CSCI544 - Homework Assignment No 4

Precision, Recall and F1 score (dev data)

Task 1: Simple Bidirectional LSTM model

```
D:\University of Southern California\CSCI544 APPLIED NATURAL LANGUAGE PROCESSING\HW4\HW4-Advait Hemant-Naik\code>perl conll03eval.txt < dev1.out
processed 51578 tokens with 5942 phrases; found: 5183 phrases; correct: 4309.
accuracy:  95.47%; precision:  83.14%; recall:  72.52%; FB1:  77.47
              LOC: precision:  92.79%; recall:  79.15%; FB1:  85.43  1567
             MISC: precision:  87.50%; recall:  76.68%; FB1:  81.73  808
              ORG: precision:  74.17%; recall:  68.31%; FB1:  71.12  1235
              PER: precision:  78.32%; recall:  66.88%; FB1:  72.15  1573
```

Task 2: Using GloVe word embeddings

```
D:\University of Southern California\CSCI544 APPLIED NATURAL LANGUAGE PROCESSING\HW4\HW4-Advait Hemant-Naik\code>perl conll03eval.txt < dev2.out
processed 51578 tokens with 5942 phrases; found: 6048 phrases; correct: 5422.
accuracy:  98.40%; precision:  89.65%; recall:  91.25%; FB1:  90.44
              LOC: precision:  91.97%; recall:  96.03%; FB1:  93.95  1918
             MISC: precision:  82.17%; recall:  85.47%; FB1:  83.79  959
              ORG: precision:  83.82%; recall:  82.70%; FB1:  83.26  1323
              PER: precision:  95.29%; recall:  95.60%; FB1:  95.45  1848
```

Description

Task 1: Simple Bidirectional LSTM model

import numpy as np import torch import torch.nn as nn import torch.optim as optim import torch.utils.data as data from torch.utils.data import DataLoader from torch.nn.utils.rnn import pad_sequence from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence, pad_packed_sequence
```

WORD_TO_INDEX - dictionary that maps words to their corresponding indices in the vocabulary.
TAG_TO_INDEX - dictionary that maps tags to their corresponding indices.

TRAIN_PATH = '../data/train' DEV_PATH = '../data/dev' TEST_PATH = '../data/test'

WORD_TO_INDEX = dict() TAG_TO_INDEX = dict()

Functions for loading and processing text data in a format to train a model for an NLP task.

read_data - function reads in raw data from a file at the given path and passes it to the data_preparation function to convert it into a list of sentences.

data_preparation - function takes a list of strings, where each string represents a line in the dataset, and converts it into a list of sentences, where each sentence is a list of tuples containing a word and its corresponding tag. The function returns this list of sentences.

```
def data_preparation(input_data: list[str]) -> list:
    """
    read data line and convert line in format [[word1, word2, word3 ..., wordn], [tag1,
tag2, tag3, ..., tagn]]
    :param input_data:
    :return:
    """
    sentence = []
    label = []
    data = []
    for line in input_data:
        if not line.isspace():
            part = line.split()
            sentence.append(part[1])
            label.append(part[-1])
        if line.isspace():
            data.append([sentence, label])
            sentence = []
            label = []
    # print(line)
    sentence = []
    label = []
    part = line.split()
    sentence.append(part[1])
    label.append(part[-1])
    data.append([sentence, label])
    return data

def read_data(path: str) -> str:
    """
    read raw data
    :param path: raw data file path
    :return:
    """
    with open(path, 'r', encoding='utf-8') as file:
        input_data = file.readlines()
    # print(data)
    data = data_preparation(input_data)
    return data
```

Functions for converting the text data into a numerical form that can be used as input to model.

vocab_create creates two dictionaries: **WORD_TO_INDEX** and **TAG_TO_INDEX**. These dictionaries map words and tags to their corresponding numerical indices, respectively. The function takes a list of sentences train_data as input, where each sentence is a list of tuples containing a word and its corresponding tag.

The function first initializes TAG_TO_INDEX to a pre-defined dictionary where each tag is mapped to an integer. This dictionary is based on the CoNLL-2003 dataset. Next, the function iterates over each sentence in train_data and updates the global VOCAB set with all unique words in the dataset. The function then creates WORD_TO_INDEX by mapping each word to a unique index, starting from 2. Two special tokens, **< pad >** and **< unk >**, are also added to WORD_TO_INDEX and assigned indices 0 and 1, respectively. < pad > is used to pad sequences to a fixed length, and < unk > is used to represent words that are not in VOCAB.

```
def vocab_create1(train_data: list) -> None:
    """
    create WORD_TO_INDEX, TAG_TO_INDEX file
    convert the data into numerical form
    :param train_data: train read data file
    :return:
    """
    global VOCAB, WORD_TO_INDEX, TAG_TO_INDEX
    TAG_TO_INDEX = {'O': 0, 'B-MISC': 1, 'I-MISC': 2, 'B-PER': 3, 'I-PER': 4, 'B-ORG':
5, 'I-ORG': 6, 'B-LOC': 7, 'I-LOC': 8}

    for sentence, _ in train_data:
        for word in sentence:
            if word not in WORD_TO_INDEX:
                WORD_TO_INDEX[word] = index
                index += 1

    WORD_TO_INDEX['<pad>'] = 0
    WORD_TO_INDEX['<unk>'] = 1
```

Class to convert the words and labels in the training data from their original string form into numerical form.

The Encode class has two methods:
Encode_Process: This method takes a sentence and its corresponding label as input and encodes them into numerical form using the **WORD_TO_INDEX** and **TAG_TO_INDEX** dictionaries, respectively. The WORD_TO_INDEX dictionary is used to convert each word in the sentence into its corresponding index, and the TAG_TO_INDEX dictionary is used to convert each label in the sentence into its corresponding index. If a word is not in the WORD_TO_INDEX dictionary, it is assigned the index 1, which corresponds to the < unk > token.

Encode_Vocab: This method takes a list of preprocessed data as input, where each element of the list containing a list of words and a list of labels. It applies the Encode_Process method to each sentence and label in the data list, and returns the encoded data as a list of tuples containing a list of word indices and a list of tag indices for each sentence in the dataset.

```
class Encode1:
    def Encode_Process(self, sentence, label):
        """
        Encode the sentence and labels into numerical form
        """
        sentence = [WORD_TO_INDEX.get(word, 1) for word in sentence]
        label = [TAG_TO_INDEX[lab] for lab in label]
        return sentence, label

    def Encode_Vocab(self, data):
        data = [self.Encode_Process(sentence, label) for sentence, label in data]
        return data
```

```
train_data = read_data(TRAIN_PATH)
vocab_create1(train_data)

encode = Encode1()
train_data = encode.Encode_Vocab(train_data)
```

```
INDEX_TO_TAG = {index: tag for tag, index in TAG_TO_INDEX.items()}
INDEX_TO_WORD = {v: k for k, v in WORD_TO_INDEX.items()}
```

NERDataset - PyTorch Dataset class for the Named Entity Recognition (NER) task. It takes in a list of preprocessed data, where each item is a tuple of a list of encoded sentence and a list of encoded labels. The **getitem** method returns the encoded sentence and label at the given index as PyTorch tensors. The **len** method returns the length of the dataset, i.e., the number of examples in the data.

```
class NERDataset1(data.Dataset):
    def __init__(self, data):
        self.data = data

    def __getitem__(self, index):
        sentence, label = self.data[index]
        sentence = torch.LongTensor(sentence)

        label = torch.LongTensor(label)
        return sentence, label

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

The BLSTM1 class defines a neural network model architecture for Named Entity Recognition (NER).

input_size - size of the vocabulary, which is the number of unique words in the input dataset
output_size - number of unique tags in the output dataset
embedding_dim - size of the word embedding
lstm_hidden_dim - number of hidden units in the LSTM layer
linear_output_dim - number of output units in the linear layers
num_layers - number of LSTM layers in the model
dropout - dropout probability

The model architecture consists of an Embedding layer that converts the input words to vectors, a bidirectional LSTM layer that processes the input sequence and produces output at each time step, followed by two Linear layers with ELU activation function, and a final Linear layer with softmax activation function. The input sequence is first packed and then processed by the LSTM layer to avoid processing padded tokens. The dropout layer is applied to the output of the LSTM layer to prevent overfitting.

```
class BLSTM1(nn.Module):
    def __init__(self, input_size, output_size, embedding_dim, lstm_hidden_dim,
linear_output_dim, num_layers, dropout):
        super(BLSTM1, self).__init__()

        self.embedding = nn.Embedding(input_size, embedding_dim) #padding_idx=0
```

```
        self.blstm = nn.LSTM(embedding_dim, lstm_hidden_dim, num_layers=num_layers,
bidirectional=True) #dropout=dropout
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(lstm_hidden_dim * 2, linear_output_dim)
        self.activation = nn.ELU()
        self.classifier = nn.Linear(linear_output_dim, output_size)

    def forward(self, sentence):
        embedded = self.embedding(sentence)

        length = torch.sum(sentence != 0, dim=1)
        packed_output = pack_padded_sequence(embedded, length, batch_first=True,
enforce_sorted=False)
        blstm_output, _ = self.blstm(packed_output)
        unpacked_output, _ = pad_packed_sequence(blstm_output, batch_first=True)

        blstm_output = self.dropout(unpacked_output)

        linear_output = self.fc(blstm_output)
        elu_output = self.activation(linear_output)

        output = self.classifier(elu_output)
        return output
```

The **collate_fn** function pads the sequences to the same length, and the **DataLoader** batches the data and shuffles it during training.

```
def collate_fn1(batch):
    # batch = sorted(batch, key=lambda x: len(x[0]), reverse=True)
    sentence, label = zip(*batch)
    sentence = pad_sequence(sentence, batch_first=True, padding_value=0)
    label = pad_sequence(label, batch_first=True, padding_value=-1)
    return sentence, label

batch_size = 64
train_dataset = NERDataset1(train_data)
train_loader = DataLoader(train_dataset,
                          batch_size=batch_size,
                          shuffle=False,
                          collate_fn=collate_fn1)
```

Hyper-parameters

input size = len(WORD_TO_INDEX)
output size = len(TAG_TO_INDEX)
embedding dimension = 101
lstm hidden dimension = 256
linear output dimension = 128
number of LSTM layers = 1
dropout = 0.33
learning rate = 0.5
number of epochs = 10

The **CrossEntropyLoss** function is used as the loss function, along with an optimizer **SGD** with momentum=0.9.
weight - The second argument to CrossEntropyLoss is the weight for each class. This is used to give more

importance to certain classes while computing the loss. In this case, class 0 ['O'] is given a weight of 0.7 and all other classes are given a weight of 1.
ignore_index is used to ignore the padded values (-1) while computing the loss.

```
input_size = len(WORD_TO_INDEX)
output_size = len(TAG_TO_INDEX)
embedding_dim = 100
lstm_hidden_dim = 256
linear_output_dim = 128
num_layers = 1
dropout = 0.33
lr = 0.1
num_epochs = 20
```

```
model1 = BLSTM1(input_size, output_size, embedding_dim, lstm_hidden_dim,
linear_output_dim, num_layers, dropout)
print(model1)
optimizer = optim.SGD(model1.parameters(), lr=lr, momentum=0.9)
class_weights = torch.tensor([0.7, 1, 1, 1, 1, 1, 1, 1, 1])
ignore_index = -1
loss_fn = nn.CrossEntropyLoss(weight=class_weights, ignore_index=ignore_index)
```

Model_Train() - trains the model for a given number of epochs using the provided data loader, loss function and optimizer.

```
def Model_Train():
    for epoch in range(num_epochs):
        running_loss = 0
        model1.train()
        for sentence, label in train_loader:
            # print(sentence.shape, label.shape)
            optimizer.zero_grad()
            output = model1(sentence)
            output = output.permute(0, 2, 1)
            # print(output.shape, label.shape)
            loss = loss_fn(output, label)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print('Epoch %d \t Training Loss: %.6f' % (epoch+1,
running_loss/len(train_loader)))
```

```
Model_Train()
```

Model_Evaluate() -

- The code is evaluating the trained model on the development dataset and writing the output to a file.
- It uses a dataloader to iterate over the sentences in the development dataset, feeds them to the trained model, and writes the predicted tags for each word in the sentence along with the original word and tag to an output file.
- The output is formatted in a way that can be compared to the original development dataset to evaluate the performance of the model.

```
dev_data = read_data(DEV_PATH)
encode = Encode1()
```

```
dev_dataset = encode.Encode_Vocab(dev_data)
dev_dataset = NERDataset1(dev_dataset)
dev_loader = DataLoader(dev_dataset, batch_size=1, shuffle=False)
```

```
def Model_Evaluate(output_file):
    model1.eval()


    with open(output_file, "w") as f:
        for i, (sentence, label) in enumerate(dev_loader):
            output = model1(sentence)
            predicted = output.argmax(dim = -1)
            index = 0
            for j in range(len(sentence[0])):
                index = index + 1
                word = dev_data[i][0][j]
                original_tag = INDEX_TO_TAG[label[0][j].item()]
                predicted_tag = INDEX_TO_TAG[predicted[0][j].item()]
                f.write("{} {} {}\n".format(index, word, predicted_tag))
            if (i != len(dev_loader)-1): f.write("\n")

DEV_OUT = 'dev1.out'
Model_evaluate(DEV_OUT)
```

```
MODEL_PATH = 'blstm1.pt'

# Model Saving-
torch.save(model1, MODEL_PATH)

# Model Loading:
model1 = torch.load(MODEL_PATH)
model1.eval()
```

Model_Predict()

- The function first sets the model to evaluation mode using model2.eval(). It then iterates over each batch of data in the test_loader and passes the data through the model using model1(sentence, capitalization). The model's output is then converted into predicted labels using output.argmax(dim=-1).
- For each sentence in the batch, the function writes the predicted tag for each word to the specified output file. The output file is opened using the with statement, which ensures that the file is closed properly even if an error occurs during the writing process.
- Finally, the function is called with the argument test1.out, which specifies the output file path.

```
test_data = read_data_test(test_path)
encode = Encode1()
test_dataset = encode.Encode_Vocab(test_data)
test_dataset = NERDataset1(test_dataset)
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)
```

```
def Model_Predict(output_file):
    model2.eval()

    with open(output_file, "w") as f:
        for i, (sentence, label) in enumerate(test_loader):
            output = model1(sentence)
            predicted = output.argmax(dim = -1)
```

```
            index = 0
            for j in range(len(sentence[0])):
                index = index + 1
                word = test_data[i][0][j]
                predicted_tag = INDEX_TO_TAG[predicted[0][j].item()]
                f.write("{} {} {}\n".format(index, word, predicted_tag))
            if (i != len(test_loader)-1): f.write("\n")

TEST_OUT = 'test2.out'
Model_Predict(TEST_OUT)
```

Task 2: Using GloVe word embeddings

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence, pad_packed_sequence
```

VOCAB - set that will contain all unique words in the dataset.
WORD_TO_INDEX_GLOVE - dictionary that maps words to their corresponding indices in the vocabulary.
TAG_TO_INDEX - dictionary that maps tags to their corresponding indices.

```
TRAIN_PATH = '../data/train'
DEV_PATH = '../data/dev'
TEST_PATH = '../data/test'

GLOVE_VECTOR = dict()
WORD_TO_INDEX_GLOVE = dict()
INDEX_TO_WORD_GLOVE = dict()
TAG_TO_INDEX = dict()
INDEX_TO_TAG = dict()
```

Functions for loading and processing text data in a format to train a model for an NLP task.

read_data - function reads in raw data from a file at the given path and passes it to the data_preparation function to convert it into a list of sentences.

data_preparation - function takes a list of strings, where each string represents a line in the dataset, and converts it into a list of sentences, where each sentence is a list of tuples containing a word and its corresponding tag. The function returns this list of sentences.

```
def data_preparation(input_data: list[str]) -> list:
    """
    read data line and convert line in format [[word1, word2, word3 ..., wordn], [tag1,
tag2, tag3, ..., tagn]]
    :param input_data:
    :return:
    """
    sentence = []
    label = []
    data = []
    for line in input_data:
        if not line.isspace():
            part = line.split()
            sentence.append(part[1])
            label.append(part[2])
        if line.isspace():
            data.append([sentence, label])
            sentence = []
            label = []
```

```
    # print(line)
    sentence = []
    label = []
    part = line.split()
    sentence.append(part[1])
    label.append(part[2])
    data.append([sentence, label])
    return data

def read_data(path: str) -> str:
    """
    read raw data
    :param path: raw data file path
    :return:
    """
    with open(path, 'r', encoding='utf-8') as file:
        input_data = file.readlines()
    data = data_preparation(input_data)
    return data
```

Functions for converting the text data into a numerical form that can be used as input to model.

Function creates three global variables **WORD_TO_INDEX_GLOVE**, **GLOVE_VECTOR**, and **INDEX_TO_WORD_GLOVE** and populates them with values from the pre-trained GloVe embeddings file **glove.6B.100d.txt**.

The function first initializes the WORD_TO_INDEX_GLOVE dictionary with special tokens **< pad >** and **< unk >** and assigns them index 0 and 1 respectively.

It then reads the GloVe embeddings file line by line using a with statement and splits each line into a word and its associated vector. The vector is converted to a PyTorch tensor and stored in the GLOVE_VECTOR dictionary with the word as the key. The corresponding index of the word in the embeddings file plus 2 is assigned as the value of the WORD_TO_INDEX_GLOVE dictionary for the word. This ensures that the indices of the GloVe embeddings are shifted by 2 to account for the special tokens.

After all the vectors and indices have been assigned, the function adds special token embeddings to the GLOVE_VECTOR dictionary. The < pad > token is assigned a tensor of zeros, while the < unk > token is assigned the mean of all the other embeddings.

Finally, the INDEX_TO_WORD_GLOVE dictionary is created by reversing the key-value pairs in the WORD_TO_INDEX_GLOVE dictionary. The resulting dictionary maps indices to words in the GloVe embeddings.

```
def vocab_create(train_data: list) -> None:
    """
    create WORD_TO_INDEX_GLOVE, GLOVE_VECTOR file
    convert the data into numerical form
    :param train_data: train read data file
    :return:
    """
    global WORD_TO_INDEX_GLOVE, GLOVE_VECTOR, INDEX_TO_WORD_GLOVE

    WORD_TO_INDEX_GLOVE['<pad>'] = 0
    WORD_TO_INDEX_GLOVE['<unk>'] = 1

    with open('glove.6B.100d.txt', 'r', encoding='utf8') as f:
```

```
        for i, line in enumerate(f):
            line = line.strip().split()
            word = line[0]
            vector = torch.FloatTensor([float(val) for val in line[1:]])
            GLOVE_VECTOR[word] = vector
            WORD_TO_INDEX_GLOVE[word] = i+2

    GLOVE_VECTOR['<pad>'] = torch.zeros(100)
    tensor_list = list(GLOVE_VECTOR.values())
    GLOVE_VECTOR['<unk>'] = torch.mean(torch.stack(tensor_list), dim=0)

    INDEX_TO_WORD_GLOVE = {v: k for k, v in WORD_TO_INDEX_GLOVE.items()}
```

TAG_TO_INDEX - dictionary that maps named entity tags to their corresponding index numbers. The keys are strings representing the named entity tags ('O', 'B-MISC', 'I-MISC', 'B-PER', 'I-PER', 'B-ORG', 'I-ORG', 'B-LOC', 'I-LOC') and the values are integer indices (0-8).

INDEX_TO_TAG - dictionary that maps index numbers back to their corresponding named entity tags. The keys are the integer indices and the values are the named entity tags.

The mappings are used to convert the named entity tags in the training data into numerical form, and then convert the predicted numerical tags back into their corresponding named entity tags for evaluation.

```
TAG_TO_INDEX = {'O': 0, 'B-MISC': 1, 'I-MISC': 2, 'B-PER': 3, 'I-PER': 4, 'B-ORG': 5,
'I-ORG': 6, 'B-LOC': 7, 'I-LOC': 8}
INDEX_TO_TAG = {index: tag for tag, index in TAG_TO_INDEX.items()}
```

Class to convert the words and labels in the training data from their original string form into numerical form.

The Encode class has two methods-
Encode_Process - Inputs sentence and its corresponding label (in the form of a list of named entity tags) and encodes them into numerical form. To handle capitalization problem, appended 1-dim vector (capitalization) to word embeddings. It first creates a list capitalization that encodes the capitalization of each word in the sentence as either 0 (for lowercase) or 1 (for uppercase or mixed case). Then it maps each word in the sentence to its corresponding index in the **WORD_TO_INDEX_GLOVE** dictionary, which was created by the vocab_create function. If a word is not in the dictionary (i.e., not in the GloVe embeddings), it is mapped to index 1, which corresponds to the < unk > token. Finally, each named entity tag in the label is mapped to its corresponding index in the **TAG_TO_INDEX** dictionary.

Encode_Vocab - Inputs list of (sentence, label) pairs and applies the Encode_Process method to each pair. It returns a list of tuples, where each tuple contains the encoded sentence, capitalization, and label.

```
class Encode2:
    def Encode_Process(self, sentence, label):
        """
        Encode the sentence and labels into numerical form
        """
        capitalization = []
        for word in sentence:
            if word.lower() == word:
                capitalization.append([0])  # lowercase
            else:
                capitalization.append([1])  # uppercase or mixed case
```

```
        sentence = [WORD_TO_INDEX_GLOVE.get(word.lower(), 1) for word in sentence]
        label = [TAG_TO_INDEX[lab] for lab in label]
        return sentence, capitalization, label

    def Encode_Vocab(self, data):
        data = [self.Encode_Process(sentence, label) for sentence, label in data]
        return data
```

```
train_data = read_data(TRAIN_PATH)
vocab_create()

encode = Encode2()
train_data = encode.Encode_Vocab(train_data)
```

NERDataset - PyTorch Dataset class for the Named Entity Recognition (NER) task. It takes in a list of preprocessed data, where each item is a tuple of a list of encoded sentence and a list of encoded labels.
The **getitem** method returns the encoded sentence, capitalization, and label at the given index as PyTorch tensors.
The **len** method returns the length of the dataset, i.e., the number of examples in the data.

```
class NERDataset2(data.Dataset):
    def __init__(self, data):
        self.data = data

    def __getitem__(self, index):
        sentence, capitalization, label = self.data[index]
        capitalization = torch.LongTensor(capitalization)
        sentence = torch.LongTensor(sentence)
        label = torch.LongTensor(label)
        return sentence, capitalization, label
        # return sentence, label

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

The **collate_fn** function pads the sequences to the same length, and the **DataLoader** batches the data and shuffles it during training.

```
def collate_fn2(batch):
    sentence, capitalization, label = zip(*batch)
    sentence = pad_sequence(sentence, batch_first=True, padding_value=0)
    capitalization = pad_sequence(capitalization, batch_first=True, padding_value=0)
    label = pad_sequence(label, batch_first=True, padding_value=-1)
    return sentence, capitalization, label

batch_size = 64
train_dataset = NERDataset(train_data)
train_loader = DataLoader(train_dataset,
                          batch_size=batch_size,
                          shuffle=False,
                          collate_fn=collate_fn2)
```

Hyper-parameters

input size = len(WORD_TO_INDEX_GLOVE)
output size = len(TAG_TO_INDEX)
embedding dimension = 101
lstm hidden dimension = 256
linear output dimension = 128
number of LSTM layers = 1
dropout = 0.33
learning rate = 0.5
number of epochs = 10

The **CrossEntropyLoss** function is used as the loss function, along with an optimizer **SGD** with momentum=0.9.
weight - The second argument to CrossEntropyLoss is the weight for each class. This is used to give more importance to certain classes while computing the loss. In this case, class 0 ['O'] is given a weight of 0.7 and all other classes are given a weight of 1.
ignore_index is used to ignore the padded values (-1) while computing the loss.

```
input_size = len(WORD_TO_INDEX_GLOVE)
output_size = len(TAG_TO_INDEX)
embedding_dim = 100
lstm_hidden_dim = 256
linear_output_dim = 128
num_layers = 1
dropout = 0.33
lr = 0.5
num_epochs = 10
```

weights_matrix - size (input_size, embedding_dim), where input_size is the size of the vocabulary (the number of unique words in the training data plus 2 for the padding and unknown tokens), and embedding_dim is the dimensionality of the word embeddings (100).

The for loop iterates through all the words in **WORD_TO_INDEX_GLOVE**. For each word, it converts the word to lowercase and checks whether it is in the **GLOVE_VECTOR** dictionary. If it is, it retrieves the pre-trained word embedding vector from GLOVE_VECTOR and assigns it to the corresponding row of weights_matrix. If the word is not in GLOVE_VECTOR, its row in weights_matrix remains all zeros.

```
weights_matrix = torch.zeros((input_size, embedding_dim))
for word, index in WORD_TO_INDEX_GLOVE.items():
        word = word.lower()
        if word in GLOVE_VECTOR:
            weights_matrix[index] = GLOVE_VECTOR[word]
```

The BLSTM2 class defines a neural network model architecture for Named Entity Recognition (NER).

input_size - size of the vocabulary, which is the number of unique words in the input dataset
output_size - number of unique tags in the output dataset
embedding_dim - size of the word embedding
lstm_hidden_dim - number of hidden units in the LSTM layer

linear_output_dim - number of output units in the linear layers
num_layers - number of LSTM layers in the model
dropout - dropout probability

The BLSTM2 model architecture is an improved version of the BLSTM1 model. This model includes a pre-trained embedding layer initialized with GloVe embeddings for each word in the vocabulary and bidirectional LSTM layer that processes the input sequence and produces output at each time step concate with Capitalization vector, followed by two Linear layers with ELU activation function, and a final Linear layer with softmax activation function. The input sequence is first packed and then processed by the LSTM layer to avoid processing padded tokens. The dropout layer is applied to the output of the LSTM layer to prevent

```
class BLSTM2(nn.Module):
    def __init__(self, input_size, output_size, embedding_dim, lstm_hidden_dim,
linear_output_dim, num_layers, dropout, glove_embeddings):
        super(BLSTM2, self).__init__()

        # self.embedding = nn.Embedding(input_size, embedding_dim)
        self.embedding = nn.Embedding.from_pretrained(glove_embeddings)

        self.blstm = nn.LSTM(embedding_dim+1, lstm_hidden_dim, num_layers=num_layers,
bidirectional=True) #dropout=dropout
        self.dropout = nn.Dropout(dropout)

        self.fc = nn.Linear(lstm_hidden_dim * 2, linear_output_dim)
        self.activation = nn.ELU()
        self.classifier = nn.Linear(linear_output_dim, output_size)

    def forward(self, sentence, capitalization):

        word_embedded = self.embedding(sentence)

        embedded = torch.cat((word_embedded, capitalization), dim=-1)

        length = torch.sum(sentence != 0, dim=1)
        packed_output = pack_padded_sequence(embedded, length, batch_first=True,
enforce_sorted=False)
        blstm_output, _ = self.blstm(packed_output)
        unpacked_output, _ = pad_packed_sequence(blstm_output, batch_first=True)

        blstm_output = self.dropout(unpacked_output)

        linear_output = self.fc(blstm_output)
        elu_output = self.activation(linear_output)

        output = self.classifier(elu_output)
        return output
```

```
model2 = BLSTM2(input_size, output_size, embedding_dim, lstm_hidden_dim,
linear_output_dim, num_layers, dropout, weights_matrix)
print(model2)
optimizer = optim.SGD(model2.parameters(), lr=lr, momentum=0.9)
class_weights = torch.tensor([0.7, 1, 1, 1, 1, 1, 1, 1, 1])
ignore_index = -1

loss_fn = nn.CrossEntropyLoss(weight=class_weights, ignore_index=ignore_index)
```

Model_Train() - trains the model for a given number of epochs using the provided data loader, loss function

```
def Model_Train():
    for epoch in range(num_epochs):
        running_loss = 0
        model2.train()
        for sentence, capitalization, label in train_loader:
            optimizer.zero_grad()
            # print(sentence.shape, capitalization.shape)
            output = model2(sentence, capitalization)
            output = output.permute(0, 2, 1)
            loss = loss_fn(output, label)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print('Epoch %d \t Training Loss: %.6f' % (epoch+1,
running_loss/len(train_loader)))
```

```
Model_Train()
```

Model_Evaluate() -

- The code is evaluating the trained model on the development dataset and writing the output to a file.
- It uses a dataloader to iterate over the sentences in the development dataset, feeds them to the trained model, and writes the predicted tags for each word in the sentence along with the original word and tag to an output file.
- The output is formatted in a way that can be compared to the original development dataset to evaluate the performance of the model.

```
dev_data = read_data(DEV_PATH)
encode = Encode2()
dev_dataset = encode.Encode_Vocab(dev_data)
dev_dataset = NERDataset2(dev_dataset)
dev_loader = DataLoader(dev_dataset, batch_size=1, shuffle=False)
```

```
def Model_Evaluate(output_file):
    model2.eval()

    with open(output_file, "w") as f:
        for i, (sentence, capitalization, label) in enumerate(dev_loader):
            output = model2(sentence, capitalization)
            predicted = output.argmax(dim = -1)
            index = 0
            for j in range(len(sentence[0])):
                index = index + 1
                word = dev_data[i][0][j]
                original_tag = INDEX_TO_TAG[label[0][j].item()]
                predicted_tag = INDEX_TO_TAG[predicted[0][j].item()]
                f.write("{} {} {}\n".format(index, word, predicted_tag))
            if (i != len(dev_loader)-1): f.write("\n")

DEV_OUT = 'dev2.out'
Model_Evaluate(DEV_OUT)
```

```
MODEL_PATH = 'blstm2.pt'

# Model Saving-
```

```
torch.save(model2, MODEL_PATH)

# Model Loading:
model2 = torch.load(MODEL_PATH)
model2.eval()
```

Model_Predict()

- The function first sets the model to evaluation mode using model2.eval(). It then iterates over each batch of data in the test_loader and passes the data through the model using model1(sentence, capitalization). The model's output is then converted into predicted labels using output.argmax(dim=-1).
- For each sentence in the batch, the function writes the predicted tag for each word to the specified output file. The output file is opened using the with statement, which ensures that the file is closed properly even if an error occurs during the writing process.
- Finally, the function is called with the argument test1.out, which specifies the output file path.

```
test_data = read_data_test(TEST_PATH)
encode = Encode2()
test_dataset = encode.Encode_Vocab(test_data)
test_dataset = NERDataset2(test_dataset)
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)
```

```
def Model_Predict(output_file):
    model2.eval()

    with open("test2.out", "w") as f:
        for i, (sentence, capitalization, label) in enumerate(test_loader):
            output = model2(sentence, capitalization)
            predicted = output.argmax(dim = -1)
            index = 0
            for j in range(len(sentence[0])):
                index = index + 1
                word = test_data[i][0][j]
                predicted_tag = INDEX_TO_TAG[predicted[0][j].item()]
                f.write("{} {} {}\n".format(index, word, predicted_tag))
            if (i != len(test_loader)-1): f.write("\n")

TEST_OUT = 'test2.out'
Model_Predict(TEST_OUT)
```