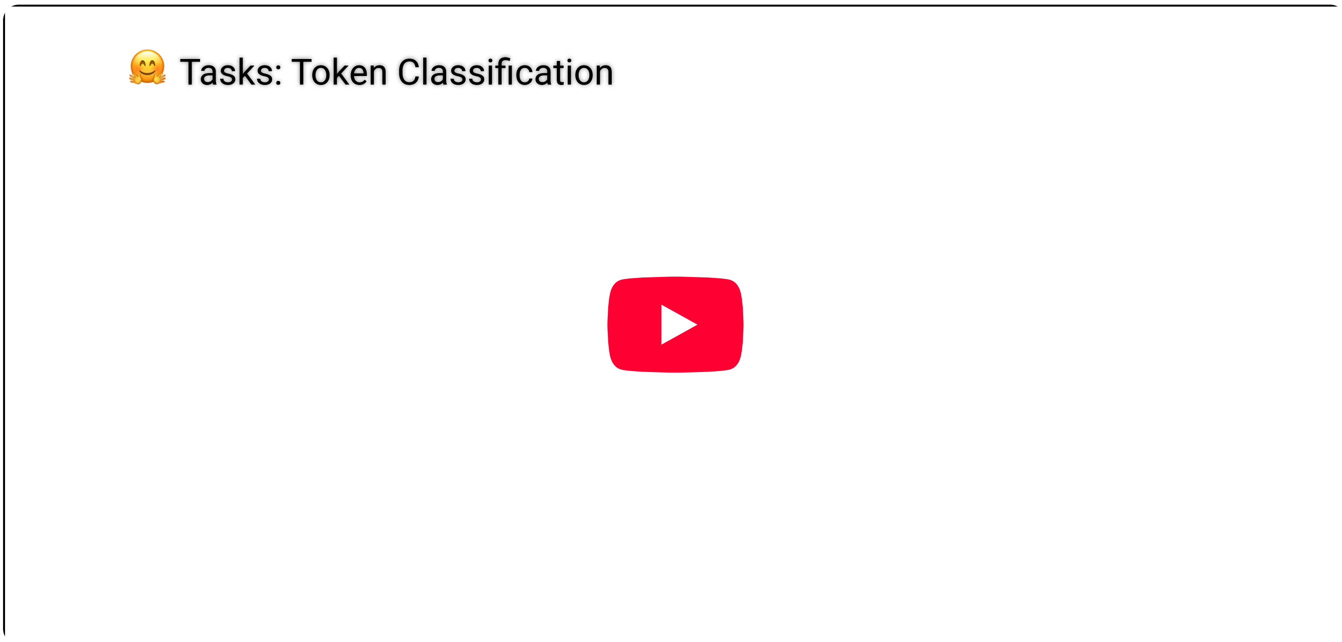
Token classification

Open in Colab Open Studio Lab Copy page



Token classification assigns a label to individual tokens in a sentence. One of the most common token classification tasks is Named Entity Recognition (NER). NER attempts to find a label for each entity in a sentence, such as a person, location, or organization.

This guide will show you how to:

1. Finetune DistilBERT on the WNUT 17 dataset to detect new entities.
2. Use your finetuned model for inference.

> To see all architectures and checkpoints compatible with this task, we recommend checking the task-page.

Before you begin, make sure you have all the necessary libraries installed:

```
pip install transformers datasets evaluate seqeval
```

We encourage you to login to your Hugging Face account so you can upload and share your model with the community. When prompted, enter your token to login:

```
>>> from huggingface_hub import notebook_login

>>> notebook_login()
```

Load WNUT 17 dataset

Start by loading the WNUT 17 dataset from the 🤗 Datasets library:

```
>>> from datasets import load_dataset

>>> wnut = load_dataset("wnut_17")
```

Then take a look at an example:

```
>>> wnut["train"][0]
{'id': '0',
 'ner_tags': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 8, 8, 0, 7, 0, 0, 0, 0, 0,
 'tokens': ['@paulwalk', 'It', "'s", 'the', 'view', 'from', 'where', 'I', "'m", 'livin
}
```

Each number in `ner_tags` represents an entity. Convert the numbers to their label names to find out what the entities are:

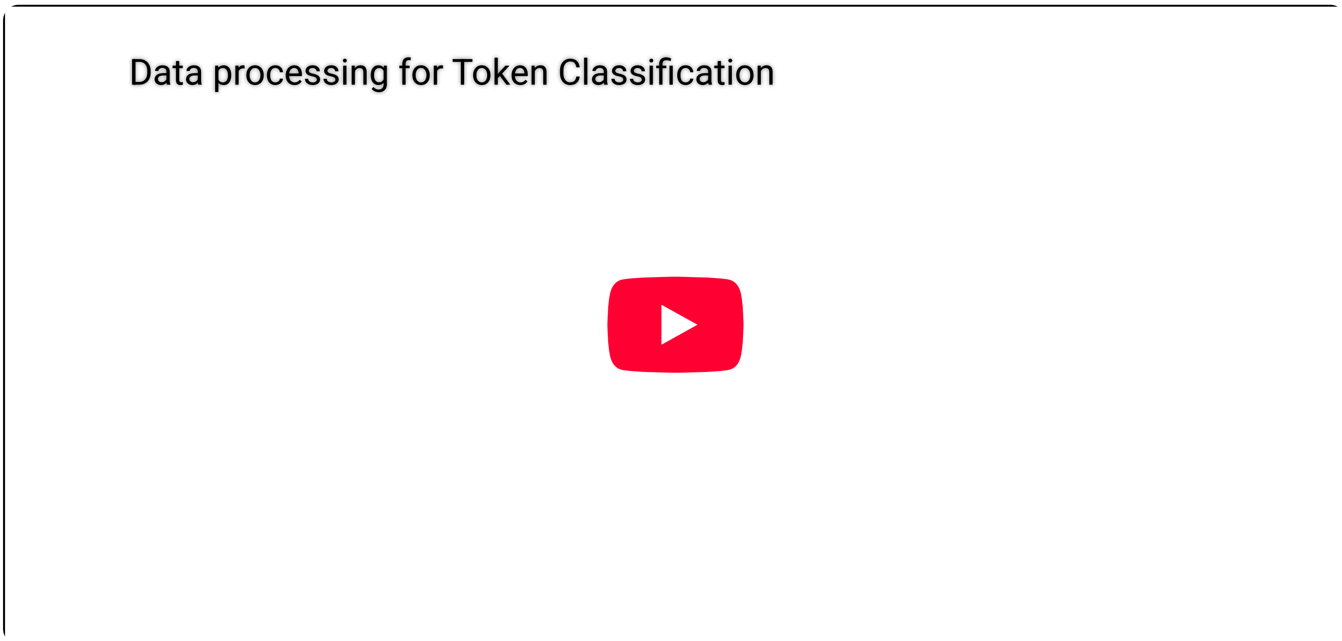
```
>>> label_list = wnut["train"].features[f"ner_tags"].feature.names
>>> label_list
[
    "O",
    "B-corporation",
    "I-corporation",
    "B-creative-work",
    "I-creative-work",
    "B-group",
    "I-group",
```

```
    "B-location",
    "I-location",
    "B-person",
    "I-person",
    "B-product",
    "I-product",
]
```

The letter that prefixes each `ner_tag` indicates the token position of the entity:

- `B-` indicates the beginning of an entity.
- `I-` indicates a token is contained inside the same entity (for example, the `State` token is a part of an entity like `Empire State Building`).
- `0` indicates the token doesn't correspond to any entity.

Preprocess



The next step is to load a DistilBERT tokenizer to preprocess the `tokens` field:

```
>>> from transformers import AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("distilbert/distilbert-base-uncased")
```

As you saw in the example `tokens` field above, it looks like the input has already been tokenized. But the input actually hasn't been tokenized yet and you'll need to set `is_split_into_words=True` to tokenize the words into subwords. For example:

```
>>> example = wnut["train"][0]
>>> tokenized_input = tokenizer(example["tokens"], is_split_into_words=True)
>>> tokens = tokenizer.convert_ids_to_tokens(tokenized_input["input_ids"])
>>> tokens
['[CLS]', '@', 'paul', '##walk', 'it', "'", 's', 'the', 'view', 'from', 'where', 'i',
```

However, this adds some special tokens `[CLS]` and `[SEP]` and the subword tokenization creates a mismatch between the input and labels. A single word corresponding to a single label may now be split into two subwords. You'll need to realign the tokens and labels by:

1. Mapping all tokens to their corresponding word with the [word_ids](#) method.
2. Assigning the label `-100` to the special tokens `[CLS]` and `[SEP]` so they're ignored by the PyTorch loss function (see [CrossEntropyLoss](#)).
3. Only labeling the first token of a given word. Assign `-100` to other subtokens from the same word.

Here is how you can create a function to realign the tokens and labels, and truncate sequences to be no longer than DistilBERT's maximum input length:

```
>>> def tokenize_and_align_labels(examples):
...     tokenized_inputs = tokenizer(examples["tokens"], truncation=True, is_split_int

...     labels = []
...     for i, label in enumerate(examples[f"ner_tags"]):
...         word_ids = tokenized_inputs.word_ids(batch_index=i)  # Map tokens to their
...         previous_word_idx = None
...         label_ids = []
...         for word_idx in word_ids:  # Set the special tokens to -100.
...             if word_idx is None:
...                 label_ids.append(-100)
...             elif word_idx != previous_word_idx:  # Only label the first token of a
...                 label_ids.append(label[word_idx])
...             else:
...                 label_ids.append(-100)
...             previous_word_idx = word_idx
...         labels.append(label_ids)
```

```
...     tokenized_inputs["labels"] = labels
...     return tokenized_inputs
```

To apply the preprocessing function over the entire dataset, use 🤗 Datasets map function. You can speed up the `map` function by setting `batched=True` to process multiple elements of the dataset at once:

```
>>> tokenized_wnut = wnut.map(tokenize_and_align_labels, batched=True)
```

Now create a batch of examples using DataCollatorWithPadding. It's more efficient to *dynamically pad* the sentences to the longest length in a batch during collation, instead of padding the whole dataset to the maximum length.

```
>>> from transformers import DataCollatorForTokenClassification

>>> data_collator = DataCollatorForTokenClassification(tokenizer=tokenizer)
```

Evaluate

Including a metric during training is often helpful for evaluating your model's performance. You can quickly load a evaluation method with the 🤗 Evaluate library. For this task, load the seqeval framework (see the 🤗 Evaluate quick tour to learn more about how to load and compute a metric). Seqeval actually produces several scores: precision, recall, F1, and accuracy.

```
>>> import evaluate

>>> seqeval = evaluate.load("seqeval")
```

Get the NER labels first, and then create a function that passes your true predictions and true labels to compute to calculate the scores:

```
>>> import numpy as np

>>> labels = [label_list[i] for i in example[f"ner_tags"]]
```

```
>>> def compute_metrics(p):
...     predictions, labels = p
...     predictions = np.argmax(predictions, axis=2)

...     true_predictions = [
...         [label_list[p] for (p, l) in zip(prediction, label) if l != -100]
...         for prediction, label in zip(predictions, labels)
...     ]
...     true_labels = [
...         [label_list[l] for (p, l) in zip(prediction, label) if l != -100]
...         for prediction, label in zip(predictions, labels)
...     ]

...     results = seqeval.compute(predictions=true_predictions, references=true_labels
...     return {
...         "precision": results["overall_precision"],
...         "recall": results["overall_recall"],
...         "f1": results["overall_f1"],
...         "accuracy": results["overall_accuracy"],
...     }
```

Your `compute_metrics` function is ready to go now, and you'll return to it when you setup your training.

Train

Before you start training your model, create a map of the expected ids to their labels with `id2label` and `label2id`:

```
>>> id2label = {
...     0: "O",
...     1: "B-corporation",
...     2: "I-corporation",
...     3: "B-creative-work",
...     4: "I-creative-work",
...     5: "B-group",
...     6: "I-group",
...     7: "B-location",
...     8: "I-location",
...     9: "B-person",
...     10: "I-person",
...     11: "B-product",
...     12: "I-product",
```

```
... }
>>> label2id = {
...     "O": 0,
...     "B-corporation": 1,
...     "I-corporation": 2,
...     "B-creative-work": 3,
...     "I-creative-work": 4,
...     "B-group": 5,
...     "I-group": 6,
...     "B-location": 7,
...     "I-location": 8,
...     "B-person": 9,
...     "I-person": 10,
...     "B-product": 11,
...     "I-product": 12,
... }
```

> If you aren't familiar with finetuning a model with the Trainer, take a look at the basic tutorial here!

You're ready to start training your model now! Load DistilBERT with AutoModelForTokenClassification along with the number of expected labels, and the label mappings:

```
>>> from transformers import AutoModelForTokenClassification, TrainingArguments, Train

>>> model = AutoModelForTokenClassification.from_pretrained(
...     "distilbert/distilbert-base-uncased", num_labels=13, id2label=id2label, label2
... )
```

At this point, only three steps remain:

1. Define your training hyperparameters in TrainingArguments. The only required parameter is `output_dir` which specifies where to save your model. You'll push this model to the Hub by setting `push_to_hub=True` (you need to be signed in to Hugging Face to upload your model). At the end of each epoch, the Trainer will evaluate the seqeval scores and save the training checkpoint.
2. Pass the training arguments to Trainer along with the model, dataset, tokenizer, data collator, and `compute_metrics` function.

3. Call train() to finetune your model.

```
>>> training_args = TrainingArguments(
...     output_dir="my_awesome_wnut_model",
...     learning_rate=2e-5,
...     per_device_train_batch_size=16,
...     per_device_eval_batch_size=16,
...     num_train_epochs=2,
...     weight_decay=0.01,
...     eval_strategy="epoch",
...     save_strategy="epoch",
...     load_best_model_at_end=True,
...     push_to_hub=True,
... )

>>> trainer = Trainer(
...     model=model,
...     args=training_args,
...     train_dataset=tokenized_wnut["train"],
...     eval_dataset=tokenized_wnut["test"],
...     processing_class=tokenizer,
...     data_collator=data_collator,
...     compute_metrics=compute_metrics,
... )

>>> trainer.train()
```

Once training is completed, share your model to the Hub with the push_to_hub() method so everyone can use your model:

```
>>> trainer.push_to_hub()
```

> For a more in-depth example of how to finetune a model for token classification, take a look at the corresponding PyTorch notebook.

Inference

Great, now that you've finetuned a model, you can use it for inference!

Grab some text you'd like to run inference on:

```
>>> text = "The Golden State Warriors are an American professional basketball team bas
```

The simplest way to try out your finetuned model for inference is to use it in a pipeline(). Instantiate a `pipeline` for NER with your model, and pass your text to it:

```
>>> from transformers import pipeline

>>> classifier = pipeline("ner", model="stevhliu/my_awesome_wnut_model")
>>> classifier(text)
[{'entity': 'B-location',
  'score': 0.42658573,
  'index': 2,
  'word': 'golden',
  'start': 4,
  'end': 10},
 {'entity': 'I-location',
  'score': 0.35856336,
  'index': 3,
  'word': 'state',
  'start': 11,
  'end': 16},
 {'entity': 'B-group',
  'score': 0.3064001,
  'index': 4,
  'word': 'warriors',
  'start': 17,
  'end': 25},
 {'entity': 'B-location',
  'score': 0.65523505,
  'index': 13,
  'word': 'san',
  'start': 80,
  'end': 83},
 {'entity': 'B-location',
  'score': 0.4668663,
  'index': 14,
  'word': 'francisco',
  'start': 84,
  'end': 93}]
```

You can also manually replicate the results of the `pipeline` if you'd like:

Tokenize the text and return PyTorch tensors:

```
>>> from transformers import AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("stevhliu/my_awesome_wnut_model")
>>> inputs = tokenizer(text, return_tensors="pt")
```

Pass your inputs to the model and return the `logits`:

```
>>> from transformers import AutoModelForTokenClassification

>>> model = AutoModelForTokenClassification.from_pretrained("stevhliu/my_awesome_wnut_
>>> with torch.no_grad():
...     logits = model(**inputs).logits
```

Get the class with the highest probability, and use the model's `id2label` mapping to convert it to a text label:

```
>>> predictions = torch.argmax(logits, dim=2)
>>> predicted_token_class = [model.config.id2label[t.item()] for t in predictions[0]]
>>> predicted_token_class
['O',
 'O',
 'B-location',
 'I-location',
 'B-group',
 'O',
 'O',
 'O',
 'O',
 'O',
 'O',
 'O',
 'O',
 'B-location',
 'B-location',
 'O',
 'O']
```

</> Update on GitHub

← Text classification

Question answering →