Multilingual NLP
Lab 2 – How Multilingual is Multilingual BERT?

AN Ji

November 20, 2024

1 Introduction

Installation:

- `HuggingFace`: `transformer`, `datasets`

2 Fine-Tuning mBERT

Goals:

- Evaluate the performance of a PoS tagger fine-tuned on a given language and compare it to the performance of a PoS tagger trained only on this language
- Evaluate on a language B the performance of a PoS tagger fine-tuned on a language A

2.1 The Universal Dependencies Project

1. **Why do we need "consistent annotation" for our experiment?**

   For this experiment, we need to train a PoS tagger, evaluate and compare its performance on different languages. Applying a common set of PoS tags cross-linguistically allows us to do the job correctly and more easily, as well as other comparative morphosyntactic researches using UD projects. It's therefore crucial to being "uniform", although this method could imply some kind of compromise between languages that differs a lot in some subtle morphosyntactic aspects.

2. **Why do we use the `yield` keyword rather than a simple `return` (line 7) ? Why do we have to call the `list` constructor (line 9)?**

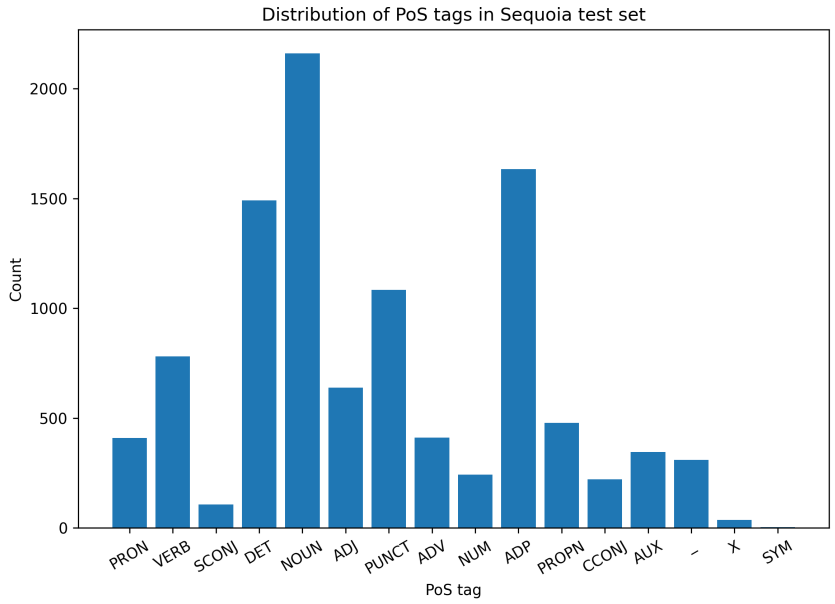
```
1 import conllu
2
3 def load_conllu(filename):
4     for sentence in conllu.parse(open(filename, "rt",
          encoding="utf-8").read()):
5         tokenized_words = [token['form'] for token in sentence]
6         gold_tags = [token["upos"] for token in sentence]
7         yield tokenized_words,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

   The `return` keyword requires to load all data into memory at once before returning a list, which may often be intensive to memory use, especially when we should deal with extremely large datasets and haven't decided how many data to process with that result later. Using `yield` to

create a generator first instead of returning directly a huge list of sentences is a better and versatile choice on both memory management and data flexibility. So when we decide to use all data, we can simply combine `list()` with `yield` to transform the generator into a full list of sentences, like what we do in line 9.

3. **Distribution of labels in the test set of the Sequoia treebank**

```
1  from collections import Counter
2  import matplotlib.pyplot as plt
3
4  def plot_pos_distribution(corpus):
5      """plot the distribution of PoS tags in the corpus"""
6      pos_counts = Counter(tag for _, tags in corpus for tag in tags)
7      plt.figure(figsize=(9, 6))
8      plt.bar(pos_counts.keys(), pos_counts.values())
9      plt.xlabel("PoS tag")
10     plt.ylabel("Count")
11     plt.xticks(rotation=30)
12     plt.title("Distribution of PoS tags in Sequoia test set")
13     plt.show()
14
15 plot_pos_distribution(corpus)
```



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

```
1  multi_tokens = [token.lower() for tokens, tags in corpus for token, tag in
       zip(tokens, tags) if tag == '_']
2  dict(Counter(multi_tokens))
```

```
{'des': 114, 'du': 115, 'aux': 22, 'au': 58, 'desdites': 1}
```

From the `conllu` file of Sequoia test set, we find the following annotation rule for the multiword tokens above: the token itself is marked with an underscore as `upos`, followed by its 2 constituents labeled as `ADP` (adposition) and `DET` (determiner) respectively. For example, `desdites` in the sentence of line 195 and `des` in the sentence of line 451:

`En tant que remarque isolée, je voudrais donc remercier Mme Cederschiöld du fait qu'elle ait clarifié par exemple la notion desdites copies temporaires dans son avis.`

```
ID      FORM       LEMMA   UPOS
24-25   desdites       _      _
24      de            de    ADP
25      lesdites   ledit    DET
```

Comme vous tous, j'entends les appels à l'aide des mères, des épouses et des soeurs des Kosovars albanais qui croupissent dans les geôles serbes.

```
ID      FORM   LEMMA   UPOS
12-13   des        _      _
12      de        de    ADP
13      les       le    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?**

   Since it is emphasized that the basic units of annotation are **syntactic words** (not phonological or orthographic words) [1], French contractions are actually not compatible with any of the PoS tags defined in UD tokenization guidelines. They are mostly composed of a preposition (marked as ADP, like *à*, *de*) and a determiner (marked as DET, like *le*, *les*). If we choose only one, it would become unavoidable to lose important syntactic information of the ignored component; if we introduce a new tag for them, we will be forced to decrease consistency and comparability from a cross-linguistic perspective. Hence splitting these forms seems to be rather a reasonable solution (a step towards de-specialization?).

6. Tokens with spaces inside are numbers:

```
tokens_spaces = {token for tokens, _ in corpus for token in tokens if ' ' in
    token}
tokens_spaces

[Out]:

{'100 000','15 000','190 500','2 000','3 092','3 852','3 862','50 000','500
    000','67 025','7 736','80 000','800 000'}
```

   Remove spaces:

```
corpus = [([token.replace(' ', '') for token in tokens], tags) for tokens,
    tags in corpus]
```

2.2 mBERT Tokenization

7. **Why does mBERT use a subword tokenization?**

   Subword tokenization algorithms, here **WordPiece** used in mBERT, are able to keep frequently used words as tokens on the one hand, and break down relatively less used and unseen words into frequently occuring subword units that the model has seen and learned somewhere in the pretraining data. This approach reduces notably vocabulary size and memory use while maximizing the model's semantic representation of data, making it more efficient in multilingual text processing (e.g. for morphologically rich languages).

8. **How is the sentence *Pouvez-vous donner les mêmes garanties au sein de l'Union Européenne* tokenized according to the UD convention? How is it tokenized by mBERT tokenizer?**

[1] UD Guidelines – Tokenization and word segmentation

```
# Tokenization by UD convention:

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

# Tokenization by mBERT tokenizer:

!pip install transformers
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
sentence = "Pouvez-vous donner les mêmes garanties au sein de l'Union
    Européenne ?"
print(tokenizer.tokenize(sentence))

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

9. **Why is this difference in tokenization a problem for training a mBERT-based PoS tagger?**

   Because if we want to train an mBERT-based PoS tagger, we had better ensure that all our training tokens conform with mBERT's inherent vocabulary forms so that knowledge about them already mastered by mBERT can be optimally transferred and facilitate our training. If we don't solve this difference, then the model's pre-training vocabulary advantages would not be maximally used and it will produce a significant impact upon final performance, since it means increasing vocabulary size as well as built-in mappings unnecessarily.

2.3 Reconciling 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.**

```
def process_ud_multiwords(tokens, tags):
    """
    multiword tokens: keep contractions unchanged.
    E.g. 'aux','à','les' -> 'aux'
    multiword tags: replace '_' with the following 2 tags.
    E.g. '_','ADP','DET' -> 'ADP+DET'
    """
    updated_tokens, updated_tags = list(), list()
    i = 0
    while i < len(tokens):
        updated_tokens.append(tokens[i])
        if tags[i] == '_':
            updated_tags.append(f'{tags[i+1]}+{tags[i+2]}')
            i += 3
        else:
            updated_tags.append(tags[i])
            i += 1
    return updated_tokens, updated_tags
```

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.**

```
# implement 2nd principle
```

```
def tokenize_align_tags(corpus, tokenizer=tokenizer, MAX_LENGTH=64):
    """
    Let mBERT re-tokenize each token, align sub-tokens with tags, pad
        sentences to MAX_LENGTH, save model inputs (& raw tokens/tags) of each
        sentence in a dict, then all dicts in a list
    """
    all_sentences = list()
    for tokens, tags in corpus:
        sentence = dict()
        model_inputs = tokenizer(
            tokens,
            is_split_into_words=True,
            # return_offsets_mapping=True,  # no need if using word_ids
            # truncation=True,              # no need if setting max_length
            padding='max_length',
            max_length=MAX_LENGTH,
        )
        # save model inputs
        sentence['subtokens'] = model_inputs.tokens()
        sentence['input_ids'] = model_inputs['input_ids']
        sentence['attention_mask'] = model_inputs['attention_mask']
        # use for tag alignment e.g. [None, 0, 0, 0, 1, 1, 2, 3, 4, 4, 5, None]
        word_ids = model_inputs.word_ids()

        aligned_tags = []
        previous_word_id = None

        for word_id in word_ids:
            if word_id is None: # [CLS], [SEP] & [PAD]
                aligned_tags.append('<pad>')
            elif word_id != previous_word_id: # word head
                aligned_tags.append(tags[word_id])
            else: # rest of word
                aligned_tags.append('<pad>')

            previous_word_id = word_id

        sentence['tags'] = aligned_tags
        all_sentences.append(sentence)
    return all_sentences
```

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

    See 14. For convenience I handle this step after creating the multilingual dataset.

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`.**

    Here I combined functions defined above to transform a `conllu` corpus to a `Dataset`:

```
def conllu2ds(lang, split):
    """
    Encapsulate one conllu corpus into HuggingFace datasets
    """
    filename = pick_conllu(lang, split)
    corpus = list(load_conllu(filename))
    updated_corpus = [process_ud_multiwords(tokens, tags) for tokens, tags in
        corpus]
    all_sentences = tokenize_align_tags(updated_corpus, tokenizer)
    dataset = Dataset.from_list(all_sentences)
```

```
    return dataset
```

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

```
code2lang = {
    'zh':   'Chinese',
    'yue':  'Cantonese',
    'ja':   'Japanese',
    'ug':   'Uyghur',
    'sa':   'Sanskrit',
    'th':   'Thai',
}

def encode_ds_tags():
    """
    First combine all language datasets, then encode all possible tags into integers
    """
    ds_dict = DatasetDict()
    for code in code2lang.keys():
        ds = conllu2ds(code, 'train') if code == 'ug' else conllu2ds(code, 'test')
        ds_dict.update({code: ds})

    unique_tags = set([tag for _, ds in ds_dict.items() for sent in ds for tag in sent['tags'] if tag != '<pad>'])
    unique_tags = sorted(list(unique_tags))
    tag2int = {tag: i for i, tag in enumerate(unique_tags)}
    tag2int['<pad>'] = -100
    # print(f"Overall 'meaningful' unique tags: {unique_tags}")
    # print(f"Final mapping of tags to integers (including '<pad>'): {tag2int}")
    # print(f"Final total of unique tags (including '<pad>'): {len(tag2int)}")

    for lang, ds in ds_dict.items():
        ds = ds.map(lambda x: {'labels': [tag2int[tag] for tag in x['tags']]})
        ds_dict[lang] = ds

    return ds_dict


# Generate an integer-labeled dataset for all languages
ds_dict = encode_ds_tags()
ds_dict

[Out]:
Overall 'meaningful' unique tags:
['ADJ', 'ADP', 'ADV', 'AUX', 'CCONJ', 'DET', 'INTJ', 'NOUN', 'NUM', 'PART', 'PRON', 'PROPN', 'PUNCT', 'SCONJ', 'SYM', 'VERB', 'X']
Final mapping of tags to integers (including '<pad>'):
{'ADJ': 0, 'ADP': 1, 'ADV': 2, 'AUX': 3, 'CCONJ': 4, 'DET': 5, 'INTJ': 6, 'NOUN': 7, 'NUM': 8, 'PART': 9, 'PRON': 10, 'PROPN': 11, 'PUNCT': 12, 'SCONJ': 13, 'SYM': 14, 'VERB': 15, 'X': 16, '<pad>': -100}
Final total of unique tags (including '<pad>'):
18

DatasetDict({
    zh: Dataset({
        features: ['subtokens', 'input_ids', 'attention_mask', 'tags', 'labels'],
        num_rows: 1004
    })
```

```
    yue: Dataset({
        features: ['subtokens', 'input_ids', 'attention_mask', 'tags',
            'labels'],
        num_rows: 1004
    })
    ja: Dataset({
        features: ['subtokens', 'input_ids', 'attention_mask', 'tags',
            'labels'],
        num_rows: 1000
    })
    ug: Dataset({
        features: ['subtokens', 'input_ids', 'attention_mask', 'tags',
            'labels'],
        num_rows: 1656
    })
    sa: Dataset({
        features: ['subtokens', 'input_ids', 'attention_mask', 'tags',
            'labels'],
        num_rows: 2709
    })
    th: Dataset({
        features: ['subtokens', 'input_ids', 'attention_mask', 'tags',
            'labels'],
        num_rows: 1000
    })
})

def split_dataset(ds, train_size=0.8, seed=42):
    splits = ds.train_test_split(train_size=train_size, seed=seed,
        shuffle=True)
    train_ds, temp_ds = splits['train'], splits['test']
    temp = temp_ds.train_test_split(train_size=0.5, seed=seed)
    dev_ds, test_ds = temp['train'], temp['test']

    return DatasetDict({
        'train': train_ds,
        'dev': dev_ds,
        'test': test_ds
    })

for lang, ds in ds_dict.items():
    ds_dict[lang] = split_dataset(ds)
```

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?**

    We can add a `compute_metrics` function to design the evaluation steps and pass it to the `Trainer` function as an argument. It's crucial because we need to check the reported accuracy and loss on both train set and dev set to make sure the model converges well instead of ending up with overfitting. We also need to use accuracy as metric to report the performance on all test sets.

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.**

    The following languages are chosen.
    – Cantonese (`yue`, Sino-Tibetan, Chinese): HK

– Chinese (zh, Sino-Tibetan, Chinese): also HK, **parallel with Cantonese**
– Japanese (ja, Japanese): PUD
– Thai (th, Tai-Kadai): also PUD, **parallel with Japanese**
– Uyghur (ug, Turkic, Southeastern): UDT
– Sanskrit (sa, Indo-European, Indic): Vedic

```
def compute_metrics(pred):
    """ metrics computing setup"""
    metrics = evaluate.load("accuracy")
    logits, labels = pred
    predictions = np.argmax(logits, axis=-1)
    predictions = predictions.flatten()
    labels = labels.flatten()
    mask = labels != -100

    return metrics.compute(predictions=predictions[mask],
        references=labels[mask])

# training settings

batch_size = 16
model_checkpoint = "bert-base-multilingual-cased"

results = {lang: {test_lang: None for test_lang in ds_dict.keys()} for lang in
    ds_dict.keys()}

for lang, ds in ds_dict.items():
    ds_train, ds_dev, ds_test = ds['train'], ds['dev'], ds['test']
    model = AutoModelForTokenClassification.from_pretrained(
        model_checkpoint,
        num_labels=18,
    )

    training_args = TrainingArguments(
        output_dir=f"./results/{lang}",
        eval_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=f"./results/logs",
        logging_steps=10,
        report_to="none",
        push_to_hub=True,
        hub_model_id=f"ankei/{lang}_pos_tagger",
        hub_strategy="end",
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=ds_train,
        eval_dataset=ds_dev,
        compute_metrics=compute_metrics,
    )
    print(f"Training {code2lang[lang]} ...")
    trainer.train()
    trainer.evaluate()


    for test_lang, test_ds in ds_dict.items():
```

```
        test_results = trainer.predict(test_ds["test"])
        test_accuracy = test_results.metrics["test_accuracy"]
        results[lang][test_lang] = test_accuracy
        print(f"Accuracy of {code2lang[lang]} model on {code2lang[test_lang]}
            test set: {test_accuracy:.4f}")
```

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

    Theoretically yes, and I didn't change the hyperparameters for the final version of fine-tuning (i.e. using `batch_size=16`, `epoch=3`, `max_length=64` for padding and truncation). However, I tested a session out of curiosity using `epoch=10` to see how the models would perform, and I found the overall loss and accuracy became the most stable around 5th epoch and started to fluctuate heavily after. So based on this, I thought maybe for each trained language we could increase epoch accordingly to make each model better, but after all since the datasets are limited to be small on purpose (from 1000 to 2700), it is hardly possible to boost performance only by modifying epoch, which might not help much in terms of models' cross-linguistic generalization ability.

18. **What can you conclude?**

    Language-specific performance

    For each language's sake, the model fine-tuned on Japanese yields the best predictive ability (95.8%), followed by Chinese (91.0%), Cantonese (85.7%) and Thai (85.2%). Japanese, Chinese and Thai are well supported and represented during mBERT's pre-training process because of their high-quality Wikipedia contents, so their performance is less surprising. Interestingly, Cantonese was not officially chosen[2], but the Cantonese model is still not bad compared to Thai model (at least their performance gap is usually slight after comparing several trainings). Reasons for that are various, but the most important one may be that Cantonese benefits a lot from Chinese since there is considerable vocabulary overlap. In addition, we used parallel corpora for these two languages, which are mainly composed of Hong Kong student film subtitles and Hong Kong Legislative Council's meeting records. Given that the Chinese subtitles still contain many Cantonese-specific words, meetings tend to remain lexically-formal even though they were held in Cantonese before being translated into Chinese records, and mBERT's Chinese pre-training also relies largely on Traditional Chinese contents that are de facto an amalgam of different non-mainland Chinese varieties (at least so lexically), it's no wonder that Cantonese model did well.

    While the fine-tuned model excels on the above languages, we should also notice that for the rest – Uyghur and Sanskrit, that mBERT has not been explicitly pre-trained on, it still yields good results with an accuracy of 74.5% and 77.0% respectively after learning from only 1490 and 2438 sentences. On the other hand, they are also the most difficult to predict. By now we could not explain granularly enough on these observations, perhaps numerous factors can account for them more or less: lack of pre-training resources, quality and size of corpus used for fine-tuning, morphological features, alphabetical idiosyncrasies, etc. It would be reasonable to expect a much better performance using more data. We could say that these results benefit a lot from the **WordPiece** subword tokenization strategy that mBERT applied to set up and refine its vocabulary, which enables it to handle low-resource synthetic languages (Uyghur and Sanskrit here) more efficiently.

    Cross-linguistic generalization

    – Chinese model produced the most spread out performances on other languages (from Sanskrit 10.8% to Cantonese 78.1%), followed by Cantonese model (from Sanskrit 25.4% to Chinese 82.0%).

[2] See https://github.com/google-research/bert/blob/master/multilingual.md#list-of-languages

Trained on Tested on	zh	yue	ja	ug	sa	th
zh	0.910364	0.819795	0.612512	0.401494	0.324930	0.428571
yue	0.780992	0.857438	0.547521	0.393251	0.265840	0.363636
ja	0.560086	0.457797	0.957797	0.424177	0.293991	0.404149
ug	0.250252	0.278449	0.442598	0.744713	0.389225	0.286506
sa	0.108095	0.254195	0.229023	0.346989	0.770484	0.201876
th	0.418750	0.382955	0.373864	0.293182	0.336932	0.851705

Table 1: Performance of fine-tuned pos-taggers

This may indicate that models fine-tuned on morphologically-simple (isolating) languages would predict less accurately on morphologically-complex ones.

– Models trained on Chinese and Cantonese that are two members of sinitic languages show high transferability between them. This is expected given their close linguistic relatedness.

– If we ignore the reciprocal high transferability between Chinese and Cantonese, there is an obvious transfer gap of about 33%–43% between each language model's performances on its dominant language and the best-performed other language. Additionally, we can also observe that models trained on languages with relatively rich morphological features (like Uyghur and Sanskrit) yield narrowest generalization performance range (12%–13%), while models trained on more isolating languages spread out much more heavily (Chinese 45.0%, Japanese 38.4%, Thai 22.7% and Cantonese 20.4%). Is it a hint that when predicting other languages, models based on morphologically complex languages tend to produce smaller deviation than those trained on simple ones?

– Horizontally speaking, Sanskrit, an ancient language with a rich inflectional morphology, is the least predictable among all, while Japanese (agglutinating, and a bit isolating?) and Chinese (isolating) are the opposite (although the performances on them are still limited).

– Thai model seems to be struggling to generalize since it shows largest transfer gap of at least 42% when predicting other languages. More interestingly, it reports the worst score on Sanskrit from which Thai historically borrowed and derived a lot of words.

The above results seem to reflect some kind of mapping (or correlation) between morphosyntactic distinctions and language-specific fine-tuned model's cross-linguistic generalization ability. We can say that mBERT is multilingual enough in terms of its compatibility and performance on languages it is fine-tuned on, but mBERT's transferability proves to be relatively limited and might correlate with typological relatedness between languages.