University of Paris-Saclay - UFR Sciences

# Lab 3 report - Part-of-speech tagging

Diyun LU - Marwan MASHRA - Yihan ZHONG

December 23, 2022

## 1 Introduction

### 1.1 Part-Of-Speech (POS) tagging

The part-of-speech tagging task is to assign each word in a given sentence a part-of-speech tag from a given tag set. It is the process of classifying and labeling the words in the sentence, and it is actually a multi-classification task. We give each word the corresponding POS through POS classification based on the components of the word in the syntactic structure or language form.

That is to determine whether each word in the sentence is a noun, verb, adjective, or other POS, in our experiment, we used 18 different tags in total.

POS is a basic task in NLP and has applications in many fields of speech recognition, information retrieval, etc. A common approach to POS tagging is to use a sequence labeling model, such as an RNN or a transformer. Sequence labeling models take a sequence of words as input, and the output will be a sequence of POS tags, where each POS tag is a prediction for the corresponding word in the input sequence.

For example, a simple RNN model for POS tagging might consist of an embedding layer to map words to dense vectors, a recurrent layer (such as LSTM) to process sequences of word vectors, and a fully connected layer to predict POS tags. The model will be trained to minimize the cross-entropy loss between the predicted POS tag and the true POS tag for a given input sentence. In our experiment, we used a multi-stack BiLSTM.

### 1.2 Multi-stack BiLSTM

First of all, BiLSTM network is a type of LSTM that processes input sequences in both forward and backward directions, allowing the model to capture contextual dependencies in the input data from both past and future contexts. As the name suggests, a multi-stack BiLSTM is a type of BiLSTM that consists of multiple layers of BiLSTMs. In a multi-stack BiLSTM, the input sequence is processed by multiple layers of BiLSTMs, where each layer is stacked on top of the previous layer. This allows the model to learn hierarchical representations of the input data, capturing increasingly complex patterns as the input data is processed through the layers.

### 1.3 Fast-text embeddings

FastText is a library for efficiently learning word representations and sentence classification. It was developed by Facebook AI Research. It includes many techniques for learning word embeddings and also provides many pre-trained word embeddings. Since we would like to evaluate in cross-lingual fields, so in our experiments, we will be using both English and French Embeddings.

## 2 Implementation

### 2.1 Data preprocessing

#### 2.1.1 Data loader

The ID column contains 3 types of values in the input files, so we just used *.isdigit*() to check if it is the correct lines that we need to upload, then we converted all words to lowercase as we only have embeddings

for lowercased words.

```
def read_conll(in_file, lowercase=True, max_example=None):
    list_txt = []
    list_pos = []
    with open(in_file) as f:
        word, pos = [], []
        for line in f.readlines():
            sp = line.strip().split('\t')
            if len(sp) == 10:
                if sp[0].isdigit():
                    word.append(sp[1].lower() if lowercase else sp[1])
                    pos.append(sp[3])
            elif len(word) > 0:
                list_txt.append(word)
                list_pos.append(pos)
                word, pos = [], []
                if (max_example is not None) and (len(list_txt) == max_example):
                    break
        if len(word) > 0:
            list_txt.append(word)
            list_pos.append(pos)
    return list_txt, list_pos
```

Then we read the word embeddings from the given documents, and also build a word dictionary that maps words and integers at the same time. Note that we add a $< unk >$ token with a zero vector embedding for unseen words.

```
def read_emb(file_path):
    emb_dim = 300
    with open(file_path, 'r') as f:
        lines = f.readlines()
        emb = torch.empty(size=(len(lines)+1, emb_dim))
        word_to_id = {}
        id_to_word = {}
        i = len(lines)
        word_to_id['<unk>'] = i
        id_to_word[i] = '<unk>'
        emb[i].zero_()
        for i, line in enumerate(lines):
            splitted_lines = line.split()
            word = splitted_lines[0]
            word_to_id[word] = i
            id_to_word[i] = word
            emb[i] = torch.tensor(list(map(float, splitted_lines[-emb_dim:])), requires_grad=False)
        return emb, word_to_id, id_to_word
```

### 2.1.2 Data encoder

First, in order to reduce the number of unknown words, we preprocessed the words (in other parts of the code) to remove the "s" for plural, which helps to reduce the unknown by half. Then, we encode both the text and the labels into a list of sentences, each made of a list of integers (either the word ID, or the tag ID).

```
def encoder(txt, pos, word_to_id, pos_to_id):
    def word_encoder(word):
        if word not in word_to_id and word[-1]=='s' and word[:-1] in word_to_id:
            word = word[:-1]
        elif word not in word_to_id:
            word = '<unk>'
        return word_to_id[word]
    sentence_encoder = lambda sentence: list(map(word_encoder, sentence))
    pos_encoder = lambda sentence: list(map(lambda p: pos_to_id[p], sentence))
    txt_encoded = list(map(sentence_encoder, txt))
    pos_encoded = list(map(pos_encoder, pos))
    return txt_encoded, pos_encoded
```

## 2.2 Modelling

In this lab, we have implemented two modules: WordEmbedding class and POSTagging class.

### 2.2.1 Embedding Module

Embedding is utilized to retrieve word embeddings from fast-text files. It takes as input a list of sentences (each sentence is a list of IDs), and outputs a list of tensors, each of size $(len\_sentence, 300)$.

```
class Embedding(nn.Module):
    def __init__(self, emb_table: torch.Tensor):
        super(Embedding, self).__init__()
        self.emb_table = emb_table

    def forward(self, input):
        emb = list()
        for sentence in input:
            emb.append(self.emb_table[sentence].to(device))
        return emb
```

### 2.2.2 POSTagging Module

#### 2.2.2.1 Batching

Batching the input here is quite simple. We forward a list of sentences (with words as IDs) to our Embedding Module, and get back a list of sentences (with words as embeddings). Therefore, our input batch will be a list of sentences of length $(batch\_size)$ and each element is a tensor of size $(len\_sentence, 300)$. On the other hand, we batch the gold labels differently. We convert them from a list of tags for each sentence, to a one dimensional tensor $(total\_num\_tags)$, the total number of tags in the batch input. This allows us to pass the batched gold labels to the loss function directly.

```
gold = torch.cat([torch.tensor(y) for y in y_train[i:i+batch_size]]).to(device)
```

#### 2.2.2.2 Padding the data

Since sentences can vary in length, our input data is a list of tensors of different sizes. Therefore, in order to forward them through the LSTM, we need to pad them. We had this issue in Lab 2, which we solved by simply padding token with a zero vector embedding and then masking the output of the LSTM. This approach worked previously because we were using a Unidirectional LSTM, meaning that the output of the padding tokens at the end won't represent an input for the actual tokens. However, since we're

using a bidirectional LSTM here, having padding tokens at the end will affect the actual tokens in during backward direction pass. To solve this problem, PyTorch has a built-in function that allows us to create a *PackedSequence*, which contains the data as well as a list of batch sizes to allow the LSTM to know what tokens are real and what tokens are padding. To use this function, we first need to pad the input data using *pad_sequence*, then create the *PackedSequence* using *pack_padded_sequence* giving it the padded sequence and the list of batch sizes (lengths). We then forward the packed sequence through our stack of LSTMs and get the output as a *PackedSequence*. Finally, we use the function *pad_packed_sequence* to convert the packed sequence back to a tensor of size $(batch\_size, max\_lenght, hidden\_size)$. The output contains the output of the padding tokens as a 0 vector, which we can remove using a binary mask with the help of the lengths. Here is the code for the forward pass :

```
def forward(self, emb):
        # pad the sequence
        lengths = list(map(len, emb))
        padded_emb = pad_sequence(emb, batch_first=True)
        packed_emb = pack_padded_sequence(padded_emb, batch_first=True, lengths=lengths, enforce_sorted=False)
        # forward through LSTMs
        packed_output = packed_emb
        for lstm in self.lstms:
            packed_output, _ = lstm(packed_output)
        unpacked_output, unpacked_lengths = pad_packed_sequence(packed_output, batch_first=True)
        stacked_output = unpacked_output.view(-1, unpacked_output.size(2))
        # create the mask
        mask = torch.zeros(size=unpacked_output.shape[:2]).to(device)
        for i, length in enumerate(unpacked_lengths):
            mask[i, :length] = 1
        mask = mask.view(-1).to(bool)
        # forward through MLP
        output = stacked_output[mask]
        logits = self.mlp(output)
        log_probs = F.log_softmax(logits, dim=1)
        return log_probs
```

#### 2.2.2.3 Multi-stack BiLSTM

To create the multi-stack BiLSTM, we use the *ModuleList* module of PyTorch.

```
self.lstms = nn.ModuleList( nn.LSTM(input_size=self.hidden_size*2, hidden_size=self.hidden_size,
                            num_layers=num_layers, batch_first=True, bidirectional=True).to(device)
                            for _ in range(stack_size-1))
self.lstms.insert(0, nn.LSTM(input_size=self.embedding_dim, hidden_size=self.hidden_size,
                num_layers=num_layers, batch_first=True, bidirectional=True).to(device))
```

The input of each LSTM is of size $(hidden\_size \times 2)$ because its input is the output of the LSTM before, and the output of a BiLSTM is of size $\times 2$ the hidden size because we stack the output of both passes. Note that the first LSTM is an exception to that because its input is the embeddings.

## 2.3 Training

We train two Models. One has only one stack of BiLSTM, and the other has two stacks. We use the negative log-likelihood as a loss function, and track the loss of the training and dev set as well as the accuracy.

## 3 Results of our experiments

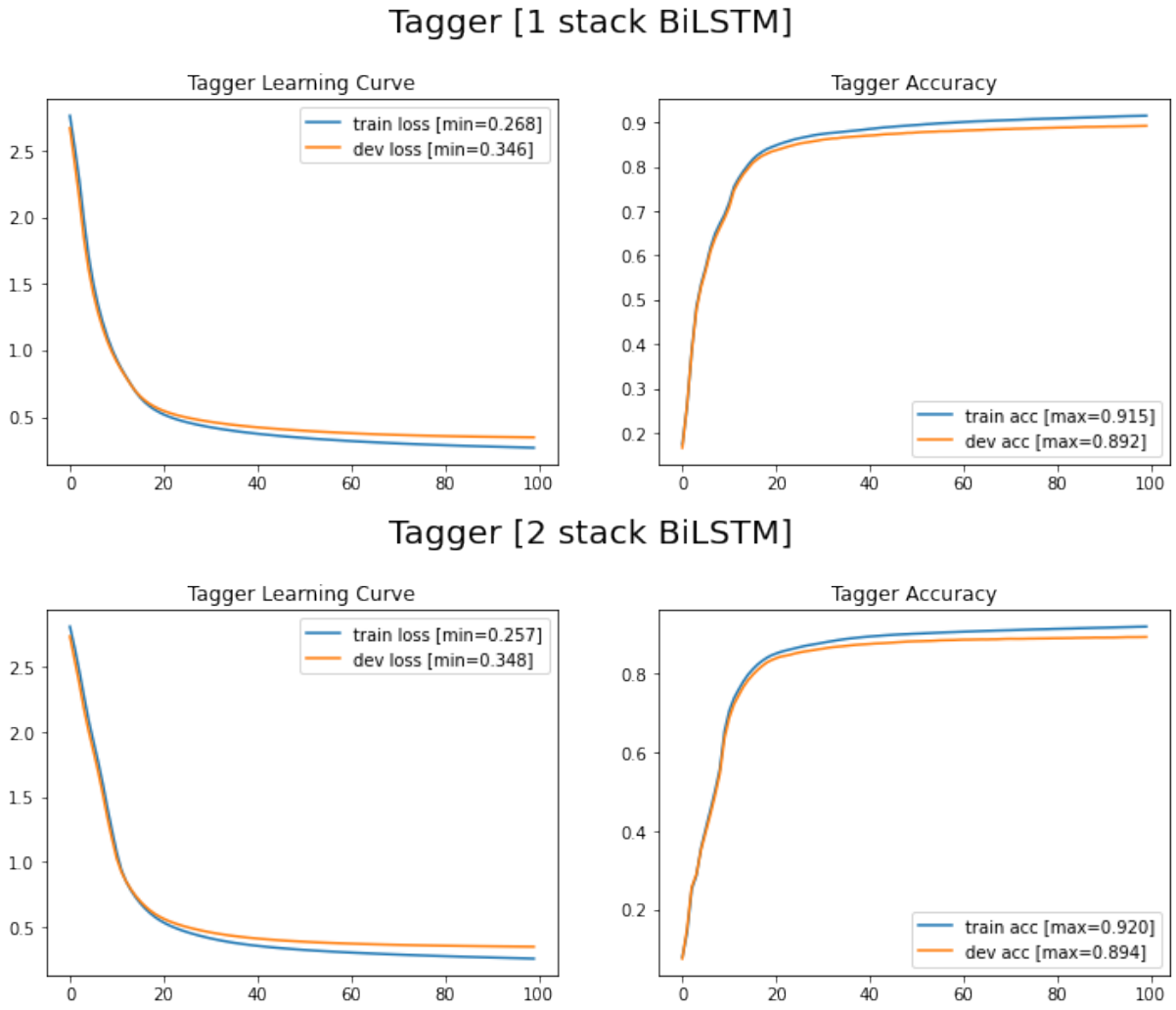


Figure 1: Learning curve and accuracy of both models

| | One Stack BiSTLM | Two Stacks BiLSTM |
|---|---|---|
| English Test Data | 0.898 | 0.901 |
| Out of Domain Data | 0.894 | 0.897 |
| French Data | 0.200 | 0.036 |

Table 1: Accuracy of both models on different test sets

We tested our model on the same language with different domains and different languages with the same domains. Interestingly, we notice here that having two stacks didn't make any significant difference to the performance of the model on English data. However, it did make the model generalize even less to french data. Overall, We can see that the performance is good with the same languages even if they are different domains. However, if we switch the language, even if the context is in the same domain, the performance is still really poor. We think that this may be because the BiLSTM model is built on certain assumptions, one of which is that there is a certain correlation between context words in the input sequence. If the correlation

is strong, then the BiLSTM model may perform well. But if we use a different language, the language syntactic structure of the input sequence is different from the language structure assumed by the BiLSTM model, then the performance of the BiLSTM model will be poor.

Here we can see a more detailed evaluation with the precision, recall, and F1-score of each tag on all test sets:

| | English Test Data | | | Out of Domain Data | | | French Data | | |
|---|---|---|---|---|---|---|---|---|---|
| | precision | recall | F1-score | precision | recall | F1-score | precision | recall | F1-score |
| PRON | 0.87 | 0.76 | 0.81 | 0.82 | 0.80 | 0.81 | 0.20 | 0.00 | 0.00 |
| PUNCT | 0.93 | 0.96 | 0.94 | 0.97 | 0.96 | 0.97 | 0.39 | 0.55 | 0.46 |
| ADJ | 0.88 | 0.89 | 0.89 | 0.82 | 0.89 | 0.85 | 1.00 | 0.00 | 0.00 |
| NOUN | 0.88 | 0.90 | 0.89 | 0.91 | 0.89 | 0.90 | 0.21 | 0.09 | 0.13 |
| VERB | 0.86 | 0.89 | 0.87 | 0.84 | 0.84 | 0.84 | 0.40 | 0.18 | 0.25 |
| DET | 0.97 | 0.96 | 0.96 | 0.98 | 0.96 | 0.97 | 0.20 | 0.20 | 0.20 |
| ADP | 0.92 | 0.96 | 0.94 | 0.95 | 0.97 | 0.96 | 0.31 | 0.05 | 0.09 |
| AUX | 0.92 | 0.90 | 0.91 | 0.92 | 0.88 | 0.90 | 0.55 | 0.29 | 0.38 |
| PRON | 0.94 | 0.96 | 0.95 | 0.84 | 0.92 | 0.88 | 0.11 | 0.38 | 0.17 |
| PART | 0.92 | 0.90 | 0.91 | 0.88 | 0.89 | 0.88 | 0.00 | nan | nan |
| SCONJ | 0.90 | 0.71 | 0.79 | 0.64 | 0.54 | 0.59 | 0.04 | 0.47 | 0.07 |
| NUM | 0.63 | 0.78 | 0.70 | 0.72 | 0.81 | 0.76 | 0.14 | 0.42 | 0.21 |
| ADV | 0.87 | 0.86 | 0.87 | 0.84 | 0.79 | 0.82 | nan | 0.00 | nan |
| CCONJ | 1.00 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.27 | 0.78 | 0.40 |
| X | 0.67 | 0.10 | 0.18 | nan | 0.00 | nan | nan | 0.00 | nan |
| INTJ | 0.94 | 0.73 | 0.82 | 0.33 | 1.00 | 0.50 | 0.00 | 0.00 | nan |
| SYM | 0.73 | 0.62 | 0.67 | 0.84 | 0.64 | 0.73 | 0.32 | 0.18 | 0.23 |

Table 2: Evaluation of the One Stack BiLSTM on all test sets

| | English Test Data | | | Out of Domain Data | | | French Data | | |
|---|---|---|---|---|---|---|---|---|---|
| | precision | recall | F1-score | precision | recall | F1-score | precision | recall | F1-score |
| PRON | 0.88 | 0.77 | 0.82 | 0.82 | 0.80 | 0.81 | 0.00 | 0.00 | nan |
| PUNCT | 0.94 | 0.95 | 0.94 | 0.98 | 0.96 | 0.97 | 0.03 | 0.02 | 0.03 |
| ADJ | 0.89 | 0.90 | 0.90 | 0.83 | 0.89 | 0.86 | nan | 0.00 | nan |
| NOUN | 0.89 | 0.91 | 0.90 | 0.91 | 0.89 | 0.90 | 0.71 | 0.00 | 0.01 |
| VERB | 0.85 | 0.90 | 0.87 | 0.84 | 0.86 | 0.85 | 0.14 | 0.00 | 0.00 |
| DET | 0.97 | 0.95 | 0.96 | 0.97 | 0.96 | 0.97 | 0.00 | 0.00 | nan |
| ADP | 0.92 | 0.97 | 0.95 | 0.95 | 0.97 | 0.96 | 0.19 | 0.00 | 0.00 |
| AUX | 0.92 | 0.91 | 0.92 | 0.94 | 0.88 | 0.91 | 0.03 | 0.01 | 0.02 |
| PRON | 0.95 | 0.96 | 0.95 | 0.86 | 0.91 | 0.89 | 0.10 | 0.03 | 0.05 |
| PART | 0.93 | 0.90 | 0.92 | 0.89 | 0.91 | 0.90 | 0.00 | nan | nan |
| SCONJ | 0.90 | 0.75 | 0.82 | 0.62 | 0.59 | 0.60 | 0.00 | 0.00 | nan |
| NUM | 0.61 | 0.76 | 0.68 | 0.70 | 0.82 | 0.76 | 0.13 | 0.25 | 0.17 |
| ADV | 0.86 | 0.86 | 0.86 | 0.88 | 0.82 | 0.85 | nan | 0.00 | nan |
| CCONJ | 1.00 | 0.99 | 0.99 | 0.99 | 0.98 | 0.99 | 0.04 | 0.99 | 0.08 |
| X | 0.61 | 0.10 | 0.17 | 0.00 | 0.00 | nan | 0.00 | 0.00 | nan |
| INTJ | 0.94 | 0.68 | 0.79 | 0.50 | 1.00 | 0.67 | nan | 0.00 | nan |
| SYM | 0.66 | 0.64 | 0.65 | 0.87 | 0.64 | 0.74 | 0.00 | 0.03 | 0.00 |

Table 3: Evaluation of the Two Stack BiLSTM on all test sets