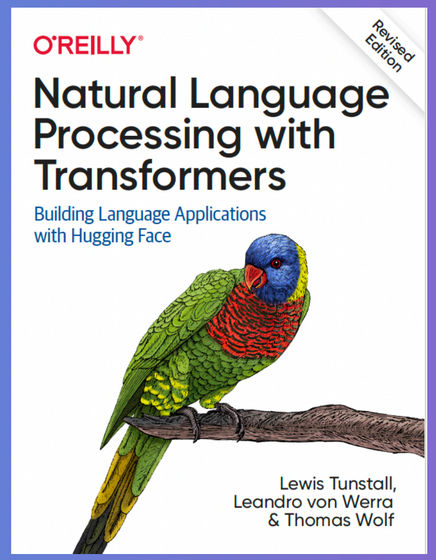
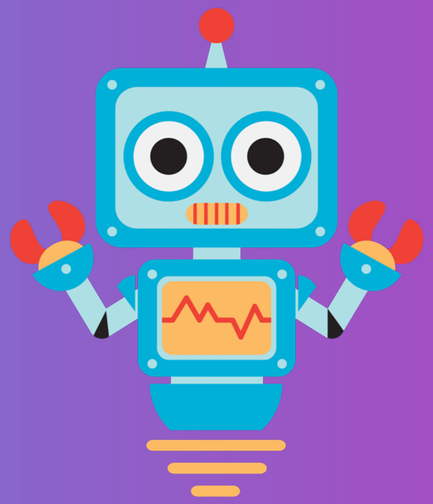
Dr. Abulkarim Albanna

End-to-End Multilingual Named Entity Recognition

Day 7

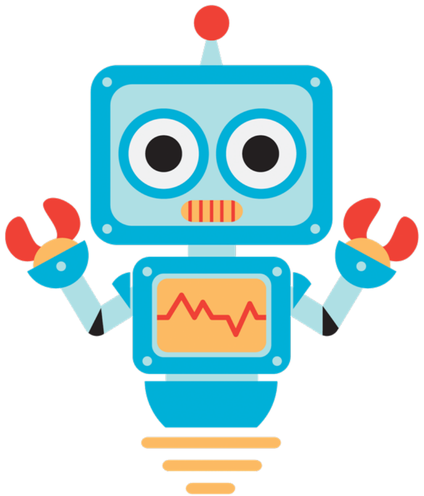
Feb 4 2024





Content

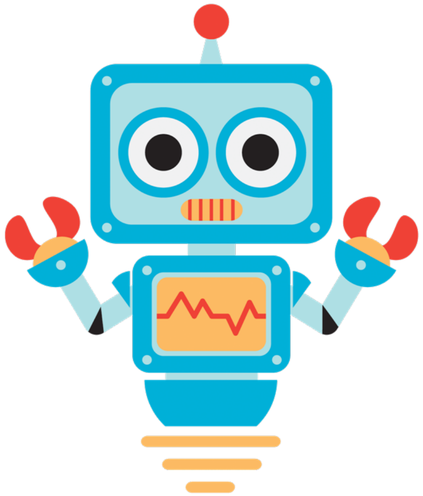
- Introduction
- The dataset
- Multilingual Transformers
- The Tokenizer Pipeline
- Transformers for Named Entity Recognition
- The Anatomy of the Transformers Model Class



Introduction

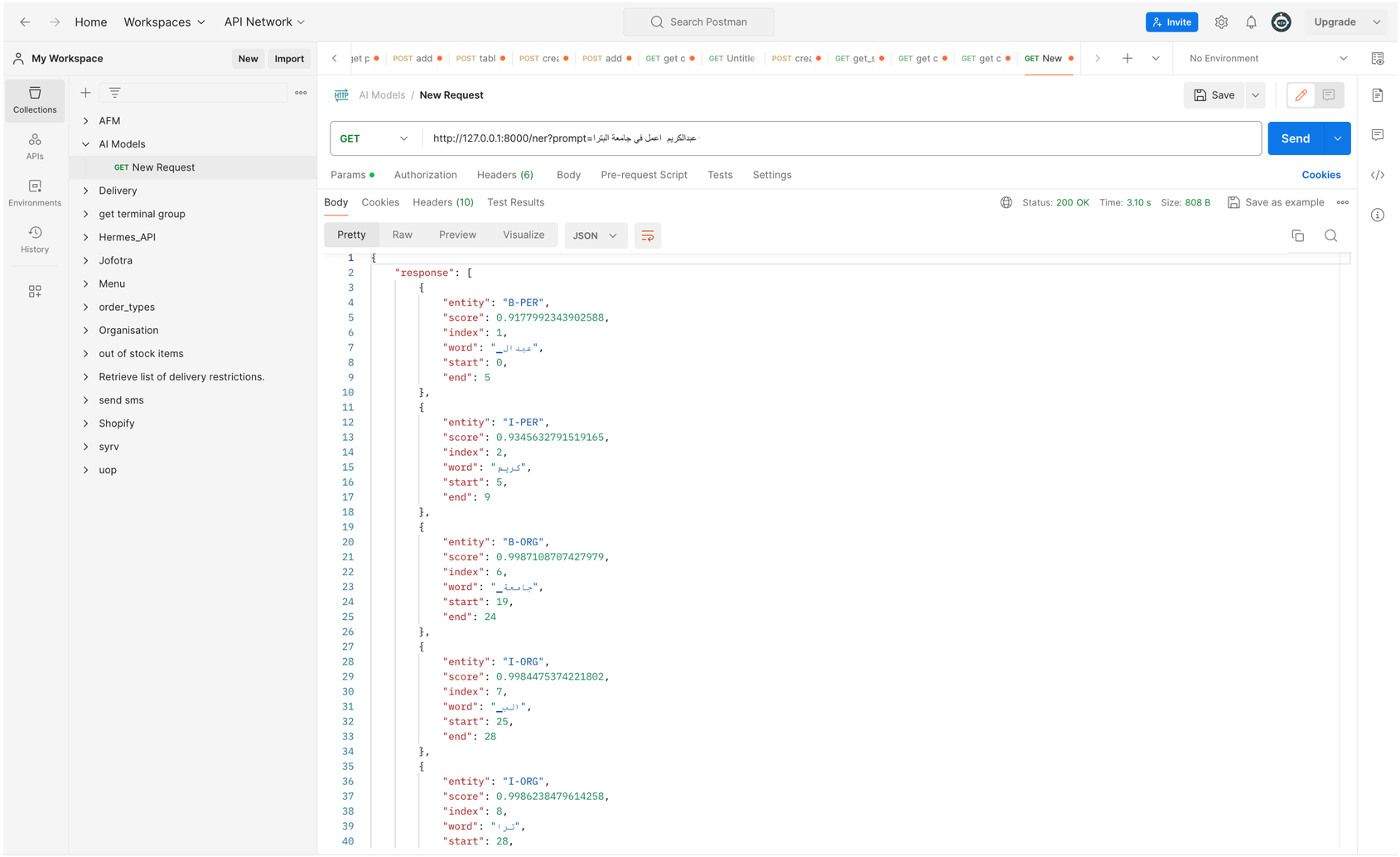
Named Entity Recognition

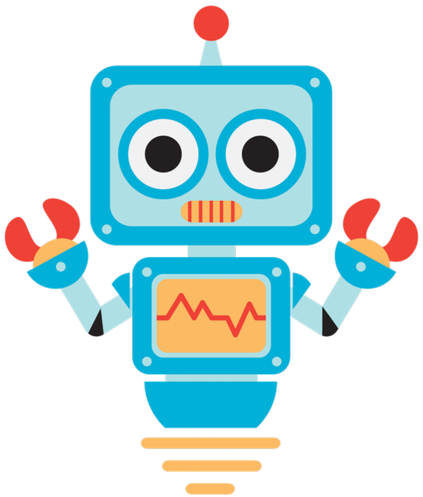
NER is a common NLP task that identifies entities like people, organizations, or locations in text. These entities can be used for various applications such as gaining insights from company documents, augmenting the quality of search engines, or simply building a structured database from a corpus.



Introduction

Named Entity Recognition





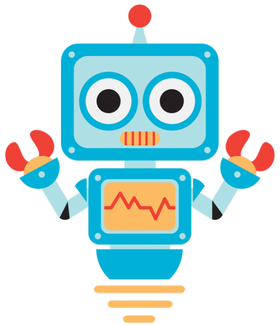
The Dataset

Named Entity Recognition

- We will be using a subset of the Cross-lingual TRansfer Evaluation of Multilingual Encoders (**XTREME**) benchmark called **WikiANN** or **PAN-X.2**

- In addition to **Arabic** the dataset consists of Wikipedia articles in many languages, including the four most commonly spoken languages.

- Each article is annotated with **LOC** (location), **PER** (person), and **ORG** (organization) tags in the “inside-outside-beginning” (**IOB2**) format. In this format, a **B- prefix** indicates the beginning of an entity, and consecutive tokens belonging to the same entity are given an **I- prefix**. An **O** tag means that the token does not belong to any entity.

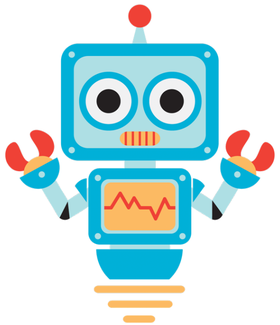


The Dataset

Named Entity Recognition

- Each article is annotated with **LOC** (location), **PER** (person), and **ORG** (organization) tags in the “inside-outside-beginning” (**IOB2**) format. In this format, a **B- prefix** indicates the beginning of an entity, and consecutive tokens belonging to the same entity are given an **I- prefix**. An **O** tag means that the token does not belong to any entity.

Tokens	Jeff	Dean	is	a	computer	scientist	at	Google	in	California
Tags	B-PER	I-PER	O	O	O	O	O	B-ORG	O	B-LOC



The Dataset

Named Entity Recognition

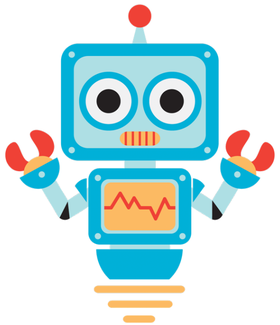
```
from datasets import get_dataset_config_names

xtreme_subsets = get_dataset_config_names("xtreme")
print(f"XTREME has {len(xtreme_subsets)} configurations")
```

```
XTREME has 183 configurations
```

```
panx_subsets = [s for s in xtreme_subsets if s.startswith("PAN")]
panx_subsets[:3]
```

```
['PAN-X.af', 'PAN-X.ar', 'PAN-X.bg']
```

```
# hide_output
from datasets import load_dataset

load_dataset("xtreme", name="PAN-X.ar")
```



The Dataset

Named Entity Recognition

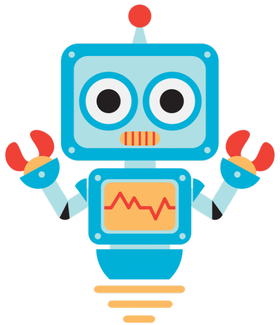
- To keep track of each language, let's create a Python defaultdict that stores the language code as the key and a PAN-X corpus of type DatasetDict as the value:

```
[9]  1 # hide_output
     2 from collections import defaultdict
     3 from datasets import DatasetDict
     4
     5 langs = ["ar", "fr", "it", "en"]
     6 fracs = [0.60, 0.10, 0.10, 0.20]
     7 # Return a DatasetDict if a key doesn't exist
     8 panx_ch = defaultdict(DatasetDict)
     9
    10 for lang, frac in zip(langs, fracs):
    11     # Load monolingual corpus
    12     ds = load_dataset("xtreme", name=f"PAN-X.{lang}")
    13     # Shuffle and downsample each split according to spoken proportion
    14     for split in ds:
    15         panx_ch[lang][split] = (
    16             ds[split]
    17             .shuffle(seed=0)
    18             .select(range(int(frac * ds[split].num_rows))))
```

```
[10] 1 import pandas as pd
     2
     3 pd.DataFrame({lang: [panx_ch[lang]["train"].num_rows] for lang in langs},
     4              index=["Number of training examples"])
```

	ar	fr	it	en
Number of training examples	12000	2000	2000	4000

The **`shuffle()`** solve bias in dataset splits. **`select()`** allows us to downsample each corpus according to the values in `fracs`.

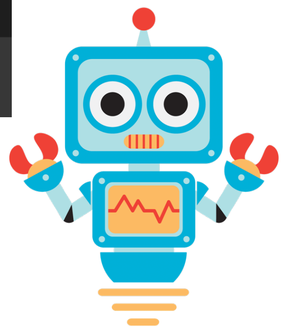


The Dataset

Named Entity Recognition

- To keep track of each language, let's create a Python defaultdict that stores the language code as the key and a PAN-X corpus of type DatasetDict as the value:

```
0s
  1 for key, value in panx_ch["ar"]["train"].features.items():
  2     print(f"{key}: {value}")

tokens: Sequence(feature=Value(dtype='string', id=None), length=-1, id=None)
ner_tags: Sequence(feature=ClassLabel(names=['O', 'B-PER', 'I-PER', 'B-ORG', 'I-ORG', 'B-LOC', 'I-LOC'], id=None), length=-1, id=None)
langs: Sequence(feature=Value(dtype='string', id=None), length=-1, id=None)

0s
[17]  1 tags = panx_ch["ar"]["train"].features["ner_tags"].feature
      2 print(tags)

ClassLabel(names=['O', 'B-PER', 'I-PER', 'B-ORG', 'I-ORG', 'B-LOC', 'I-LOC'], id=None)
```



The Dataset

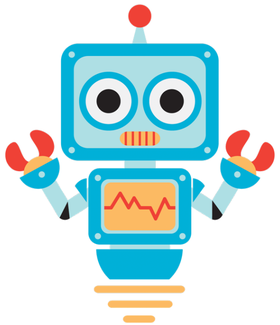
Named Entity Recognition

Apply String labelling

```
[19]  1 # hide_output
      2 def create_tag_names(batch):
      3     return {"ner_tags_str": [tags.int2str(idx) for idx in batch["ner_tags"]]}
      4
      5 panx_ar = panx_ch["ar"].map(create_tag_names)
```

```
1 # hide_output
2 ar_example = panx_ar["train"][0]
3 pd.DataFrame([ar_example["tokens"], ar_example["ner_tags_str"]],
4 ['Tokens', 'Tags'])
```

	0	1	2	3
Tokens	تحويل	أحمد	سعيد	الكاظمي
Tags	O	B-PER	I-PER	I-PER



The Dataset

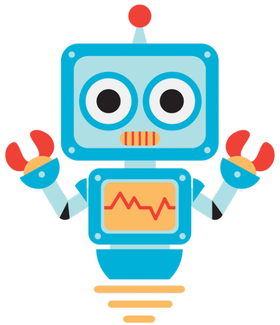
Named Entity Recognition

To calculate the frequencies of each entity across each split:

```
0s
```

```python
from collections import Counter

split2freqs = defaultdict(Counter)
for split, dataset in panx_ar.items():
    for row in dataset["ner_tags_str"]:
        for tag in row:
            if tag.startswith("B"):
                tag_type = tag.split("-")[1]
                split2freqs[split][tag_type] += 1
pd.DataFrame.from_dict(split2freqs, orient="index")
```

	PER	ORG	LOC
train	4583	4296	4594
validation	2290	2136	2318
test	2233	2197	2321



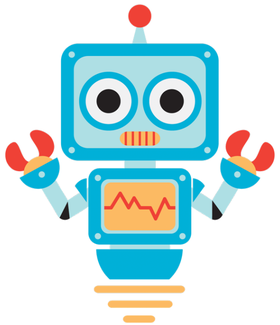
Multilingual Transformers

Named Entity Recognition

- Multilingual transformers involve similar architectures and training procedures as their monolingual counterparts, except that the corpus used for pretraining consists of documents in many languages.

- A remarkable feature of this approach is that despite receiving no explicit information to differentiate among the languages, the resulting linguistic representations are able to generalize well across languages for a variety of downstream tasks.

- In some cases, this ability to perform cross-lingual transfer can produce results that are competitive with those of monolingual models, which circumvents the need to train one model per language!

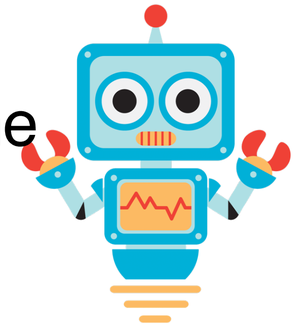


Multilingual Transformers

Named Entity Recognition

- XLM-R uses only MLM as a pretraining objective for 100 languages.Wikipedia dumps for each language and 2.5 terabytes of Common Crawl data from the web. This corpus is several orders of magnitude larger than the ones used in earlier models and provides a significant boost in signal for low-resource languages like Burmese and Swahili, where only a small number of Wikipedia articles exist.

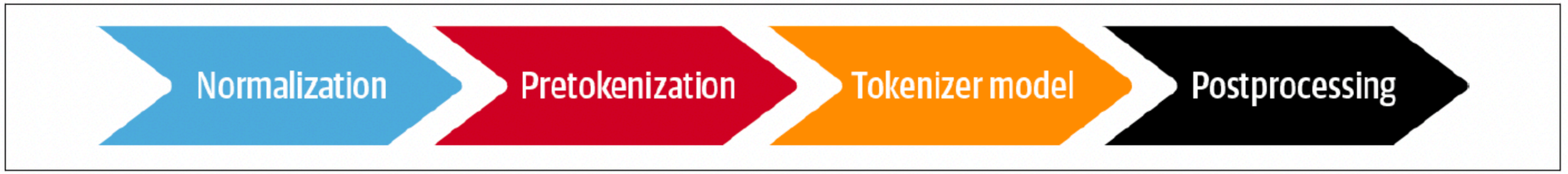
- XLM-R also drops the language embeddings used in XLM and uses SentencePiece to tokenize the raw texts directly.Besides its multilingual nature, a notable difference between XLM-R and RoBERTa is the size of the respective vocabularies: 250,000 tokens versus 55,000!.

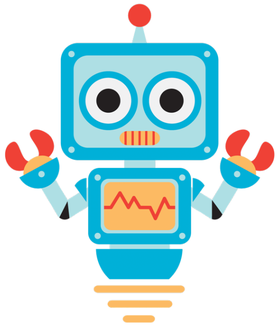


- XLM-R is a great choice for multilingual NLU tasks.

The Tokenizer Pipeline

Named Entity Recognition





Normalization

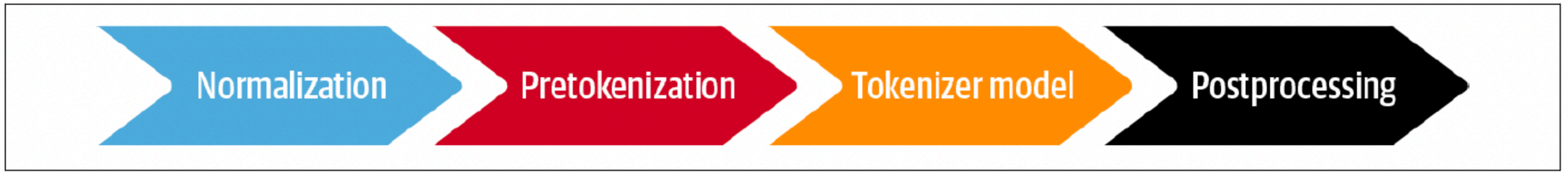
- **Common operations** include stripping whitespace and removing accented characters.

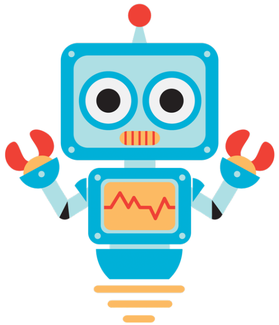
- **Unicode normalization** is another common normalization operation applied by many tokenizers to deal with the fact that there often exist various ways to write the same character. schemes like NFC, NFD, NFKC, and NFKD replace the various ways to write the same character with standard forms.

- **Lowercasing**. If the model is expected to only accept and use lowercase characters, this technique can be used to reduce the size of the vocabulary it requires.

The Tokenizer Pipeline

Named Entity Recognition



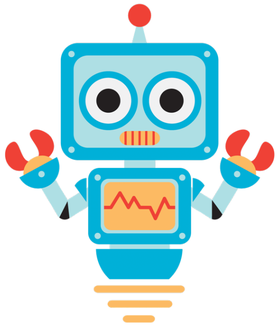


Pretokenization

- This step splits a text into smaller objects that give an upper bound to what your tokens will be at the end of training. A good way to think of this is that the pretokenizer will split your text into “words,” and your final tokens will be parts of those words. For the languages that allow this (English, German, and many Indo- uropean languages), strings can typically be split into words on whitespace and punctuation. **Byte-Pair Encoding (BPE) or Unigram algorithms** can be used.

The Tokenizer Pipeline

Named Entity Recognition



Tokenizer model

- The role of the model is to split the words into subwords to reduce the size of the vocabulary and try to reduce the number of vocabulary tokens. Several subword tokenization algorithms exist, including WordPiece,SentencePiece

The Tokenizer Pipeline

Named Entity Recognition

```
[25]  1 # hide_output
      2 from transformers import AutoTokenizer
      3
      4 bert_model_name = "bert-base-cased"
      5 xlmr_model_name = "xlm-roberta-base"
      6 bert_tokenizer = AutoTokenizer.from_pretrained(bert_model_name)
      7 xlmr_tokenizer = AutoTokenizer.from_pretrained(xlmr_model_name)
```

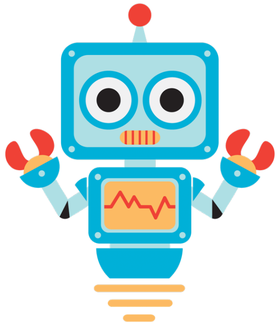
```
tokenizer_config.json: 100%  29.0/29.0 [00:00<00:00, 370B/s]
config.json: 100%  570/570 [00:00<00:00, 6.51kB/s]
vocab.txt: 100%  213k/213k [00:00<00:00, 3.16MB/s]
tokenizer.json: 100%  436k/436k [00:00<00:00, 4.11MB/s]
config.json: 100%  615/615 [00:00<00:00, 10.2kB/s]
sentencepiece.bpe.model: 100%  5.07M/5.07M [00:00<00:00, 26.5MB/s]
tokenizer.json: 100%  9.10M/9.10M [00:00<00:00, 39.6MB/s]
```

```
1 text="My name is Kareem from Jordan"
2 bert_tokens = bert_tokenizer(text).tokens()
3 xlmr_tokens = xlmr_tokenizer(text).tokens()
4
```

+ Code

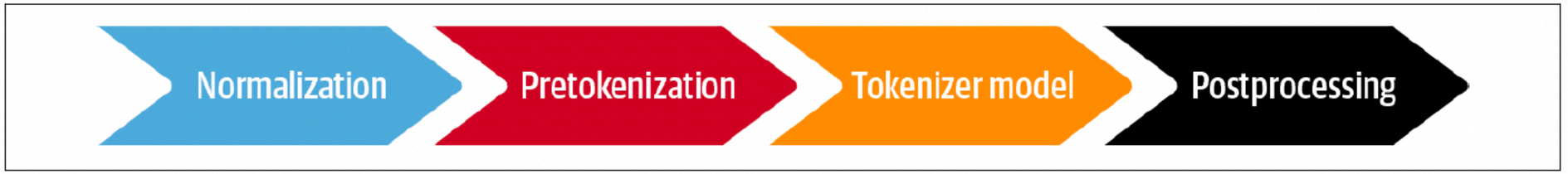
```
1 #hide_input
2 df = pd.DataFrame([bert_tokens, xlmr_tokens], index=["BERT", "XLM-R"])
3 df
```

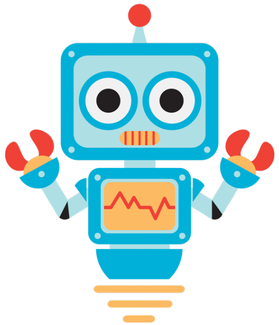
	0	1	2	3	4	5	6	7	8	9
BERT	[CLS]	My	name	is	Ka	##ree	##m	from	Jordan	[SEP]
XLM-R	<s>	_My	_name	_is	_Kare	em	_from	_Jordan	</s>	None



The Tokenizer Pipeline

Named Entity Recognition





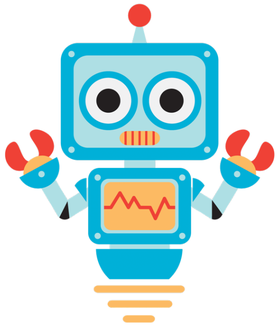
Postprocessing

- This is the last step of the tokenization pipeline, in which some additional transformations can be applied on the list of tokens—for instance, adding special tokens at the beginning or end of the input sequence of token indices.

- SentencePiece adds `<s>` and `</s>` instead of `[CLS]` and `[SEP]` in the postprocessing step (as a convention, we'll continue to use `[CLS]` and `[SEP]` in the graphical illustrations)

The Tokenizer Pipeline

Named Entity Recognition

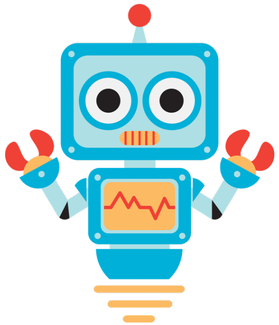


The SentencePiece Tokenizer

- The SentencePiece tokenizer is based on a type of subword segmentation called Unigram and encodes each input text as a sequence of **Unicode characters**. This last feature is especially useful for multilingual corpora since it allows SentencePiece to be agnostic about accents, punctuation, and the fact that many languages, like Japanese, do not have whitespace characters. Another special feature of SentencePiece is that whitespace is assigned the Unicode symbol U+2581, or the ▁ character, also called the l**ower one quarter block character**.

The Tokenizer Pipeline

Named Entity Recognition



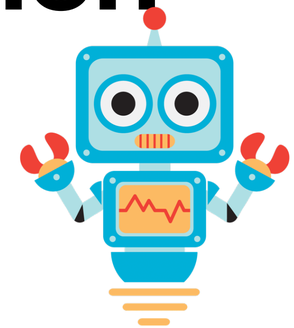
The SentencePiece Tokenizer

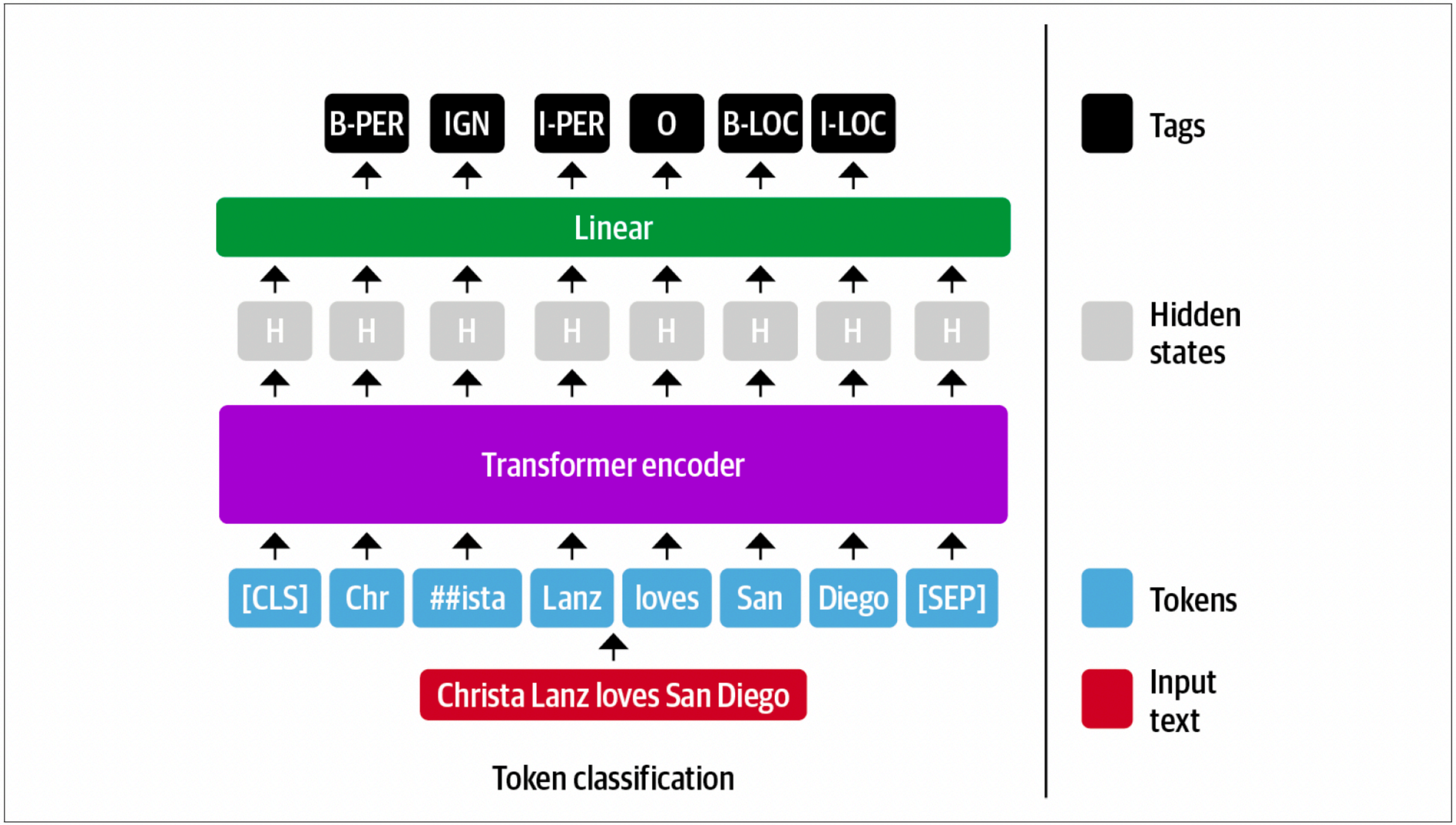
- This enables SentencePiece to detokenize a sequence without ambiguities and without relying on language-specific pretokenizers.

```
1 "".join(xlmr_tokens).replace(u"\u2581", " ")

'<s> My name is Kareem from Jordan</s>'
```

Transformers for Named Entity Recognition

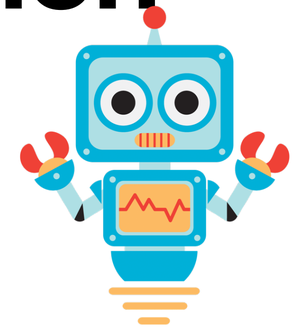
Named Entity Recognition





Transformers for Named Entity Recognition

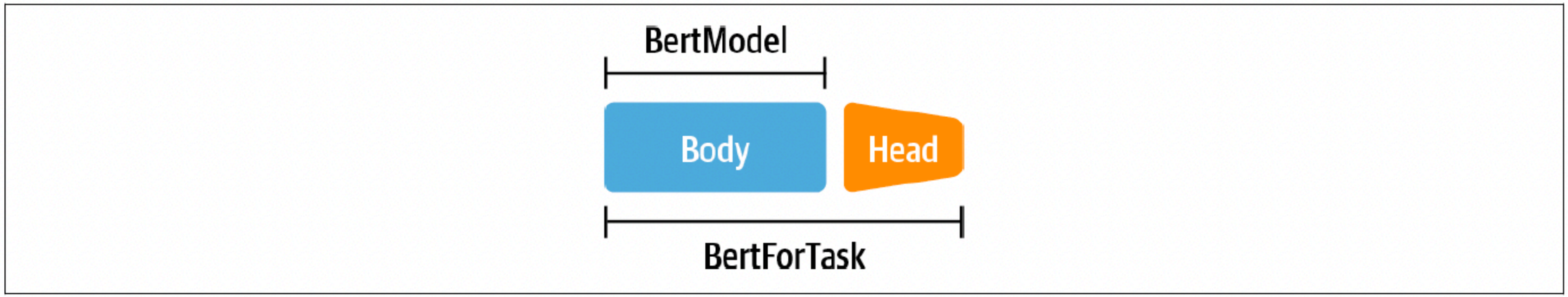
Named Entity Recognition

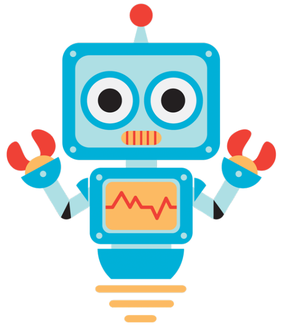


BERT and other encoder-only transformers take a similar approach for NER, except that the representation of each individual input token is fed into the same fully connected layer to output the entity of the token. For this reason, NER is often framed as a **token classification task.** but how should we handle subwords in a token classification task? For example, the first name “Christa” in previous Figure is tokenized into the subwords “Chr” and “##ista”, so which one(s) should be assigned the **`B-PER`** label?

The Anatomy of the Transformers Model Class

Named Entity Recognition





The Anatomy of the Transformers Model Class

Creating a Custom Model for Token Classification

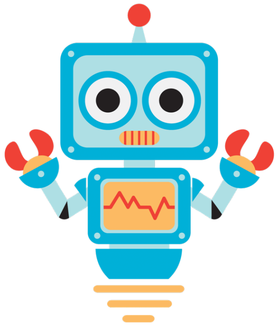
```python
import torch.nn as nn
from transformers import XLMRobertaConfig
from transformers.modeling_outputs import TokenClassifierOutput
from transformers.models.roberta.modeling_roberta import RobertaModel
from transformers.models.roberta.modeling_roberta import RobertaPreTrainedModel

class XLMRobertaForTokenClassification(RobertaPreTrainedModel):
    config_class = XLMRobertaConfig

    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels
        # Load model body
        self.roberta = RobertaModel(config, add_pooling_layer=False)
        # Set up token classification head
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)
        # Load and initialize weights
        self.init_weights()
```

The **`config_class`** ensures that the standard XLM-R settings are used .

The **`super()`** method we call the initialization function of the `RobertaPreTrainedModel` class. This abstract class handles the initialization or loading of pretrained weights. Then we load our model body, which is **`RobertaModel`**.

Extend model with our own classification head consisting of a dropout and a standard feed-forward layer. Note that we set **`add_pooling_layer`**`=False` to ensure all hidden states are returned and not only the one associated with the `[CLS]` token. Finally,

Initialize all the weights by calling the **`init_weights()`** method we inherit from `RobertaPreTrainedModel`, which will load the pretrained weights for the



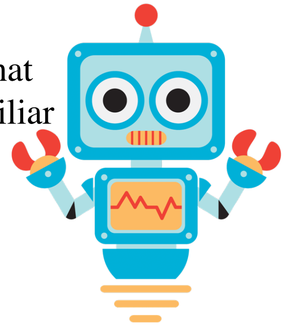
The Anatomy of the Transformers Model Class

Creating a Custom Model for Token Classification

```
21    def forward(self, input_ids=None, attention_mask=None, token_type_ids=None,
22                labels=None, **kwargs):
23        # Use model body to get encoder representations
24        outputs = self.roberta(input_ids, attention_mask=attention_mask,
25                               token_type_ids=token_type_ids, **kwargs)
26        # Apply classifier to encoder representation
27        sequence_output = self.dropout(outputs[0])
28        logits = self.classifier(sequence_output)
29        # Calculate losses
30        loss = None
31        if labels is not None:
32            loss_fct = nn.CrossEntropyLoss()
33            loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
34        # Return model output object
35        return TokenClassifierOutput(loss=loss, logits=logits,
36                                     hidden_states=outputs.hidden_states,
37                                     attentions=outputs.attentions)
```

- The only thing left to do is to define what the model should do in a forward pass with a **`forward()`** method. During the forward pass, the data is first fed through the model body. There are a number of input variables, but the only ones we need for now are **`input_ids`** and **`attention_mask`**.

- The **hidden state**, which is part of the model body output, is then fed through the dropout and classification layers. If we also provide labels in the forward pass, we can directly calculate the loss. If there is an attention mask we need to do a little bit more work to make sure we only calculate the loss of the unmasked tokens.
- Finally, we wrap all the outputs in a **`TokenClassifierOutput`** object that allows us to access elements in a the familiar named tuple.



The Anatomy of the Transformers Model Class

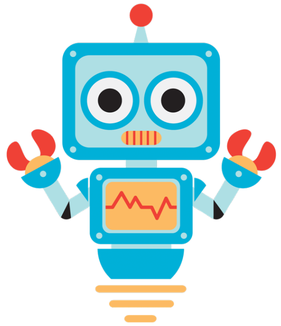
Loading a Custom Model

```
index2tag = {idx: tag for idx, tag in enumerate(tags.names)}
tag2index = {tag: idx for idx, tag in enumerate(tags.names)}
```

```
# hide_output
from transformers import AutoConfig

xlmr_config = AutoConfig.from_pretrained(xlmr_model_name,
                                         num_labels=tags.num_classes,
                                         id2label=index2tag, label2id=tag2index)
```

Now we are ready to load our token classification model. We'll need to provide some additional information beyond the model name, including the tags that we will use to label each entity and the mapping of each tag to an ID and vice versa. All of this information can be derived from our `tags` variable, which as a `ClassLabel` object has a `names` attribute that we can use to derive the mapping.

The **`AutoConfig`** class contains the blueprint of a model's architecture. When we load a model with `AutoModel.from_pretrained(model_ckpt)`, the configuration file associated with that model is downloaded automatically. However, if we want to modify something like the number of classes or label names, then we can load the configuration first with the parameters we would like to customize. Now, we can load the model weights as usual with the `from_pretrained()` function with the additional `config` argument.



The Anatomy of the Transformers Model Class

Loading a Custom Model

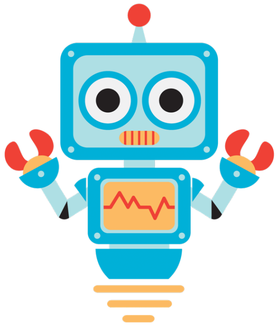
```
[37]  1 # hide_output
      2 import torch
      3
      4 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
      5 xlmr_model = (XLMRobertaForTokenClassification
      6               .from_pretrained(xlmr_model_name, config=xlmr_config)
      7               .to(device))
```

```
model.safetensors: 100% ████████████████████ 1.12G/1.12G [00:13<00:00, 119MB/s]
Some weights of XLMRobertaForTokenClassification were not initialized from the model c
You should probably TRAIN this model on a down-stream task to be able to use it for pr
```

```
1 # hide_output
2 input_ids = xlmr_tokenizer.encode(text, return_tensors="pt")
3 pd.DataFrame([xlmr_tokens, input_ids[0].numpy()], index=["Tokens", "Input IDs"])
```

	0	1	2	3	4	5	6	7	8
Tokens	<s>	_My	_name	_is	_Kare	em	_from	_Jordan	</s>
Input IDs	0	2646	9351	83	84540	195	1295	44532	2

As a quick check that we have initialized the tokenizer and model correctly, let's test the predictions on our small sequence of known entities:



The Anatomy of the Transformers Model Class

Tokenizing Texts for NER

```
[43]  1 words, labels = ar_example["tokens"], ar_example["ner_tags"]
```

```
[44]  1 tokenized_input = xlmr_tokenizer(ar_example["tokens"], is_split_into_words=True)
      2 tokens = xlmr_tokenizer.convert_ids_to_tokens(tokenized_input["input_ids"])
```

```
[45]  1 #hide_output
      2 pd.DataFrame([tokens], index=["Tokens"])
```

	0	1	2	3	4	5	6	7	8
Tokens	<s>	__	تحويل	__أحمد	__سعيد	__الكا	ظ	مي	</s>

```
[46]  1 # hide_output
      2 word_ids = tokenized_input.word_ids()
      3 pd.DataFrame([tokens, word_ids], index=["Tokens", "Word IDs"])
```

	0	1	2	3	4	5	6	7	8
Tokens	<s>	__	تحويل	__أحمد	__سعيد	__الكا	ظ	مي	</s>
Word IDs	None	0	0	1	2	3	3	3	None

Since the XLM-R tokenizer returns the input IDs for the model's inputs, we just need to augment this information with the attention mask and the label IDs that encode the information about which token is associated with each NER tag.

`is_split_into_words` argument to tell the tokenizer that our input sequence has already been split into words:

The Anatomy of the Transformers Model Class

Tokenizing Texts for NER

```
#hide_output
previous_word_idx = None
label_ids = []

for word_idx in word_ids:
    if word_idx is None or word_idx == previous_word_idx:
        label_ids.append(-100)
    elif word_idx != previous_word_idx:
        label_ids.append(labels[word_idx])
    previous_word_idx = word_idx

labels = [index2tag[l] if l != -100 else "IGN" for l in label_ids]
index = ["Tokens", "Word IDs", "Label IDs", "Labels"]

pd.DataFrame([tokens, word_ids, label_ids, labels], index=index)
```

	0	1	2	3	4	5	6	7	8
Tokens	<s>	_	تحويل	أحمد_	سعيد_	الكا_	ظ	مي	</s>
Word IDs	None	0	0	1	2	3	3	3	None
Label IDs	-100	0	-100	1	2	2	-100	-100	-100
Labels	IGN	O	IGN	B-PER	I-PER	I-PER	IGN	IGN	IGN

Here we can see that **`word_ids`** has mapped each subword to the corresponding index in the `words` sequence,

Why did we choose –100 as the ID to mask subword representations?

The reason is that in PyTorch the cross-entropy loss class **torch.nn.CrossEntropyLoss** has an attribute called **ignore_index** whose value is –100. This index is ignored during training, so we can use it to ignore the tokens associated with consecutive subwords.

The Anatomy of the Transformers Model Class

Tokenizing Texts for NER

```
[53]  1 def tokenize_and_align_labels(examples):
      2     tokenized_inputs = xlmr_tokenizer(examples["tokens"], truncation=True,
      3                                       is_split_into_words=True)
      4     labels = []
      5     for idx, label in enumerate(examples["ner_tags"]):
      6         word_ids = tokenized_inputs.word_ids(batch_index=idx)
      7         previous_word_idx = None
      8         label_ids = []
      9         for word_idx in word_ids:
     10             if word_idx is None or word_idx == previous_word_idx:
     11                 label_ids.append(-100)
     12             else:
     13                 label_ids.append(label[word_idx])
     14             previous_word_idx = word_idx
     15         labels.append(label_ids)
     16     tokenized_inputs["labels"] = labels
     17     return tokenized_inputs
```

```
[54]  1 def encode_panx_dataset(corpus):
      2     return corpus.map(tokenize_and_align_labels, batched=True,
      3                       remove_columns=['langs', 'ner_tags', 'tokens'])
```

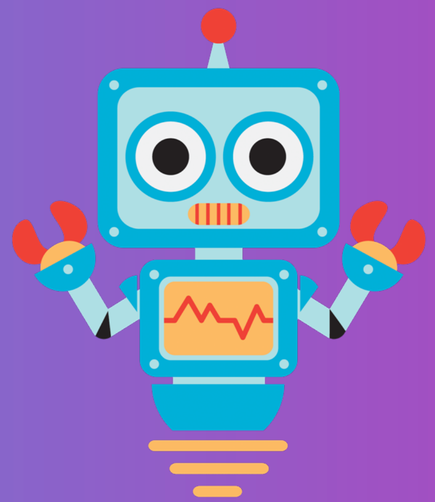
```
[56]  1 # hide_output
      2 panx_ar_encoded = encode_panx_dataset(panx_ch["ar"])
```

```
      1 panx_ar_encoded
```

```
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 12000
    })
    validation: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 6000
    })
    test: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 6000
    })
})
```

Scale this out to the whole dataset by defining a single function that wraps all the logic.

Thank you for your attention



Day 7

Feb 4 2024