How Multilingual is Multilingual BERT?

He HUANG 22107447

November 23, 2023

1 Introduction

2 Fine-Tuning mBERT

```
1 import conllu
2 # Extracting PoS information from a conllu file
3 def load_conllu(filename):
4     for sentence in conllu.parse(open(filename, "rt", encoding="utf-8").read()):
5         tokenized_words = [token["form"] for token in sentence]
6         gold_tags = [token["upos"] for token in sentence]
7         yield tokenized_words, gold_tags
8 corpus = list(load_conllu("fr_sequoia-ud-test.conllu"))
```

2.1 The Universal Dependencies Project

1. Why do we need "consistent annotation" for our experiment ?
Our experiment aims to test a model across multiple languages. Consistent annotation ensures that similar linguistic structures or features are annotated in a uniform way between them. Therefore, it allows us to compare more effectively these languages that differ in grammar, syntax, and morphology.

2. What do we use the `yield` keyword rather than a simple `return` (line 7) ? Why do we have to call the `list` constructor (line 9) ?
The keyword `yield` creates a generator object, it will process one sentence at a time instead of loading the entire file, while `return` executes the function immediately and returns the results, thus it would be more time-consuming and take up a lot of memory.

The `list` will iterate over the entire generator and convert the generator into a list.

3. What is the distribution of labels in the test set of the `Sequoia` treebank ?

```
1 # count the number of occurrences of each POS tag
2 from collections import Counter
3 tag_counts = Counter(tag for _, tags in corpus for tag in tags)
4
5 # plot the distribution of POS tags
6 import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(10, 8))
plt.bar(tag_counts.keys(), tag_counts.values())
plt.xlabel("Label", fontsize=10)
plt.ylabel("Occurrences", fontsize=10)
plt.title("Distribution of POS tags", fontsize=14)
plt.show()
```

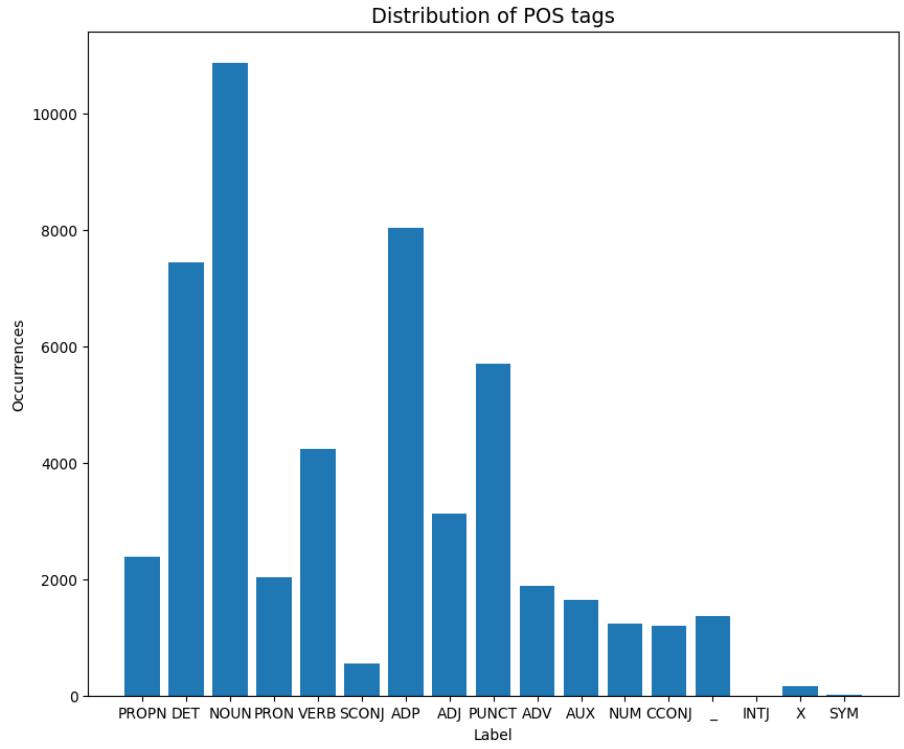

Figure 1: Distribution of POS tags in Sequoia train corpus

4. What are the multiword tokens in the test set of `Sequoia`? How are they annotated ?

According to the document UD French Sequoia, there 8 types multi-word tokens. Examples: des, du, au, aux, auxquels, auxquelles, duquel, desdites. There are annotated by label "_" or "ADP", "DET".

5. Why did the UD project made the decision of splitting multiword tokens into several (grammatical) words ? What is your opinion on this decision ?

The basic units of annotation are syntactic words (not phonological or orthographic words)[1]. A single word in French "au", can be a combination of the preposition "à" and the definite article "le", which contains morphological and syntactic characteristics. Therefore, splitting these words is necessary for exploring morphosyntactic dependencies.

6. According to UD tokenization guidelines, a token can contain spaces. Are there any token in the `Sequoia` corpora that contain spaces ? If so these spaces should be removed.

```
# find the tokens that contains spaces then move the spaces
```

[1] https://universaldependencies.org/u/overview/tokenization.html

```
num_space = 0
for sents,_ in corpus:
    for token in sents:
        if " " in token:
            num_space += 1
            token = token.replace(" ", "")
print(f"Number of spaces: {num_space} in Sequoia text corpus")

# Number of spaces: 13 in Sequoia text corpus
```

2.2 mBERT tokenization

7. Why does mBERT use a subword tokenization ?
Using subword tokenization allows us to have a smaller vocabulary size that can reduce the amount of memory. If we regard every possible word and wordform as a distinct token, the vocabulary size would become very large.

8. How is the sentence "Pouvez-vous donner les mêmes garanties au sein de l'Union Européene" tokenized according to the UD convention ? How is it tokenized by `mBERT` tokenizer ?
The result of mBERT tokenizer is:

```
!pip install transformers

sentence = "Pouvez-vous donner les mêmes garanties au sein de l'Union Européene"
# Tokenizing a single sentence with mBERT tokenizer.
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
# show the resulting tokenization
mbert_tokenized = tokenizer.tokenize(sentence)
print(mbert_tokenized)

Output:
['Po', '##uve', '##z', '-', 'vous', 'donner', 'les', 'mêmes', 'gara', '##nties',
    'au', 'sein', 'de', 'l', "'", 'Union', 'Euro', '##pée', '##ne']
```

According to the UD convention:

```
['Pouvez', '-vous', 'donner', 'les', 'mêmes', 'garanties', 'à', 'le', 'sein', '
   de', "l'", 'Union', 'Européenne', '?']
```

9. Why is this difference in tokenization a problem for training a `mBERT`-based PoS tagger ?
A word with UD convention that's tokenized as a single unit might be tokenized into multiple subword in mBERT model. This difference directly affects the size of the vocabulary, and the learning and prediction ability.

2.3 Reconciling the Two Tokenizations

10. Write a function that takes as parameter a sentence tokenized according to UD rules (i.e. a variable of type `List[str]` and its gold labels (also a variable of type `List[str]`) and implements the first principle explained above.

```
# first principle: concatenation of the label for multiword tokens
def concat_labels(tokens, labels):
    updated_ud_labels, updated_ud_tokens = [], []
```

```
    i = 0
    while i < len(tokens):
          updated_ud_tokens.append(tokens[i])

          # this is a maker of multiword tokens
          if labels[i] == "_":
             updated_ud_labels.append(labels[i+1] + "+" +labels[i+2])
             i += 2
          else:
             updated_ud_labels.append(labels[i])
          i += 1
    return updated_ud_tokens, updated_ud_labels

# Apply the princeple to the corpus corresponding to the UD dataset
def up_corpus(corpus):
  new_ud_corpus = []

  for tok, tag in corpus:
    new_ud_tokens, new_ud_labels = concat_labels(tok, tag)
    new_ud_corpus.append((new_ud_tokens, new_ud_labels))
  return new_ud_corpus
```

11. Write a function that takes as input the sentences and labels created by "normalizing" UD sentences, apply the mBERT tokenizer and compute the corresponding label. You must ensure each subtoken (including padding symbols) has a label.

```
# second principle: normalization the corresponding labels
from copy import deepcopy

def normalize_labels(corpus,tokenizer, MAX_LENGTH=50):
    """
    corpus is a list of tuples updated (ud_tokens, ud_labels)
    """
    updated_mbert_labels = []
    updated_mbert_corpus = []

    for data in corpus:
        texts = data[0]
        texts_tags = data[1]

        model = tokenizer(texts,
                      is_split_into_words=True,
                      return_offsets_mapping=True,
                      max_length=MAX_LENGTH,
                      padding='max_length',
                      truncation=True)

        tokens = tokenizer.tokenize(texts,
                                is_split_into_words=True,
                                padding=True,
                                truncation=True)
        tokens.insert(0, '<pad>')
        tokens.append('<pad>')

        input_ids = model["input_ids"]
        attention_masks = model["attention_mask"]

```

```
        updated_mbert_labels = [None] * len(model["offset_mapping"])
        original_labels = deepcopy(texts_tags)

        for i in range(len(model["offset_mapping"])):
            if model["offset_mapping"][i][0]!=0 or model["offset_mapping"][i
    ][1]==0:
                  updated_mbert_labels[i] = '<pad>'
            else:
                  updated_mbert_labels[i] = original_labels[0]
                  original_labels.pop(0)
        updated_mbert_corpus.append((tokens, updated_mbert_labels, input_ids,
    attention_masks))
    return updated_mbert_corpus
```

12. Write a function that encodes the labels into integers.

```
# Encodes the mbert labels into integers:  <pad> label must be mapped to -100.
def create_l2i(mbert_corpus_updated):

    all_labels = set()
    labels_to_idx = {}

    for pair in mbert_corpus_updated:
        for labels in pair[1]:
            all_labels.add(labels)
        labels_to_idx = {label: i for i, label in enumerate(all_labels)}
        labels_to_idx["<pad>"] = -100
    return labels_to_idx
```

2.4 Creating a `Dataset`

13. Using the `Dataset.from_list` method, write a method that creates a `Dataset` that encapsulates a corpus in the conllu.

```
# create the dataset
!pip install datasets

from datasets import Dataset
def create_dataset(corpus):
    """
    corpus is a list of tuples (mbert_tokens, mbert_labels, input_ids,
    attention_masks)
    """
    examples = []

    for data in corpus:
        # Encode the labels into integers
        ids_labels = [l2i[label] for label in data[1]]
        examples.append({"input_ids": data[2], "attention_mask": data[3], "
    labels": ids_labels})
    ds = Dataset.from_list(examples)
    return ds
```

14. Create three instances of `Dataset` : one for the train set, one for the dev set and the last one for the test set.

```
# create three instances of the dataset

# train set
train_corpus = list(load_conllu(trainfile))

train_ud_corpus = up_corpus(train_corpus)
train_mbert_corpus = normalize_labels(train_ud_corpus,tokenizer)
l2i = create_l2i(train_mbert_corpus)
ds_train = create_dataset(train_mbert_corpus)

# dev set
dev_corpus = list(load_conllu(devfile))

dev_ud_corpus = up_corpus(dev_corpus)
dev_mbert_corpus = normalize_labels(dev_ud_corpus,tokenizer)
ds_dev = create_dataset(dev_mbert_corpus)

# test set
test_corpus = list(load_conllu(testfile))

test_ud_corpus = up_corpus(test_corpus)
test_mbert_corpus = normalize_labels(test_ud_corpus,tokenizer)
ds_test = create_dataset(test_mbert_corpus)
```

2.5 Fine-Tuning mBERT

15. How can you modify the code of Figure 4 to report the PoS tagging accuracy during optimization. Why is this information important ?

I add a function `compute_metrics` to use with loss as the model evaluation, and pass it to the trainer :

```
# add the evaluation metric
!pip install evaluate

import numpy as np
import evaluate

from transformers import AutoModelForTokenClassification, TrainingArguments,
    Trainer
model_checkpoint = "bert-base-multilingual-cased"
model = AutoModelForTokenClassification.from_pretrained(model_checkpoint,
    num_labels=len(l2i))

# evaluate the model
def compute_metrics(eval_pred):
    metric = evaluate.load("accuracy")
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    predictions = predictions[:,1]
    labels = labels[:,1]
    return metric.compute(predictions=predictions, references=labels)

def train_process(ds_train, ds_dev, batch_size=16):

  training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
```

```
    per_device_eval_batch_size=batch_size,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=10,
    optim = "adamw_torch") # avoid deprecation warning

  trainer = Trainer(
      model=model,
      args=training_args,
      train_dataset=ds_train, # the train set
      eval_dataset=ds_dev, # the dev set
      compute_metrics=compute_metrics
      )
  trainer.train()
  trainer.evaluate()

train_process(ds_train, ds_dev)
```

Epoch	Training Loss	Validation Loss	Accuracy
1	0.135300	0.107672	0.961165
2	0.066700	0.074755	0.983010
3	0.044800	0.069465	0.983010

Figure 2: Loss and accuracy on the french train and dev set

3 Evaluating the multilingual capacity of mBERT

16. Choose 5 languages from the UD project. For each language train a PoS tagger using the code of the previous section and test it on the 5 languages you have chosen.

For the experiment, I chose five languages, they are Vietnamese, Korean, Dutch, Italian and Turkish. Here are the number of training sets, validation sets for each experimental language :

Language	Train set	Dev set
Vietnamese	1400	1123
Korean	4400	950
Dutch	5789	676
Italian	1781	156
Turkish	3435	1100

17. Can you use the same hyperparameters for the different languages ?

For each language, I didn't change the hyperparameters during the training.

18. What can you conclude ?

The findings indicate that among the five languages evaluated, Vietnamese and Turkish yielded accuracies below 80%. In contrast, the remaining three languages—Korean, Italian, and Dutch—surpassed the 90% accuracy threshold. Notably, Dutch achieved the pinnacle of precision with a remarkable 97% accuracy rate.

In terms of corpus size, both Vietnamese and Italian have considerably smaller datasets, each with fewer than 2,000 training examples. Despite this, Italian demonstrates a significantly higher precision rate of 91%, which is markedly greater than that of Vietnamese. In addition, Turkish has more than 3000 training examples, but the accuracy is not much different from that of Vietnamese.
Vietnamese is an isolating language, where word meanings are typically modified through the use of unique words rather than changing word forms directly. On the other hand, Turkish is an agglutinative language characterized by complex words composed of multiple affixes, each denoting distinct grammatical features. mBERT employs subword tokenization, it struggles to capture meanings in uninflected words as in Vietnamese, and similarly faces difficulty in managing the extended word structures inherent to Turkish.

Here are all the results:

Epoch	Training Loss	Validation Loss	Accuracy
1	0.573100	0.547699	0.745325
2	0.394600	0.470056	0.782725
3	0.297800	0.465663	0.795191

Figure 3: Loss and accuracy on the Vietnamese train and dev set

Epoch	Training Loss	Validation Loss	Accuracy
1	0.270400	0.223100	0.934737
2	0.145400	0.192093	0.949474
3	0.135200	0.178400	0.955789

Figure 4: Loss and accuracy on the Korean train and dev set

Epoch	Training Loss	Validation Loss	Accuracy
1	0.139600	0.124084	0.970414
2	0.081400	0.107161	0.977811
3	0.049300	0.108571	0.976331

Figure 5: Loss and accuracy on the Dutch train and dev set

Epoch	Training Loss	Validation Loss	Accuracy
1	0.180000	0.196991	0.903846
2	0.110600	0.159628	0.923077
3	0.099000	0.152014	0.916667

Figure 6: Loss and accuracy on the Italian train and dev set

Epoch	Training Loss	Validation Loss	Accuracy
1	0.520900	0.541689	0.738182
2	0.366900	0.454626	0.787273
3	0.283200	0.430282	0.795455

Figure 7: Loss and accuracy on the Turkish train and dev set