropout = 0.33
num_labels = 9
num_epochs = 20
initial_lr = 0.01


model_glove = BiLSTM_glove(embedding_dim, hidden_dim, output_dim, num_layers, dropout, num_labels)
model_glove.to(device)

criterion_glove = nn.CrossEntropyLoss(ignore_index = 9)
optimizer_glove = optim.AdamW(model_glove.parameters(), lr=initial_lr)


step_size_glove = 5
scheduler_glove = StepLR(optimizer=optimizer_glove, step_size=step_size_glove, gamma = 0.1)


# Set the model in training mode
model_glove.train()

for epoch in range(num_epochs):
    torch.cuda.empty_cache()

    for batch in train_data_loader_glove:
        optimizer_glove.zero_grad()
        # Extract input_ids and ner_tags from the batch
        input_ids, add_features, labels = batch


        input_ids = input_ids.to(device)
        add_features = add_features.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model_glove(input_ids, add_features)
        outputs = outputs.permute(0,2,1)
        # Compute the loss
        loss = criterion_glove(outputs, labels)
        # Backpropagation
        loss.backward()
        optimizer_glove.step()

    # Update the learning rate at the beginning of each epoch
    scheduler_glove.step()

    # Optionally clear GPU cache at the end of each epoch to manage memory
    print(f"epoch {epoch+1}/{num_epochs}: loss is {loss}")
    torch.cuda.empty_cache()

     epoch 1/20: loss is 0.03337996453046799
     epoch 2/20: loss is 0.12078648060560226
     epoch 3/20: loss is 0.08091679215431213
     epoch 4/20: loss is 0.013401861302554607
     epoch 5/20: loss is 0.050898414105176926
     epoch 6/20: loss is 0.05137806758284569
     epoch 7/20: loss is 0.00881861336529255
     epoch 8/20: loss is 0.013339130207896233
     epoch 9/20: loss is 0.01268848218023777
     epoch 10/20: loss is 0.00479291332885623
     epoch 11/20: loss is 0.032377131283283234
     epoch 12/20: loss is 0.0015617531025782228
     epoch 13/20: loss is 0.0008879965753294528
     epoch 14/20: loss is 0.0006026356131769717
     epoch 15/20: loss is 0.004186670761555433
     epoch 16/20: loss is 0.0019996350165456533
     epoch 17/20: loss is 0.00022307844483293593
     epoch 18/20: loss is 0.0021445825695991516
     epoch 19/20: loss is 0.0010260352864861488
     epoch 20/20: loss is 0.0008403216488659382
```

```
torch.save(model_glove.state_dict(), 'bilstm2_state_dict.pt')
```

Task 2 Validation dataset

```
dev_addn_features = find_additional_features(dataset_dev['tokens'], max_seq_length_dev_glove)
test_addn_features = find_additional_features(dataset_test['tokens'], max_seq_length_test_glove)


dev_data_loader_glove, dev_input_ids_glove = get_padded_input_ids_and_labels_glove(dataset_dev, dev_addn_features, max_seq_length_dev_glo
test_data_loader_glove, test_input_ids_glove = get_padded_input_ids_and_labels_glove(dataset_test, test_addn_features, max_seq_length_tes


def remove_padding(preds, unpadded_labels):
    unpadded_preds = []
    for i in range(len(unpadded_labels)):
        unpadded_preds.append(preds[i][:len(unpadded_labels[i])])

    unpadded_preds = [pred.tolist() for pred in unpadded_preds]

    return unpadded_preds


#input tokens are padded
#dataset_dev
def evaluate_model_glove(model, padded_input_tokens, dataset, add_features):
    model.eval()

    with torch.no_grad():
        padded_input_tokens = padded_input_tokens.to(device)
        predictions = model(padded_input_tokens, add_features)
        predictions = torch.argmax(predictions, dim=-1)

    unpadded_preds = remove_padding(predictions, dataset['labels'])


    idx2tag = {0: 'O', 1: 'B-PER', 2: 'I-PER', 3: 'B-ORG', 4: 'I-ORG', 5: 'B-LOC', 6: 'I-LOC', 7: 'B-MISC', 8: 'I-MISC'}
    labels = [list(map(idx2tag.get, labels)) for labels in dataset['labels']]
    preds_string = [list(map(idx2tag.get, labels)) for labels in unpadded_preds]

    precision, recall, f1 = evaluate(itertools.chain(*labels), itertools.chain(*preds_string))

    return precision, recall, f1


precision, recall, f1 = evaluate_model_glove(model_glove, dev_input_ids_glove, dataset_dev, dev_addn_features)
```

```
 processed 51362 tokens with 5942 phrases; found: 6037 phrases; correct: 5552.
 accuracy:  93.70%; (non-O)
 accuracy:  98.76%; precision:  91.97%; recall:  93.44%; FB1:  92.70
              LOC: precision:  95.21%; recall:  96.35%; FB1:  95.78  1859
             MISC: precision:  86.41%; recall:  86.88%; FB1:  86.64  927
              ORG: precision:  87.87%; recall:  89.11%; FB1:  88.49  1360
              PER: precision:  94.45%; recall:  96.96%; FB1:  95.69  1891
 Precision for vaolidation: 91.97%
 Recall: 93.44%
 F1-Score: 92.70%
```

What are the precision, recall, and F1 score on the validation data?

Precision for validation data: 91.97% Recall for validation data: 93.44% F1-Score for validation data: 92.70%

Task 2 Test Dataset

```
precision, recall, f1 = evaluate_model_glove(model_glove, test_input_ids_glove, dataset_test, test_addn_features)
```

```
 processed 46435 tokens with 5648 phrases; found: 5813 phrases; correct: 5040.
 accuracy:  90.43%; (non-O)
 accuracy:  97.69%; precision:  86.70%; recall:  89.24%; FB1:  87.95
              LOC: precision:  89.95%; recall:  92.81%; FB1:  91.35  1721
             MISC: precision:  73.77%; recall:  80.91%; FB1:  77.17  770
              ORG: precision:  82.89%; recall:  85.43%; FB1:  84.14  1712
              PER: precision:  93.48%; recall:  93.07%; FB1:  93.28  1610
 Precision for vaolidation: 86.70%
```

```
Recall: 89.24%
F1-Score: 87.95%
```

What are the precision, recall, and F1 score on the test data?

Precision for test dataset: 86.70% Recall for test dataset: 89.24% F1-Score for test dataset: 87.95%

BiLSTM with Glove Embeddings outperforms the model without. Can you provide a rationale for this?

I think that GloVe performs better owing to having access to huge and informative dataset to create its embeddings. These embeddings might have a rich information of positional semantics. GloVe embeddings encode semantic similarities between words, allowing the model to grasp subtle nuances and meaning in the language. This can be crucial for tasks where understanding context is essential. Also, GloVe embeddings often have lower-dimensional representations compared to training embeddings from scratch. This reduction in dimensionality can facilitate faster training and improved generalization, especially when working with limited data. Also, I feel that the pre-training of GloVe on extensive datasets enables the model to generalize well across various domains. It brings a broader understanding of language, making it adaptable to different contexts and tasks.