USC ID: 1041050678
Name: Shoumik Atul Gandre

CSCI 544: HOMEWORK 4

Contents

Results Table

Task	Precision	Recall	F1
Task 1	80.61%	81.99%	81.29
Task 2	90.86%	92.53%	91.69
Bonus Task	91.16%	92.54%	91.85

Answers to Questions

What is the precision, recall and F1 score on the dev data for Task1?

precision: 80.61%; recall: 81.99%; FB1: 81.29

Report from Perl script:

accuracy: 96.92%; precision: 80.61%; recall: 81.99%; FB1: 81.29

LOC: precision: 84.34%; recall: 92.65%; FB1: 88.30 2018

MISC: precision: 75.97%; recall: 82.65%; FB1: 79.17 1003

ORG: precision: 74.23%; recall: 55.41%; FB1: 63.45 1001

PER: precision: 82.34%; recall: 90.39%; FB1: 86.18 2022

What is the precision, recall and F1 score on the dev data for Task1?

precision: 90.86%; recall: 92.53%; FB1: 91.69

Report from Perl script:

processed 51578 tokens with 5942 phrases; found: 6051 phrases; correct: 5498.

accuracy: 98.59%; precision: 90.86%; recall: 92.53%; FB1: 91.69

LOC: precision: 94.13%; recall: 96.95%; FB1: 95.52 1892

MISC: precision: 82.27%; recall: 86.55%; FB1: 84.36 970

ORG: precision: 87.89%; recall: 83.89%; FB1: 85.85 1280

PER: precision: 93.98%; recall: 97.39%; FB1: 95.65 1909

What is the precision, recall and F1 score on the dev data for Task1?

precision: 91.16%; recall: 92.54%; FB1: 91.85

Report from Perl script:

processed 51578 tokens with 5942 phrases; found: 6032 phrases; correct: 5499.

accuracy: 98.65%; precision: 91.16%; recall: 92.54%; FB1: 91.85

LOC: precision: 93.37%; recall: 97.39%; FB1: 95.34 1916

MISC: precision: 84.29%; recall: 87.85%; FB1: 86.03 961

ORG: precision: 88.21%; recall: 82.55%; FB1: 85.29 1255

PER: precision: 94.37%; recall: 97.34%; FB1: 95.83 1900

How to deal with the capital word problem?

The way to deal with capital word problem is to create Boolean mask for each category of capital words.

There are four classes of capital words:

1. First word uppercase
2. Complete lowercase word
3. Complete uppercase word
4. Mix case word

Therefore, we create 4 Boolean masks and concatenate them with the output of the embeddings layer.

Task 1: Simple Bidirectional LSTM model

Model Definition:

```
BLSTM(
  (embedding): Embedding(10997, 100, padding_idx=0)
  (bilstm): LSTM(104, 256, batch_first=True, bidirectional=True)
  (lstm_dropout): Dropout(p=0.33, inplace=False)
  (linear_elu): Sequential(
    (0): Linear(in_features=512, out_features=128, bias=True)
    (1): ELU(alpha=1.0)
  )
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)
```

Hyperparameters:

For first 100 epochs:

BATCH_SIZE = 256

NUM_EPOCHS = 100

LEARNING_RATE = 0.7

TRAIN_EMBEDDINGS = True

SHUFFLE_DATASET = True

For Next 400 epochs:

BATCH_SIZE = 256

LEARNING_RATE = 0.01

TRAIN_EMBEDDINGS = True

SHUFFLE_DATASET = True

Solution Details:

First, we handle unknown words. All words with a threshold below 2 are converted to unknown tags.

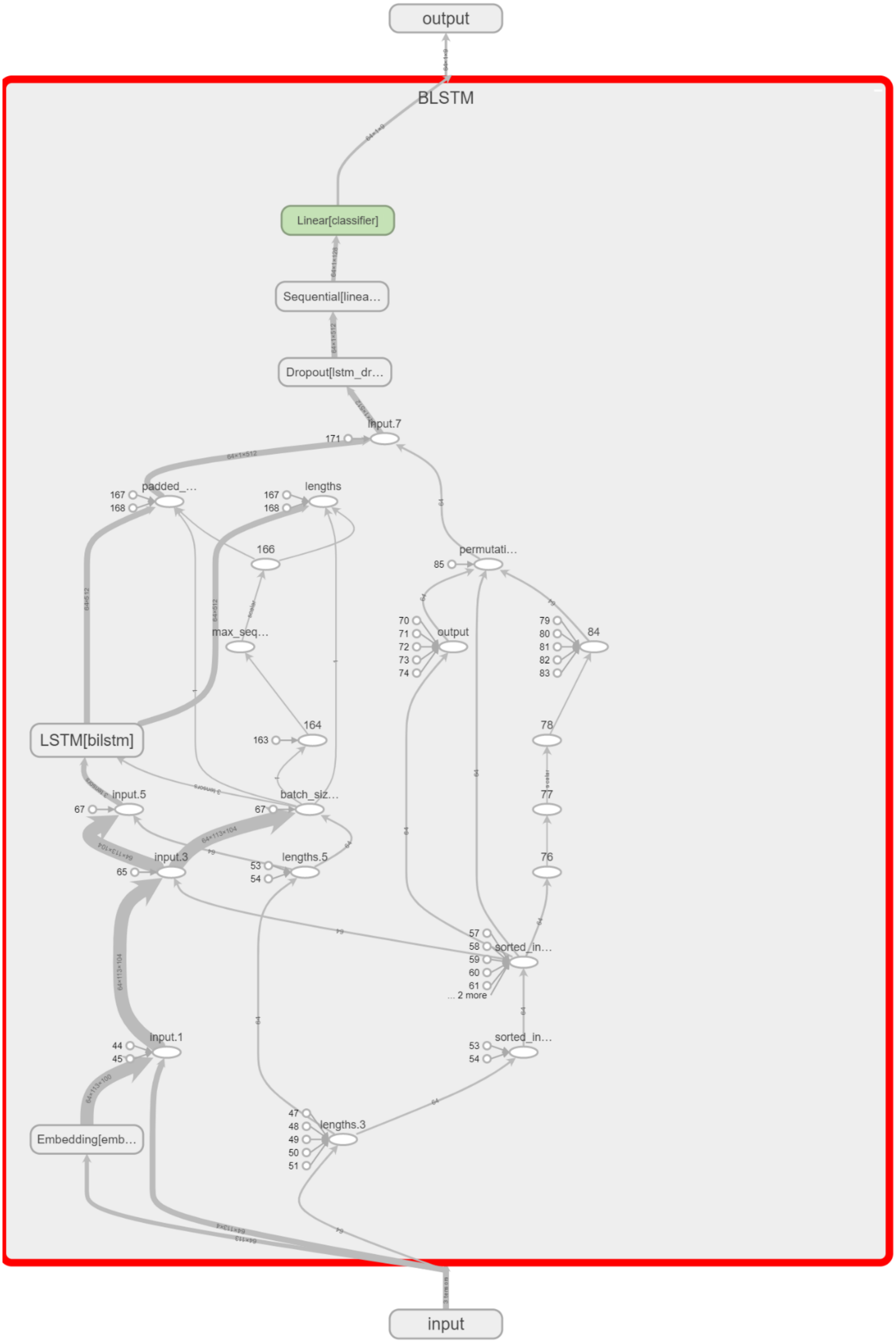
We handle unknown words by the use of pseudowords. (I have mentioned this process in my HW2)

create a vocabulary from this new dataset.

create a capital mask.

convert all words to lowercase.

Pass pre-processed dataset with the capital mask in the model.


output
BLSTM
Linear[classifier]
Sequential[linea…
Dropout[lstm_dr…
input.7
171
padded_…
lengths
167
168
167
168
166
permutati…
85
max_seq…
70
71
72
73
74
output
79
80
81
82
83
84
LSTM[bilstm]
164
163
78
input.5
67
batch_siz…
67
77
input.3
65
lengths.5
53
54
76
57
58
59
60
61
… 2 more
sorted_in…
input.1
44
45
sorted_in…
53
54
Embedding[emb…
lengths.3
47
48
49
50
51
input

Task 2: Using GloVe word embeddings

Model Definition

```
Blstm(
  (embedding): Embedding(400002, 100, padding_idx=0)
  (bilstm): LSTM(104, 256, batch_first=True, bidirectional=True)
  (lstm_dropout): Dropout(p=0.33, inplace=False)
  (linear_elu): Sequential(
    (0): Linear(in_features=512, out_features=128, bias=True)
    (1): ELU(alpha=1.0)
  )
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)
```

Hyperparameters:

BATCH_SIZE = 256

NUM_EPOCHS = 1->300

LEARNING_RATE = 0.1

TRAIN_EMBEDDINGS = False

SHUFFLE = True

BATCH_SIZE = 64

NUM_EPOCHS = 300->320

LEARNING_RATE = 0.0105

TRAIN_EMBEDDINGS = False

SHUFFLE = True

BATCH_SIZE = 256

NUM_EPOCHS = 320->700

LEARNING_RATE = 0.0105

TRAIN_EMBEDDINGS = True

SHUFFLE = True

Solution Details:

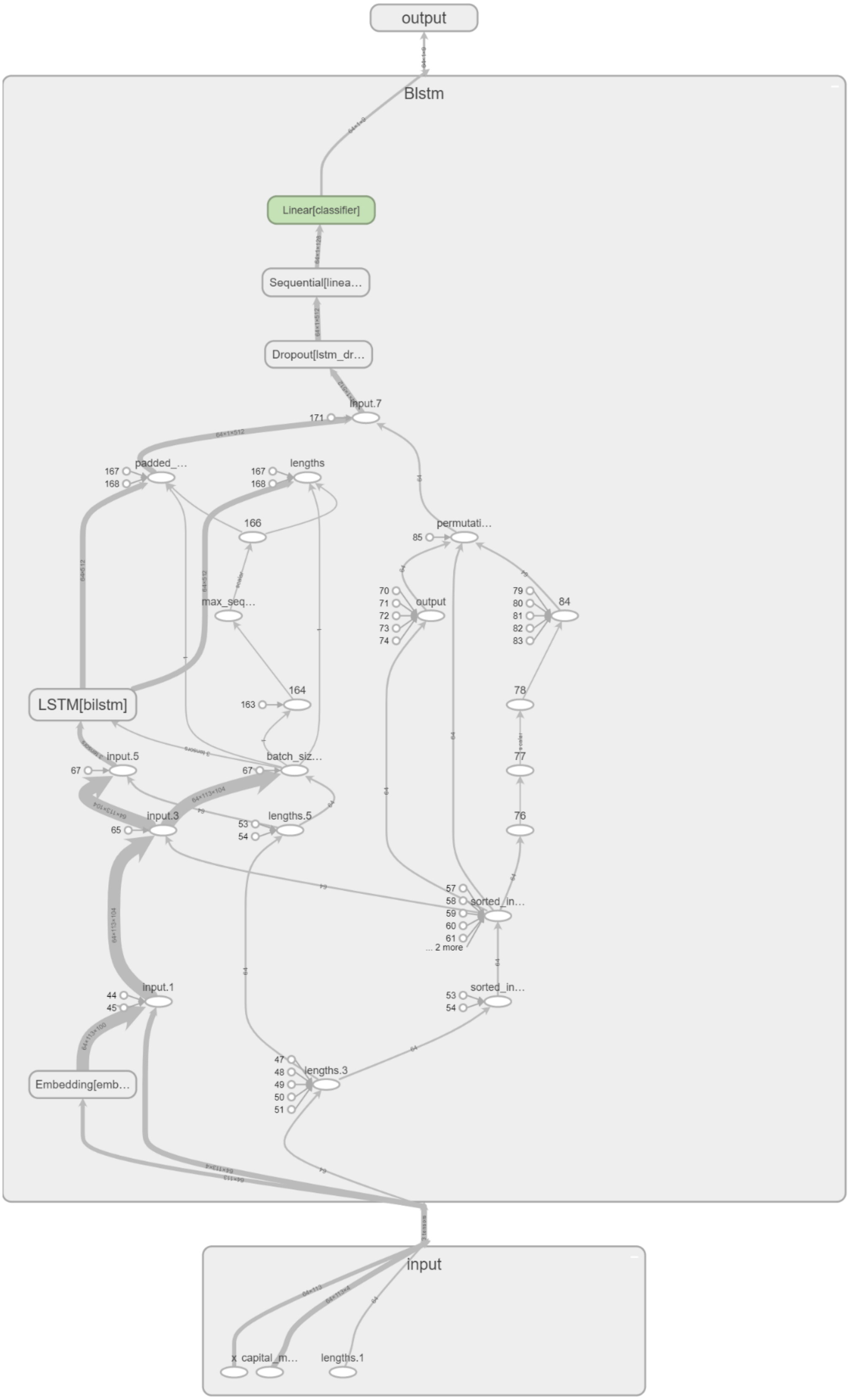
create a vocabulary from the dataset.

create a capital mask.

convert all words to lowercase.

Load glove embeddings into model.

Pass pre-processed dataset with the capital mask in the model.


output
Blstm
Linear[classifier]
Sequential[linea…
Dropout[lstm_dr…
input.7
padded_…
lengths
166
max_seq…
permutati…
output
84
LSTM[bilstm]
164
78
77
input.5
batch_siz…
input.3
lengths.5
76
sorted_in…
input.1
sorted_in…
Embedding[emb…
lengths.3
input
x
capital_m…
lengths.1

Bonus: LSTM-CNN model

Model Definition

```
BlstmCnn(
  (glove_encoder): Embedding(400002, 100, padding_idx=0)
  (token_encoder): TokenCharacterEncoder(
    (embedding): Embedding(85, 30)
    (conv): Conv1d(30, 30, kernel_size=(3,), stride=(1,), padding=(1,))
    (conv2): Conv1d(30, 30, kernel_size=(3,), stride=(1,), padding=(1,))
    (conv4): Conv1d(30, 30, kernel_size=(3,), stride=(1,), padding=(1,))
    (maxpool): AdaptiveMaxPool1d(output_size=1)
  )
  (bilstm): LSTM(134, 256, batch_first=True, bidirectional=True)
  (lstm_dropout): Dropout(p=0.33, inplace=False)
  (linear_elu): Sequential(
    (0): Linear(in_features=512, out_features=128, bias=True)
    (1): ELU(alpha=1.0)
  (classifier): Sequential(
    (0): Linear(in_features=128, out_features=9, bias=True)
  )
)
```

Hyperparameters:

Number of cnn layers = 4

For every cnn layer,

Number of input channels = 30

Number of output channels = 30

Kernel size = 3

Padding = 1

BATCH_SIZE = 9

NUM_EPOCHS = 80

LEARNING_RATE = 0.0105

shuffle=False

embeddings_train = True

BATCH_SIZE = 64

NUM_EPOCHS = 1-120

LEARNING_RATE = 0.0105

shuffle=False

embeddings_train = True

BATCH_SIZE = 64

NUM_EPOCHS = 120-145

LEARNING_RATE = 0.0105

shuffle=True

embeddings_train = True

BATCH_SIZE = 256

NUM_EPOCHS = 145-200

LEARNING_RATE = 0.1

shuffle=True

embeddings_train = True

Solution Details:

create a vocabulary from the dataset.

create a capital mask.

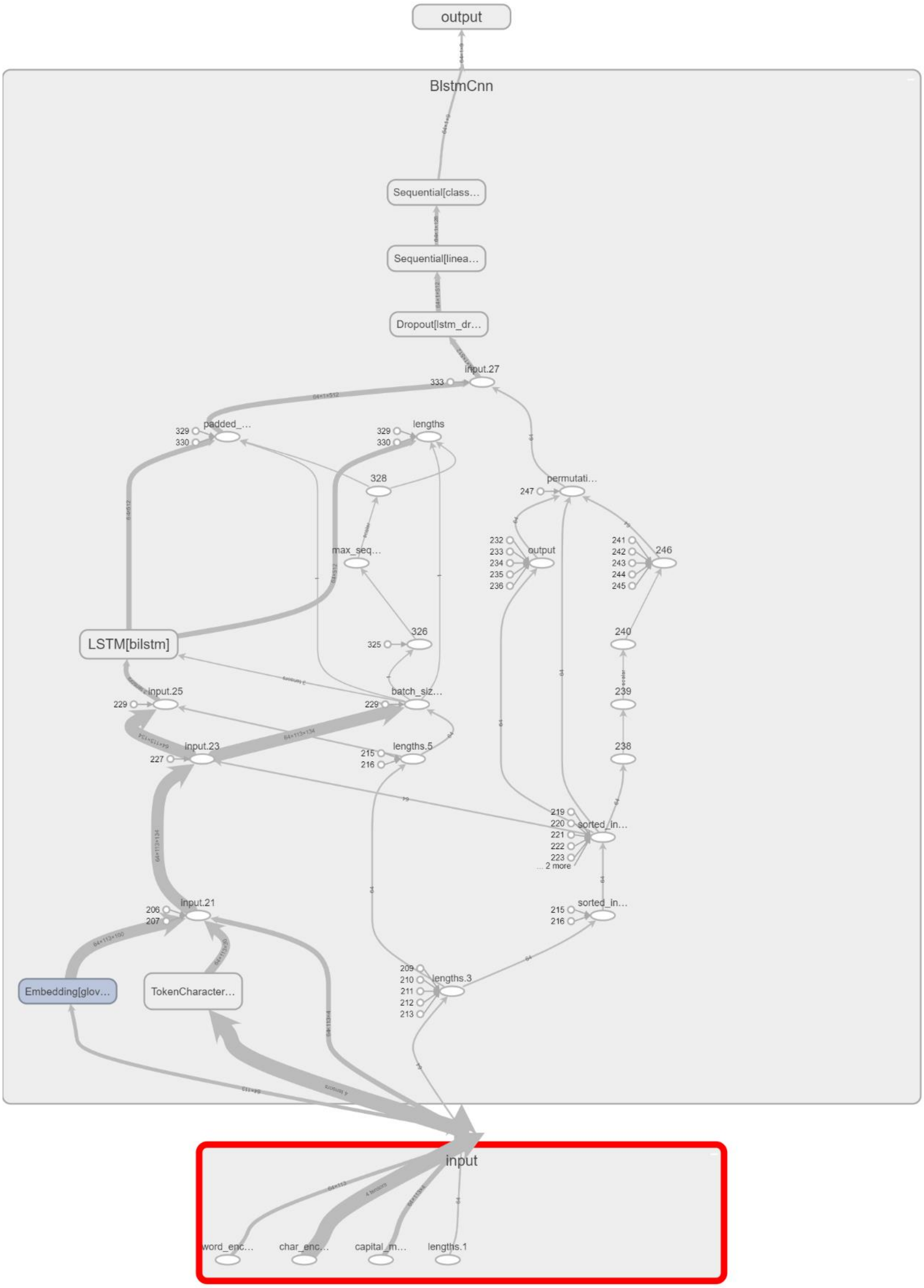
convert all words to lowercase.

Feature extract character level label encodings

Load glove embeddings into model.

Pass pre-processed dataset with the capital mask and character encodings in the model.

As instructed by Professor Xuezhe Ma, I am masking the padded character embeddings in the CNN layers



Code Documentation:

Vocabulary

```python
class Vocabulary(UserDict[str, int]):
    """A dictionary of keys of type str and values of type int"""

    def get_inverse(self) -> dict[int, str]:
        return {value: key for key, value in self.items()}

    @staticmethod
    def from_sequences(sequences: list[list[str]], start_index=1) -> Vocabulary:
        vocab = Vocabulary()
        index = start_index

        for sent in sequences:
            for item in sent:
                if item not in vocab:
                    vocab[item] = index
                    index += 1

        return vocab

    def to_dict(self) -> dict[str, int]:
        return self.data

    @staticmethod
    def from_dict(data: dict[str, int]) -> Vocabulary:
        return Vocabulary(data)

    def to_file(self, path: str | PathLike) -> None:
        with open(path, 'w') as f:
            json.dump(self.to_dict(), f)

    @staticmethod
    def from_file(path: str | PathLike) -> Vocabulary:
        if Path(path).exists():
            with open(path) as f:
                word_vocab = Vocabulary.from_dict(json.load(f))
            return word_vocab
        else:
```

```
            raise FileNotFoundError()
```

This class is a convenience class for creating and storing vocabularies for the word level

```
class CharacterVocabulary(UserDict[str, int]):
    """A dictionary of keys of type str and values of type int"""

    def get_inverse(self) -> dict[int, str]:
        return {value: key for key, value in self.items()}

    @staticmethod
    def from_sequences(sequences: list[list[str]], start_index=1) -> CharacterVocabulary:
        vocab = CharacterVocabulary()
        index = start_index

        for sequence in sequences:
            for token in sequence:
                for character in token:
                    if character not in vocab.keys():
                        vocab[character] = index
                        index +=1

        return vocab

    def to_dict(self) -> dict[str, int]:
        return self.data

    @staticmethod
    def from_dict(data: dict[str, int]) -> CharacterVocabulary:
        return CharacterVocabulary(data)

    def to_file(self, path: str | PathLike) -> None:
        with open(path, 'w') as f:
            json.dump(self.to_dict(), f)

    @staticmethod
    def from_file(path: str | PathLike) -> CharacterVocabulary:
        if Path(path).exists():
            with open(path) as f:
                word_vocab = CharacterVocabulary.from_dict(json.load(f))
            return word_vocab
        else:
            raise FileNotFoundError()
```

This class is a convenience class for creating and storing vocabularies for the character level.

Label Encoding

```
@dataclass
class LabelEncoder:
    vocab: Vocabulary

    def __post_init__(self):
        self.inverse_vocab: dict[int, str] = self.vocab.get_inverse()

    def transform(self, sequences: list[list[str]], default_value=1, key: Callable[[str], str]=lambda x: x) -> list[list[int]]:
        return [
            [self.vocab.get(key(word), default_value) for word in sequence]
            for sequence in sequences
        ]

    def inverse_transform(self, sequences: list[list[int]], default_value='O') -> list[list[str]]:
        return [
            [self.inverse_vocab.get(num, default_value) for num in sequence]
            for sequence in sequences
        ]
```

This class is used to encode words into word encodings.

```
@dataclass
class CharacterLabelEncoder:
    vocab: CharacterVocabulary

    def __post_init__(self):
        self.inverse_vocab: dict[int, str] = self.vocab.get_inverse()

    def transform(self, sequences: list[list[str]], default_value=1, key: Callable[[str], str]=lambda x: x) -> list[list[list[int]]]:
        return [
            [
                [self.vocab.get(key(ch), default_value) for ch in word]
                for word in sequence
            ]
            for sequence in sequences
```

```
        ]
```

This class is used to encode characters into character encodings.

Dataset

```
class Conll03Dataset(Dataset):

    def __init__(self, path: str | PathLike,
                 tag_vocab_path: str | PathLike,
                 feature_extractors: dict[str, Callable[[list[list[str]]], list[torch.Tensor]]],
                 use_targets=True) -> None:

        # Flags
        self.use_targets = use_targets

        if use_targets:
            word_sequences, tag_sequences = read_wordsequences_tagsequences(path, ' ')
            ### Tag Preprocessing:
            # Label Encode the Tag Sequences
            tag_encoder = LabelEncoder(Vocabulary.from_file(tag_vocab_path))
            tag_sequences_le = tag_encoder.transform(tag_sequences)
            # Create List of Label Encoded Tag Sequence Tensors
            self.tag_encodings: list[torch.Tensor] = [torch.tensor(sequence) for sequence in tag_sequences_le]
        else:
            word_sequences = read_wordsequences(path, ' ')

        # Feature Extraction
        self.feature_names = set()
        for feature_name, feature_extractor in feature_extractors.items():
            self.feature_names.add(feature_name)
            setattr(self, feature_name, feature_extractor(word_sequences))

    def __getitem__(self, index: int) -> dict[str, torch.Tensor]:
        item = {feature_name: getattr(self, feature_name)[index] for feature_name in self.feature_names}
        if self.use_targets:
            item |= {'tag_encodings': self.tag_encodings[index]}

        return item

```

```
    def __len__(self) -> int:
        feature_name = next(iter(self.feature_names))
        return len(getattr(self, feature_name))
```

This is my flexible dataset class for loading and using sequence data.

You can add multiple features, using the feature extractors dictionary argument in the init function.

For Example,

Capital mask features:

```
    train_dataset = Conll03Dataset(train_path,
                                   TAG_VOCAB_PATH,
                                   feature_extractors={
                                       'word_encodings': partial(unk_handler_preprocesser, word_vocab_path=WORD_VOCAB_PATH),
                                       'capital_mask': case_feature_extractor},
                                   use_targets=True)
```

Character Level features and capital mask features:

```
    train_dataset = Conll03Dataset(train_path,
                              TAG_VOCAB_PATH,
                              feature_extractors={
                                  'word_encodings': partial(word_preprocessor, word_vocab_path=WORD_VOCAB_PATH),
                                  'capital_mask': case_feature_extractor,
                                  'char_encodings': partial(
                                      character_feature_extractor,
                                      char_vocab_path=CHAR_VOCAB_PATH,
                                      word_vocab_path=WORD_VOCAB_PATH)},
                              use_targets=True)
```

Model

Blstm1

```
class BLSTM(nn.Module):

    def __init__(self, vocab_size: int,
                 embedding_size: int=100,
                 hidden_size: int=256,
                 linear_size: int=128,
                 out_size: int=9,
                 padding_idx:int=0) -> None:

```

```
        super(BLSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_size, padding_idx=padding_idx)

        # LSTM
        self.bilstm = nn.LSTM(input_size=embedding_size+4,
                              hidden_size=hidden_size,
                              num_layers=1,
                              batch_first=True,
                              bidirectional=True)
        self.lstm_dropout = nn.Dropout(0.33)

        # Linear-ELU
        self.linear_elu = nn.Sequential(nn.Linear(2*hidden_size, linear_size),
                                        nn.ELU())

        # Classifier
        self.classifier = nn.Linear(linear_size, out_size)


    def forward(self, x: torch.Tensor, capital_mask: torch.Tensor, lengths: torch.Tensor) -> torch.Tensor:

        embeddings = self.embedding(x)

        output = torch.cat((embeddings, capital_mask), dim=-1)

        # LSTM
        packed = pack_padded_sequence(output, lengths, batch_first=True, enforce_sorted=False)
        output, _ = self.bilstm(packed)
        output, _ = pad_packed_sequence(output, batch_first=True)
        output = self.lstm_dropout(output)
        # Linear - ELU
        output = self.linear_elu(output)

        # Classifier
        output = self.classifier(output)
        return output

    def predict(model: nn.Module,
         dataloader: DataLoader,
         device: str="cpu") -> list[list[int]]:
        """
        Inputs:
            model is the Neural Network Architecture
            dataloder is the testing dataset
```

```
            device is the device on which the training needs to be run on
        """
        # Testing
        model.eval()
        y_pred = []

        for batch in tqdm(dataloader):
            with torch.no_grad():
                word_encodings = batch['word_encodings'].to(device)
                capital_mask = batch['capital_mask'].to(device)
                lengths = batch['lengths']
                probabilities = model(word_encodings, capital_mask, lengths)
                predictions = probabilities.argmax(-1)
                predictions = predictions.tolist()
                predictions = [prediction[:length] for prediction, length in zip(predictions, lengths)]

                y_pred.extend(predictions)

        return y_pred
```

Blstm2

```
class Blstm(nn.Module):

    def __init__(self,
                 embeddings: np.ndarray,
                 embedding_size: int=100,
                 hidden_size: int=256,
                 linear_size: int=128,
                 out_size: int=9,
                 padding_idx:int=0) -> None:

        super(Blstm, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(torch.from_numpy(embeddings).float(), padding_idx=padding_idx)

        # LSTM
        self.bilstm = nn.LSTM(input_size=embedding_size+4,
                              hidden_size=hidden_size,
                              num_layers=1,
                              batch_first=True,
                              bidirectional=True)

        self.lstm_dropout = nn.Dropout(0.33)
```

```

        # Linear-ELU
        self.linear_elu = nn.Sequential(nn.Linear(2*hidden_size, linear_size),
                                        nn.ELU())

        # Classifier
        self.classifier = nn.Linear(linear_size, out_size)


    def forward(self, x: torch.Tensor, capital_mask: torch.Tensor, lengths: torch.Tensor) -> torch.Tensor:

        embeddings = self.embedding(x)

        output = torch.cat((embeddings, capital_mask), dim=-1)

        # LSTM
        output = pack_padded_sequence(output, lengths, batch_first=True, enforce_sorted=False)
        output, _ = self.bilstm(output)
        output, _ = pad_packed_sequence(output, batch_first=True)
        output = self.lstm_dropout(output)
        # Linear - ELU
        output = self.linear_elu(output)

        # Classifier
        output = self.classifier(output)
        return output

    def predict(model: nn.Module,
         dataloader: DataLoader,
         device: str="cpu") -> list[list[int]]:
        """
        Inputs:
            model is the Neural Network Architecture
            dataloder is the testing dataset
            device is the device on which the training needs to be run on
        """
        # Testing
        model.eval()
        y_pred = []

        for batch in tqdm(dataloader):
            with torch.no_grad():

                word_encodings = batch['word_encodings'].to(device)
```

```
                capital_mask = batch['capital_mask'].to(device)
                lengths = batch['lengths']

                probabilities: torch.Tensor = model(word_encodings, capital_mask, lengths)
                predictions = probabilities.argmax(-1)
                predictions = predictions.tolist()
                predictions = [prediction[:length] for prediction, length in zip(predictions, lengths)]

                y_pred.extend(predictions)

        return y_pred
```

Blstm3

```
class TokenCharacterEncoder(nn.Module):
    """
    Encodes tokens into 30 dimensions characterwise
    It requires label encoded characters to function
    """

    def __init__(self, num_embeddings: int, character_pad_value: int=0):
        super(TokenCharacterEncoder, self).__init__()
        self.embedding = nn.Embedding(num_embeddings, embedding_dim=30)
        self.conv = nn.Conv1d(30, 30, kernel_size=3, padding=1)

        self.conv2 = nn.Conv1d(30, 30, kernel_size=3, padding=1)
        self.conv3 = nn.Conv1d(30, 30, kernel_size=3, padding=1)
        self.conv4 = nn.Conv1d(30, 30, kernel_size=3, padding=1)

        self.maxpool = nn.AdaptiveMaxPool1d(output_size=1)
        self.character_pad_value = character_pad_value

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        char_encodings.shape - (batch size, sentence length, word length)
        """
        pad_mask = (x == self.character_pad_value)
        pad_mask = pad_mask.repeat(30, 1, 1, 1).permute(1, 2, 3, 0)
        pad_mask = pad_mask.flatten(0, 1).permute(0, 2, 1).to(x.device)

        N, S = x.shape[:2]
        x = x.flatten(0, 1)
        # out = self.single_sentence_forward(char_encodings)
        x = self.embedding(x)
        # x.shape - (sentence length, word length, num_embeddings)
```

```
        x = x.permute(0, 2, 1)
        # x.shape - (sentence length, num_embeddings, word length)
        x = F.relu(self.conv(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))

        x = x.masked_fill(pad_mask, float("-inf"))
        # x.shape - (sentence length, conv out dim=30, word length)
        x = self.maxpool(x).squeeze(-1)
        # x.shape - (sentence length, conv out dim=30)
        x = x.unflatten(0, (N, S))
        # out.shape: (batch size, sentence length, 30)
        return x


class BlstmCnn(nn.Module):

    def __init__(self,
                 glove_embeddings: np.ndarray,
                 num_char_embeddings: int=100,
                 token_padding_idx:int=0) -> None:

        super(BlstmCnn, self).__init__()

        # Glove Embeddings
        self.glove_encoder = nn.Embedding.from_pretrained(
            torch.from_numpy(glove_embeddings).float(),
            padding_idx=token_padding_idx
        )

        # Character Level Embeddings
        self.token_encoder = TokenCharacterEncoder(num_embeddings=num_char_embeddings)

        # LSTM
        self.bilstm = nn.LSTM(input_size=100+30+1+3, hidden_size=256,
                              num_layers=1, batch_first=True, bidirectional=True)

        self.lstm_dropout = nn.Dropout(0.33)

        # Linear-ELU
        self.linear_elu = nn.Sequential(nn.Linear(2*256, 128), nn.ELU())

        # Classifier
        self.classifier = nn.Sequential(nn.Linear(128, 9))

```

```

    def forward(self,
                word_encodings: torch.Tensor,
                char_encodings: torch.Tensor,
                capital_mask: torch.Tensor,
                lengths: torch.Tensor) -> torch.Tensor:
        glove_embeddings = self.glove_encoder(word_encodings)

        token_embeddings = self.token_encoder(char_encodings)

        output = torch.cat((glove_embeddings, token_embeddings, capital_mask), dim=-1)

        # LSTM
        output = pack_padded_sequence(output, lengths, batch_first=True, enforce_sorted=False)
        output, _ = self.bilstm(output)
        output, _ = pad_packed_sequence(output, batch_first=True)
        output = self.lstm_dropout(output)
        # Linear - ELU
        output = self.linear_elu(output)

        # Classifier
        output = self.classifier(output)
        return output


def predict(model: nn.Module,
            dataloader: DataLoader,
            device: str="cpu") -> list[list[int]]:
    """
    Inputs:
        model is the Neural Network Architecture
        dataloder is the testing dataset
        device is the device on which the training needs to be run on
    """
    # Testing
    model.eval()
    y_pred = []

    for batch in tqdm(dataloader):
        with torch.no_grad():

            word_encodings = batch['word_encodings'].to(device)
            char_encodings = batch['char_encodings'].to(device)
```

```
            capital_mask = batch['capital_mask'].to(device)
            lengths = batch['lengths']

            probabilities: torch.Tensor = model(word_encodings, char_encodings, capital_mask, lengths)
            predictions = probabilities.argmax(-1)
            predictions = predictions.tolist()
            predictions = [prediction[:length] for prediction, length in zip(predictions, lengths)]

            y_pred.extend(predictions)

    return y_pred
```