Pytorch TensorFlow

Ask a question Open in Colab Open Studio Lab

За конвейером

> Это первый раздел, в котором содержание немного отличается в зависимости от того, используете ли вы PyTorch или TensorFlow. Переключите переключатель в верхней части заголовка, чтобы выбрать платформу, которую вы предпочитаете!

Давайте начнем с полноценного примера и посмотрим, что произошло за кулисами, когда мы выполнили следующий код в Главе 1:

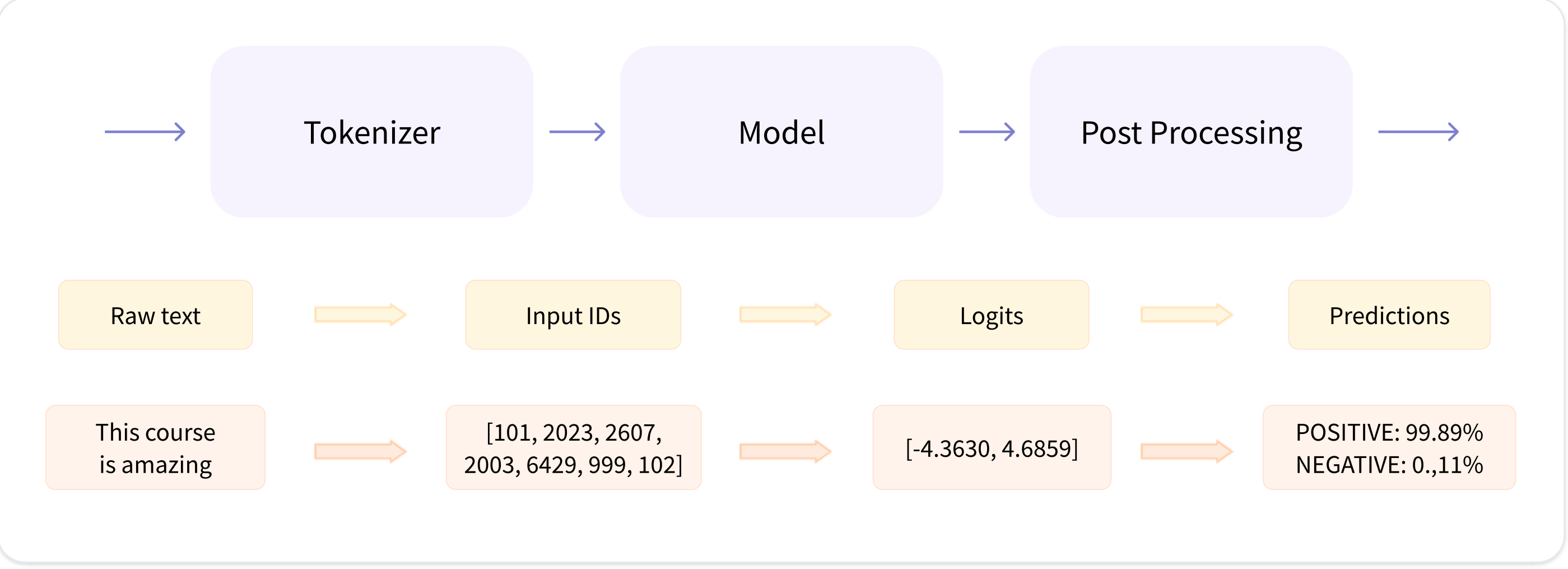
```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
classifier(
    [
        "I've been waiting for a HuggingFace course my whole life.",
        "I hate this so much!",
    ]
)
```

и получили:

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437},
 {'label': 'NEGATIVE', 'score': 0.9994558095932007}]
```

Как мы видели в [Главе 1] (../chapter1), этот конвейер объединяет три этапа: предобработку, пропуск входных данных через модель и постобработку:



Давайте быстро пройдемся по каждому из них.

Предобработка с помощью токенизатора

Как и другие нейронные сети, модели Transformer не могут напрямую обрабатывать сырой текст, поэтому первым шагом нашего конвейера является преобразование текстовых данных в числа, которые сможет воспринимать модель. Для этого мы используем *токенизатор*, который будет отвечать за:

- Разбиение входных данных на слова, подслова или символы (например, пунктуацию), которые называются *токенами*
- Сопоставление каждого токена с целым числом
- Добавление дополнительных входных данных, которые могут быть полезны для модели

Вся эта предобработка должна быть выполнена точно так же, как и при предварительном обучении модели, поэтому сначала нам нужно загрузить эту информацию из Model Hub. Для этого мы используем класс `AutoTokenizer` и его метод `from_pretrained()`. Используя имя контрольной точки нашей модели, он автоматически получает данные, ассоциированные с токенизатором модели, и кэширует их (таким образом, они скачиваются только в первый раз, когда вы выполняете приведенный ниже код).

Поскольку контрольной точкой по умолчанию конвейера `sentiment-analysis` является `distilbert-base-uncased-finetuned-sst-2-english` (вы можете посмотреть ее карточку модели здесь), мы выполняем следующее:

```
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

Когда у нас есть токенизатор, мы можем напрямую передать ему наши предложения и получить в ответ словарь, готовый к передаче в нашу модель! Осталось только преобразовать список входных идентификаторов в тензоры.

Вы можете использовать 🤗 Transformers, не задумываясь о том, какой ML-фреймворк используется в качестве бэкенда; это может быть PyTorch или TensorFlow, или Flax для некоторых моделей. Однако модели Transformer принимают на вход только *тензоры*. Если вы впервые слышите о тензорах, то можете считать их массивами NumPy. Массив NumPy может быть скаляром (0D), вектором (1D), матрицей (2D) или иметь больше измерений. По сути, это тензор; тензоры других ML-фреймворков ведут себя аналогично, и их обычно так же просто инстанцировать, как и массивы NumPy.

Чтобы указать тип тензоров, которые мы хотим получить в ответ (PyTorch, TensorFlow или обычный NumPy), мы используем аргумент `return_tensors`:

```
raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
]
inputs = tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="pt")
print(inputs)
```

Не беспокойтесь пока о дополнении и усечении, мы расскажем об этом позже. Главное, что нужно запомнить: вы можете передать одно предложение или список предложений, а также указать тип тензоров, которые вы хотите получить в ответ (если тип не указан, то в результате вы получите список списков).

Вот как выглядят результаты в виде тензоров PyTorch:

```
{
    'input_ids': tensor([
        [  101,  1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662, 12172, 2607,  2026,  2878,  2166,
        [  101,  1045,  5223,  2023,  2061,  2172,   999,   102,     0,     0,     0,     0,     0,
    ]),
    'attention_mask': tensor([
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
    ])
}
```

Сам результат представляет собой словарь, содержащий два ключа, `input_ids` и `attention_mask`. `input_ids` содержит две строки целых чисел (по одной на каждое предложение), которые являются уникальными идентификаторами токенов в каждом предложении. Мы объясним, что такое `attention_mask` позже в этой главе.

Проходя сквозь модель

Мы можем загрузить нашу предварительно обученную модель так же, как мы это делали с нашим токенизатором. 🤗 Transformers предоставляет класс `AutoModel`, у которого также есть метод `from_pretrained()`:

```
from transformers import AutoModel

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModel.from_pretrained(checkpoint)
```

В этом фрагменте кода мы загрузили ту же контрольную точку, которую использовали в нашем конвейере ранее (на самом деле она уже должна была быть кэширована), и инстанцировали модель с ее помощью.

Эта архитектура содержит только базовый модуль Transformer: при наличии некоторых входных данных он выводит то, что мы будем называть *скрытыми состояниями (hidden states)*, также известными как *признаки (features)*. Для каждого входа модели мы получим многомерный вектор, представляющий **контекстное понимание этого входа моделью Transformer**.

Если вы не поняли смысла, не волнуйтесь. Мы объясним всё это позже.

Хотя эти скрытые состояния могут быть полезны сами по себе, обычно они являются входными данными для другой части модели, известной как *голова (head)*. В Главе 1 различные задачи могли быть выполнены с помощью одной и той же архитектуры, но каждая из этих задач будет связана с разной головой.

Многомерный вектор?

Вектор, возвращаемый модулем Transformer, обычно большой. Обычно он имеет три измерения:

- **Размер батча (Batch size)**: Количество последовательностей, обрабатываемых за один раз (в нашем примере - 2).
- **Длина последовательности (Sequence length)**: Длина числового представления последовательности (в нашем примере - 16).
- **Скрытый размер (Hidden size)**: Размерность вектора каждого входа модели.

О нем говорят как о “многомерном” из-за последнего значения. Скрытый размер может быть очень большим (768 - обычное явление для небольших моделей, а в больших моделях он может достигать 3072 и более).

Мы можем убедиться в этом, если подадим в нашу модель входные данные, которые мы подвергли предобработке:

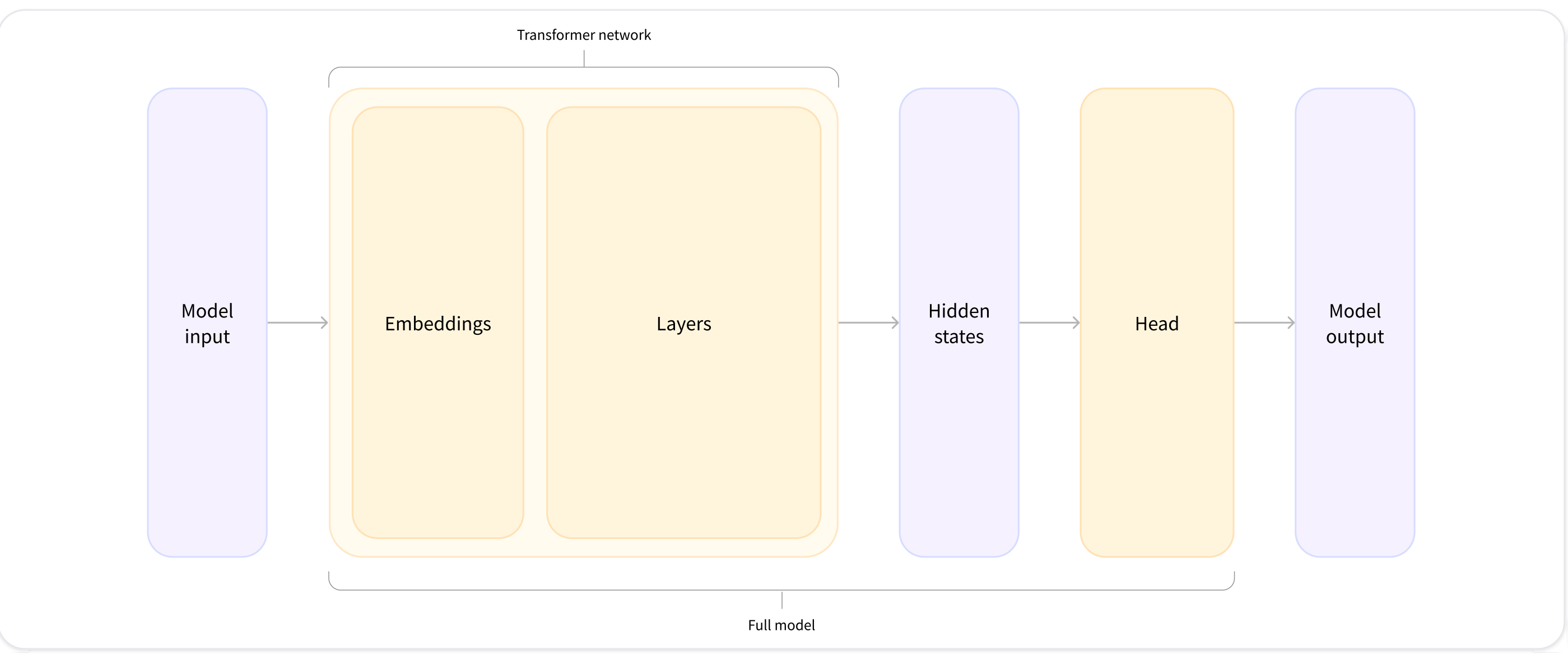
```
outputs = model(**inputs)
print(outputs.last_hidden_state.shape)
```

```
torch.Size([2, 16, 768])
```

Обратите внимание, что выходы моделей 🤗 Transformers ведут себя как `именованные кортежи` или словари. Вы можете обращаться к элементам по атрибутам (как мы это делали), по ключу (`outputs["last_hidden_state"]`) или даже по индексу, если вы точно знаете, где находится искомый элемент (`outputs[0]`).

Головы моделей: Извлечение смысла из чисел

Головы модели принимают на вход многомерный вектор скрытых состояний и проецируют его на другое измерение. Обычно они состоят из одного или нескольких линейных слоев:



Выход модели Transformer передается непосредственно в голову модели для обработки.

На этой диаграмме модель представлена слоем эмбеддингов и последующими слоями. Слой эмбеддингов преобразует каждый входной идентификатор в токенизированном входе в вектор, который представляет соответствующий токен. Последующие слои манипулируют этими векторами с помощью механизма внимания, чтобы получить окончательное представление предложений.

Существует множество различных архитектур 🤗 Transformers, каждая из которых предназначена для решения определенной задачи. Вот неполный список:

- `*Model` (извлечение скрытых состояний)
- `*ForCausalLM`
- `*ForMaskedLM`
- `*ForMultipleChoice`
- `*ForQuestionAnswering`
- `*ForSequenceClassification`
- `*ForTokenClassification`
- и другие 🤗

Для нашего примера нам понадобится модель с головой классификации последовательности (чтобы иметь возможность классифицировать предложения как положительные или отрицательные). Поэтому мы будем использовать не класс `AutoModel`, а `AutoModelForSequenceClassification`:

```
from transformers import AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
outputs = model(**inputs)
```

Теперь, если мы посмотрим на форму наших выходов, размерность будет гораздо ниже: голова модели принимает на вход высокоразмерные векторы, которые мы видели ранее, и возвращает векторы, содержащие два значения (по одному на метку):

```
print(outputs.logits.shape)
```

```
torch.Size([2, 2])
```

Поскольку у нас всего два предложения и две метки, результат, полученный с помощью нашей модели, имеет форму 2 x 2.

Постобработка вывода

Значения, которые мы получаем на выходе из нашей модели, не всегда имеют смысл сами по себе. Давайте посмотрим:

```
print(outputs.logits)
```

```
tensor([[-1.5607,  1.6123],
        [ 4.1692, -3.3464]], grad_fn=<AddmmBackward>)
```

Наша модель спрогнозировала `[-1.5607, 1.6123]` для первого предложения и `[ 4.1692, -3.3464]` для второго. Это не вероятности, а *логиты*, сырые, ненормированные оценки, выведенные последним слоем модели. Чтобы преобразовать их в вероятности, они должны пройти через слой SoftMax (все модели 🤗 Transformers выводят логиты, поскольку функция потерь для обучения обычно объединяет последнюю функцию активации, такую как SoftMax, с фактической функцией потерь, такой как кросс-энтропия):

```
import torch

predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
print(predictions)
```

```
tensor([[4.0195e-02, 9.5980e-01],
        [9.9946e-01, 5.4418e-04]], grad_fn=<SoftmaxBackward>)
```

Теперь мы видим, что модель спрогнозировала `[0.0402, 0.9598]` для первого предложения и `[0.9995, 0.0005]` для второго. Это узнаваемые оценки вероятности.

Чтобы получить метки, соответствующие каждой позиции, мы можем обратиться к атрибуту `id2label` в конфигурации модели (более подробно об этом в следующем разделе):

```
model.config.id2label
```

```
{0: 'NEGATIVE', 1: 'POSITIVE'}
```

Теперь мы можем сделать вывод, что модель спрогнозировала следующее:

- Первое предложение: NEGATIVE: 0.0402, POSITIVE: 0.9598
- Второе предложение: NEGATIVE: 0.9995, POSITIVE: 0.0005

Мы успешно воспроизвели три этапа конвейера: предобработку с помощью токенизаторов, прохождение входных данных через модель и постобработку! Теперь давайте уделим немного времени тому, чтобы углубиться в каждый из этих этапов.

> ✏️ **Попробуйте!** Выберите два (или более) собственных текста и пропустите их через конвейер `sentiment-analysis`. Затем повторите описанные здесь шаги и убедитесь, что вы получили те же результаты!