How Multilingual is Multilingual BERT?

Multilingual NLP - M2 Linguistique Informatique

Zeinab Sheikhitarghi

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

With a consistent tagging scheme like Universal POS tags, we ensure that the model's output in both training and evaluation languages is directly comparable. For example, a tag like VERB has the same linguistic meaning in both languages. Besides, We can assess the model's ability to transfer learned knowledge from one language to another. For example, if we train mBERT on Persian and then test it on Arabic, we can evaluate how well the model understands and transfers the concept of different POS tags from Persian to Arabic. When comparing performance metrics like accuracy across different languages, consistent annotation ensures that we are measuring the same linguistic phenomena.

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) ?

Using *yield* allows the function to return one sentence at a time, instead of returning all sentences at once. This is especially useful if the CoNLL-U file is large, as it prevents loading the entire file into memory. Generators can only be iterated over once; once we reach the end, we cannot iterate over them again without reinvoking the generator. By converting the generator output to a list, we can iterate over the dataset multiple times.

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

Figure 2 shows the distribution of labels in this dataset.

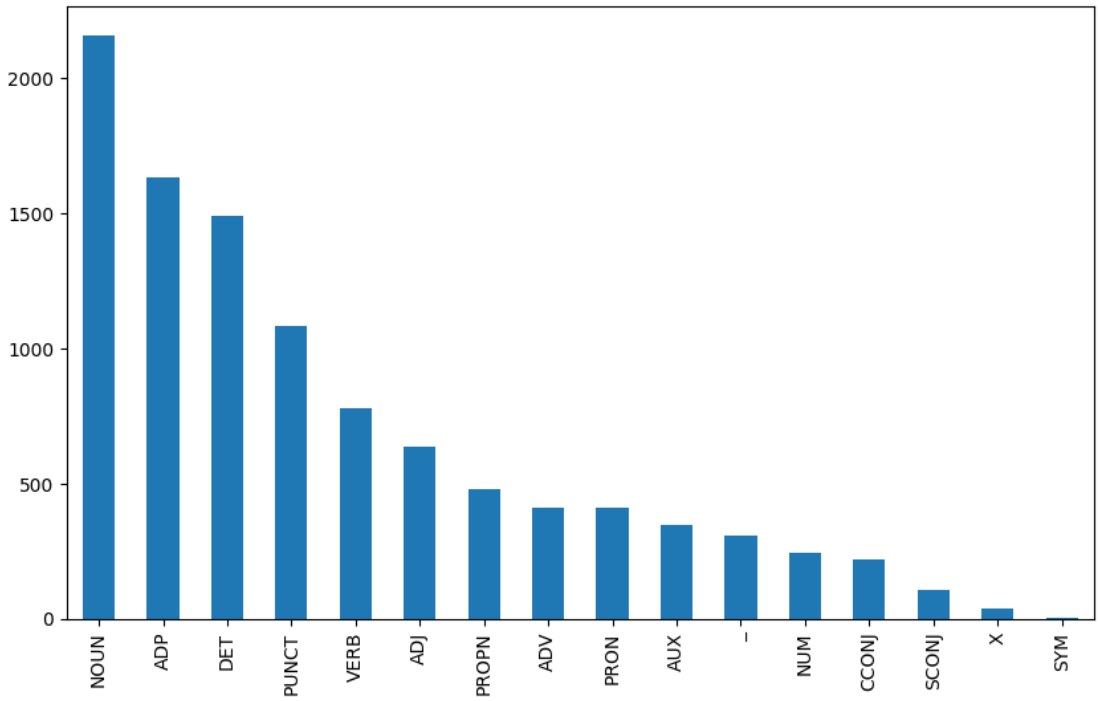

Figure 1: Distribution of labels in the test set of Sequoia treebank

```
corpus = list(load_conllu(filename=FR_TEST_PATH))
label_distribution = pd.Series([tag for ids, tokens, tags in corpus for tag in tags]).value_counts()
label_distribution.plot(kind='bar', figsize=(10, 6))
```

Figure 2: Code for generating distribution of labels

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

As show in the Figure 3 the multiword tokens are: 'au', 'aux', 'DES', 'des', 'du', 'Aux', 'desdites', 'Au'.

Multiword tokens are identified using the id column of the Universal Dependencies CoNLL-U file. In this context, the id for a multiword token is represented as a tuple, such as (8, '-', 9), which indicates the start and end Ids of its components.

```
corpus = list(load_conllu(filename=FR_TEST_PATH))
multiwords = [token for ids, tokens, gold_labels in corpus for id_, token in zip(ids, tokens)  if isinstance(id_, tuple)]
set(multiwords)
```

Figure 3: Recognizing multiword tokens in the corpus

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

For NLP tasks such as part-of-speech tagging, parsing, and machine translation, models typically rely on word-level representations. By splitting multiword tokens, the UD project simplifies the integration of these annotations with models that expect word-level inputs, making the training and evaluation process more straightforward. It also ensures consistency in linguistic annotation across diverse languages, providing a more granular analysis of syntax and morphology, which is valuable for capturing linguistic nuances. In my opinion, a challenge arises when aligning UD's word-level annotations with the tokenizers used by most modern NLP models. These models often use subword tokenization methods. These tokenizers break down words into smaller subword units, especially for words that are rare or morphologically complex. As a result, aligning the UD's split tokens with subword tokenization can be complex, since there might not be a direct one-to-one mapping between the UD's tokens and the model's tokenized output.

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.

There exists tokens(numbers with more than 3 digits) which contain spaces. Figure 4 shows the code for removing any spaces within tokens and asserting that there will be no token with whitespace in the corpus.

```
corpus = list(load_conllu(filename=FR_TEST_PATH))
corpus = [(ids, [token.replace(" ", "") for token in tokens], tags) for ids, tokens, tags in corpus]
for _, tokens, tags in corpus:
    for token in tokens:
        assert " " not in token, f'Token with Whitespace Found: {token}'
```

Figure 4: Removing spaces from tokens

7. Why does mBERT use a subword tokenization?

Subword tokenization algorithms enable better handling of morphologically rich

languages and out-of-vocabulary words. Using the subword tokenization we can solve the issues faced by word-based tokenization (very large vocabulary size, large number of OOV tokens, and different meaning of very similar words) and character-based tokenization (very long sequences and less meaningful individual tokens). Subword tokenization allows the model to have a decent vocabulary size and also be able to learn meaningful context-independent representations.

8. How is the sentence "Pouvez-vous donner les mêmes garanties au sein de l'Union Européene"4 tokenized according to the UD convention? How is it tokenized by mBERT tokenizer?

The sentence is tokenized as below according to the UD convention.
['Pouvez', '-vous', 'donner', 'les', 'mêmes', 'garanties', 'au', 'à', 'le', 'sein', 'de', "l'", 'Union', 'Européenne']

Tokenizing this sentence with mBert tokenizer returns:
['Po', '##uve', '##z', '-', 'vous', 'donner', 'les', 'mêmes', 'gara', '##nties', 'au', 'sein', 'de', 'l', '[UNK]', 'Union', 'Euro', '##pée', '##ne']

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

A single word or phrase in the original dataset might be split into multiple subwords by mBERT. The Sequoia corpus provides PoS tags for each word or meaningful token, but mBERT uses subword tokenization that can split a single word into several sub-parts (subwords). Mapping the gold standard PoS tags from the Sequoia dataset to the subword tokens generated by mBERT is non-trivial. "[UNK]" in mBert represents an unknown token and this makes it difficult to determine the correct PoS tag for those unknown tokens.

10. Write a function that takes as parameter a sentence tokenized according to UD rules and implements the first principle.

The Figure 5 shows the implementation of this method.

```python
def assign_label_to_multitoken(ids: List[int], tokens: List[str], gold_labels: List[str]) -> Tuple[List[str], List[str]]:

    df = pd.DataFrame({
        'id': ids,
        'token': tokens,
        'pos': gold_labels
    })

    final_tokens = []
    final_pos_tags = []

    skip_indices = set()

    for _, row in df.iterrows():
        id_ = row['id']

        if isinstance(id_, tuple):
            start, end = id_[0], id_[2]
            skip_indices.update(range(start, end + 1))
            pos_tags = df.loc[
                (df['id'].apply(lambda x: isinstance(x, int) and start <= x <= end)), 'pos'
            ]
            concatenated_pos = '+'.join(pos_tags)
            final_tokens.append(row['token'])
            final_pos_tags.append(concatenated_pos)
        else:
            if id_ not in skip_indices:
                final_tokens.append(row['token'])
                final_pos_tags.append(row['pos'])

    return final_tokens, final_pos_tags
```

Figure 5: Function for assigning concatenated label to a multiword token

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.

Figure 6 shows the implemented function for this task. Note that since the tokenizer has a MAX_LENGTH parameter, tokens exceeding this length will be truncated. It is important to apply the same truncation to the labels to maintain alignment between tokens and their corresponding labels.

```python
def align_labels(texts: List[List[str]], gold_labels: List[List[str]], pos2id: dict):
    max_length = 512
    info = tokenizer(
        texts,
        return_offsets_mapping=True,
        is_split_into_words=True,
        padding=True,
        truncation=True,
        max_length=max_length
    )
    new_labels = []

    for index, offsets in enumerate(info['offset_mapping']):
        labels = gold_labels[index]
        for i, (start, end) in enumerate(offsets):
            if start == 0 and end == 0:
                labels.insert(i, PAD_TAG)
            if start != 0:
                labels.insert(i, PAD_TAG)

        new_labels.append(labels[:max_length])



    aligned_data = [
        {"input_ids": input_ids, "attention_mask": attention_mask, "labels": labels}\
        for input_ids, attention_mask, labels in zip(info['input_ids'], info['attention_mask'], new_labels)
    ]

    for entry in aligned_data:
        entry["labels"] = label2int(entry["labels"], pos2id)

    return aligned_data
```

Figure 6: Function for aligning labels with tokens resulted from tokenization

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

```python
def label2int(labels: List[str], label2ids: Dict[str, int]) -> List[int]:
    return [label2ids.get(label, -100) for label in labels]
```

Figure 7: Function for encoding labels into integers

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

In the **create_dataset** in Figure8, we read the CoNLLU file. Next, we remove any whitespace that exists within the tokens. After that, we assign the appropriate label to multiword tokens by merging their labels. When creating the training dataset, it is crucial to generate pos_labels and the corresponding pos2id dictionary,

as this mapping will also be used for the dev and test sets. Then, we call the align_labels function, which aligns the tokens resulting from subword tokenization with their labels from the UD project. The pos2id dictionary is used to convert all labels into their corresponding numerical IDs.

```
def create_dataset(data_file_path, pos2id=None, split="train"):
    corpus = list(load_conllu(data_file_path))
    corpus = [(ids, [token.replace(" ", "") for token in tokens], tags) for ids, tokens, tags in corpus]
    corpus = [assign_label_to_multitoken(ids, tokens, gold_labels) for ids, tokens, gold_labels in corpus]

    texts = [tokens for tokens, _ in corpus]
    gold_labels = [labels for _, labels in corpus]

    if split == "train":
        pos_labels = sorted(set(label for labels in gold_labels for label in labels if label))
        pos2id = {label: idx for idx, label in enumerate(pos_labels)}

    aligned_corpus = align_labels(texts, gold_labels, pos2id)
    return Dataset.from_list(aligned_corpus), pos2id
```

Figure 8: Function for creating a 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.

```
train_dataset, pos2id = create_dataset(AR_TRAIN_PATH, split="train")
dev_dataset, _ = create_dataset(AR_DEV_PATH, pos2id=pos2id, split="dev")
test_dataset, _ = create_dataset(AR_TEST_PATH, pos2id=pos2id, split="test")
```

Figure 9: Function for

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 create a custom metric function(Figure 10) and pass it as the value to the **compute_metrics** argument of the **Trainer** class. This function should calculate the accuracy based on the predictions and labels for Part-of-Speech tags. This information is important because it allows us to monitor the model's performance on the PoS tagging task throughout the training process. **Trainer** in the Figure 11 evaluates and displays PoS tagging accuracy after each evaluation step.

```python
def compute_metrics(pred):
    logits, labels = pred
    predictions = np.argmax(logits, axis=-1)

    true_labels = labels[labels != -100]
    true_predictions = predictions[labels != -100]

    accuracy = accuracy_score(true_labels, true_predictions)
    return {"accuracy": accuracy}
```

Figure 10: Helper method for computing accuracy

```python
def train_pos_tagger(model, train_set, dev_set, output_dir):

    training_args = TrainingArguments(
        output_dir=output_dir,
        eval_strategy = "epoch",
        learning_rate=2e-5,
        per_device_train_batch_size=BATCH_SIZE,
        per_device_eval_batch_size=BATCH_SIZE,
        num_train_epochs=5,
        weight_decay=0.01,
        logging_steps=10)

    trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_set,
    eval_dataset=dev_set,
    compute_metrics=compute_metrics)
    trainer.train()
    model.save_pretrained(training_args.output_dir)
```

Figure 11: Trainer and its argument

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 5 chosen languages are: **Arabic, English, Farsi, French, and Japenese**. Each model has been trained on each of these languages using the corresponding train and dev datasets. Figure12 shows the steps for creating the datasets, initializing the model, training the model and evaluating the model on the test dataset.

```
train_dataset, pos2id = create_dataset(AR_TRAIN_PATH, split="train")
dev_dataset, _ = create_dataset(AR_DEV_PATH, pos2id=pos2id, split="dev")
test_dataset, _ = create_dataset(AR_TEST_PATH, pos2id=pos2id, split="test")


model = AutoModelForTokenClassification.from_pretrained(CHECKPOINT, num_labels=len(pos2id.keys()))


train_pos_tagger(model, train_dataset, dev_dataset, output_dir=AR_MODEL_PATH)


evaluate_model(test_dataset=test_dataset, model_path=AR_MODEL_PATH)
```

Figure 12: Steps for training a mBert model on Arabic language

The below table shows the POS labels for each language. Number of unique POS tag for each langauge is:

Arabic: **110**
English: **37**
Farsi: **37**
French: **18** Japanese: **16**

Language	List of POS
Arabic	ADJ, ADJ+ADP, ADJ+ADP+NOUN, ADJ+ADP+PRON, ADJ+PRON, ADP, ADP+ADJ, ADP+ADJ+PRON, ADP+ADP, ADP+CCONJ, ADP+DET, ADP+NOUN, ADP+NOUN+ADP, ADP+NOUN+DET, ADP+NOUN+PRON, ADP+NUM, ADP+PART, ADP+PRON, ADP+PROPN, ADP+SCONJ, ADP+SCONJ+PART, ADP+SCONJ+PRON, ADP+X, ADV, ADV+DET, AUX, AUX+AUX, AUX+VERB, AUX+VERB+PRON, CCONJ, CCONJ+ADJ, CCONJ+ADJ+PRON, CCONJ+ADP, CCONJ+ADP+ADJ, CCONJ+ADP+ADP, CCONJ+ADP+CCONJ, CCONJ+ADP+CCONJ+DET, CCONJ+ADP+DET, CCONJ+ADP+NOUN, CCONJ+ADP+NOUN+PRON, CCONJ+ADP+PART, CCONJ+ADP+PRON, CCONJ+ADP+SCONJ, CCONJ+ADP+SCONJ+PRON, CCONJ+ADV, CCONJ+AUX, CCONJ+AUX+AUX, CCONJ+AUX+VERB, CCONJ+AUX+VERB+PRON, CCONJ+CCONJ, CCONJ+CCONJ+DET, CCONJ+CCONJ+PRON, CCONJ+CCONJ+VERB, CCONJ+CCONJ+VERB+PRON, CCONJ+DET, CCONJ+DET+ADP, CCONJ+DET+PRON, CCONJ+INTJ, CCONJ+NOUN, CCONJ+NOUN+ADP, CCONJ+NOUN+PRON, CCONJ+NUM, CCONJ+NUM+NUM, CCONJ+PART, CCONJ+PART+AUX, CCONJ+PART+NOUN, CCONJ+PART+PART, CCONJ+PART+PRON, CCONJ+PART+VERB, CCONJ+PRON, CCONJ+PROPN, CCONJ+SCONJ, CCONJ+SCONJ+PRON, CCONJ+VERB, CCONJ+VERB+PRON, CCONJ+X, DET, DET+PRON, DET+VERB, INTJ, NOUN, NOUN+ADJ, NOUN+DET, NOUN+NOUN, NOUN+PRON, NOUN+PUNCT, NUM, PART, PART+ADJ, PART+ADV, PART+AUX, PART+CCONJ, PART+NOUN, PART+NOUN+DET, PART+PART, PART+PRON, PART+VERB, PART+VERB+PRON, PRON, PROPN, PUNCT, SCONJ, SCONJ+PART, SCONJ+PRON, SYM, VERB, VERB+ADP, VERB+PRON, X, X+DET, X+NOUN

English	'ADJ', 'ADJ+PART', 'ADP', 'ADP+ADP', 'ADV', 'ADV+AUX', 'AUX', 'AUX+AUX', 'AUX+PART', 'AUX+PART+VERB', 'CCONJ', 'DET', 'DET+NOUN', 'INTJ', 'NOUN', 'NOUN+ADP', 'NOUN+AUX', 'NOUN+PART', 'NUM', 'NUM+PART', 'PART', 'PRON', 'PRON+AUX', 'PRON+PART', 'PRON+VERB', 'PROPN', 'PROPN+AUX', 'PROPN+PART', 'PROPN+PROPN', 'PUNCT', 'SCONJ', 'SYM', 'VERB', 'VERB+ADV', 'VERB+PART', 'VERB+PRON', 'X'
Persian	'ADJ', 'ADJ+AUX', 'ADJ+PRON', 'ADJ+VERB', 'ADP', 'ADP+DET', 'ADP+PRON', 'ADP+PRON+AUX', 'ADV', 'ADV+AUX', 'ADV+PRON', 'ADV+PRON+VERB', 'ADV+VERB', 'AUX', 'AUX+PRON', 'CCONJ', 'DET', 'INTJ', 'NOUN', 'NOUN+AUX', 'NOUN+PRON', 'NOUN+PRON+AUX', 'NOUN+SCONJ', 'NOUN+VERB', 'NUM', 'NUM+PRON', 'PART', 'PRON', 'PRON+AUX', 'PRON+PART', 'PRON+PRON', 'PRON+VERB', 'PUNCT', 'SCONJ', 'VERB', 'VERB+PRON', 'X'
French	'ADJ', 'ADP', 'ADP+DET', 'ADP+PRON', 'ADV', 'AUX', 'CCONJ', 'DET', 'INTJ', 'NOUN', 'NUM', 'PRON', 'PROPN', 'PUNCT', 'SCONJ', 'SYM', 'VERB', 'X'
Japanese	'ADJ', 'ADP', 'ADV', 'AUX', 'CCONJ', 'DET', 'INTJ', 'NOUN', 'NUM', 'PART', 'PRON', 'PROPN', 'PUNCT', 'SCONJ', 'SYM', 'VERB'

The results of training mBert models on a specific language and evaluating it on the dev dataset are shown in below tables.

Epoch	Training Loss	Validation Loss	Accuracy
1	0.3949	0.3213	0.9282
2	0.2281	0.2196	0.9487
3	0.1966	0.1907	0.9547
4	0.1654	0.1810	0.9563
5	0.1581	0.1777	0.9572

Table 2: Arabic Training Result

Epoch	Training Loss	Validation Loss	Accuracy
1	0.1321	0.1613	0.9596
2	0.0857	0.1322	0.9668
3	0.0787	0.1312	0.9689
4	0.0393	0.1306	0.9699
5	0.0303	0.1338	0.9701

Table 3: English Training Result

Epoch	Training Loss	Validation Loss	Accuracy
1	0.2395	0.2045	0.9506
2	0.1532	0.1398	0.9644
3	0.1327	0.1238	0.9696
4	0.0870	0.1185	0.9716
5	0.0870	0.1168	0.9716

Table 4: Farsi Training Result

In table7, each row represents a specific language on which a model has been trained. Each column header indicates a language for the test dataset. The intersection of a row and a column contains the accuracy score of the model trained on the language corresponding to the row when evaluated on the test dataset of the language corresponding to the column. For example; a model trained on Farsi train dataset achieved 0.7295 accuracy on an Arabic test set.

Epoch	Training Loss	Validation Loss	Accuracy
1	0.2015	0.1486	0.9653
2	0.0834	0.0856	0.9786
3	0.0550	0.0711	0.9827
4	0.0429	0.0633	0.9849
5	0.0422	0.0627	0.9849

Table 5: French Training Result

Epoch	Training Loss	Validation Loss	Accuracy
1	0.1185	0.0979	0.9707
2	0.0828	0.0744	0.9791
3	0.0493	0.0697	0.9803
4	0.0412	0.0661	0.9818
5	0.0357	0.0702	0.9813

Table 6: Japanese Training Result

	Arabic	English	Farsi	French	Japanese
Arabic	0.9544	0.5907	0.7322	0.6269	0.4167
English	0.5486	0.9723	0.7197	0.8708	0.5057
Farsi	0.7295	0.7684	0.9710	0.7535	0.4956
French	0.6484	0.7683	0.7952	0.9859	0.5058
Japanese	0.5950	0.5541	0.5892	0.6061	0.9792

Table 7: Accuracy for Different Language Pairs

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

While we can use the same hyperparameters for training mBERT models on different languages(I did so for this Lab), it might not be optimal due to variations in linguistic structures, dataset sizes, and language complexity.

18. What can you conclude?

From Table 7, we observe that the diagonal values (e.g., Arabic-Arabic, English-English) are the highest in each row, indicating that the model performs best when tested on data from the same language it was trained on. This is expected since the training data directly matches the test data in terms of linguistic features and structure. The off-diagonal values show the performance of models trained on one language but tested on another. Generally, these scores are lower, suggesting that the model struggles to generalize across languages with different linguistic structures. For instance: The model trained on **Arabic** performs relatively well on **Farsi** (0.7322) but much worse on **English** (0.5907) and **Japanese** (0.4167). This may be due to the similarities between **Arabic** and **Farsi** as they share some syntactic and morphological features. The English-trained model has the best cross-linguistic performance when tested on **French** (0.8708), which could be attributed to shared alphabetic scripts and certain linguistic similarities between English and French. The Japanese-trained model shows the lowest cross-linguistic scores, likely due to the significant difference in grammar, syntax, and script compared to the other languages tested.