Pytorch TensorFlow

Ask a question Open in Colab

Токенизаторы

Токенизаторы - один из основных компонентов конвейера NLP. Они служат одной цели: преобразовать текст в данные, которые могут быть обработаны моделью. Модели могут обрабатывать только числа, поэтому токенизаторы должны преобразовывать наш текст в числовые данные. В этом разделе мы рассмотрим, что именно происходит в конвейере токенизации.

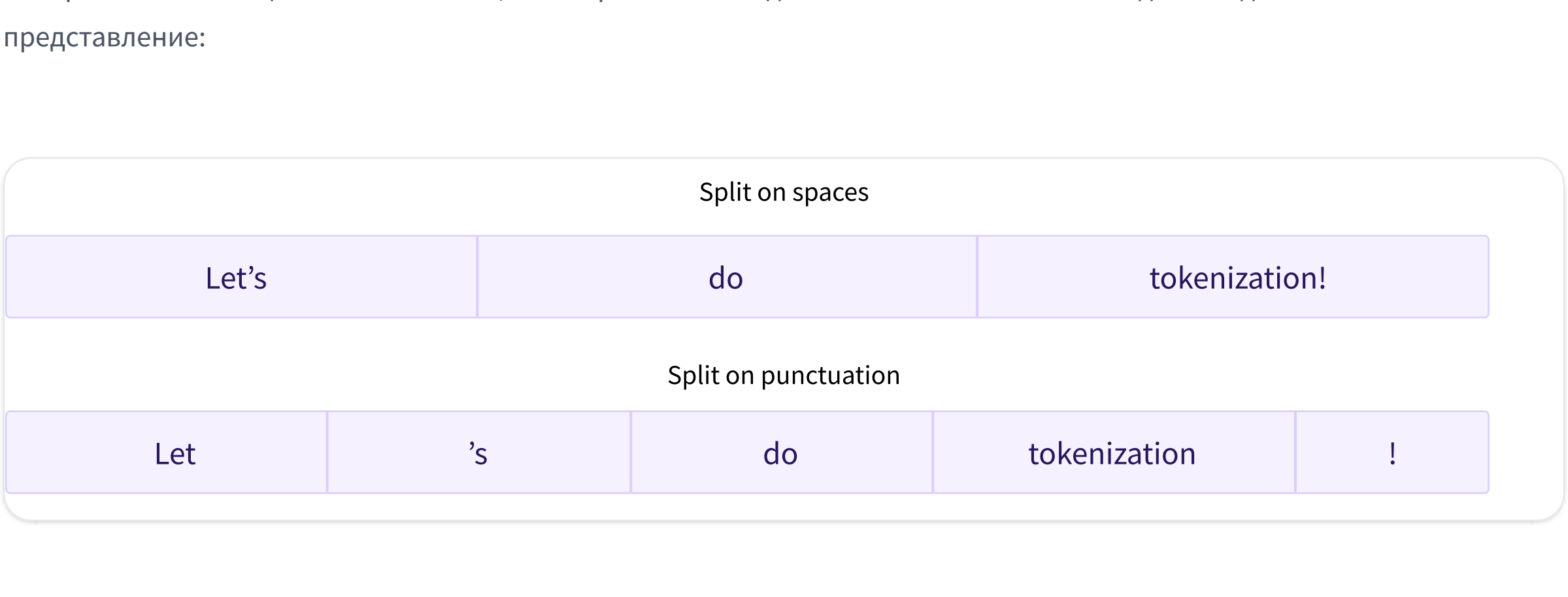
В задачах NLP данные, которые обычно подвергаются обработке, представляют собой необработанный текст. Вот пример такого текста:

```
Jim Henson was a puppeteer
```

Однако модели могут обрабатывать только числа, поэтому нам нужно найти способ преобразовать исходный текст в числа. Этим занимаются токенизаторы, и существует множество способов сделать это. Цель состоит в том, чтобы найти наиболее осмысленное представление - то есть то, которое имеет наибольший смысл для модели, - и, если возможно, наименьшее представление.

Давайте рассмотрим несколько примеров алгоритмов токенизации и постараемся ответить на некоторые вопросы, которые могут у вас возникнуть по токенизации.

На основе слов



Первый тип токенайзера, который приходит на ум, - это *на основе слов (word-based)*. Как правило, его очень легко настроить и использовать с помощью всего нескольких правил, и он часто дает достойные результаты. Например, на изображении ниже цель состоит в том, чтобы разбить исходный текст на слова и найти для каждого из них числовое представление:

Split on spaces

Let’s	do	tokenization!

Split on punctuation

Let	’s	do	tokenization	!

Разделить текст можно разными способами. Например, мы можем использовать пробельные символы, чтобы разделить текст на слова, применив функцию Python `split()`:

```
tokenized_text = "Jim Henson was a puppeteer".split()
print(tokenized_text)
```

```
['Jim', 'Henson', 'was', 'a', 'puppeteer']
```

Существуют также разновидности токенизаторов слов, которые содержат дополнительные правила для пунктуации. Используя такой токенизатор, мы можем получить довольно большие “словари”, где словарь определяется общим количеством независимых токенов, которые есть в нашем корпусе.

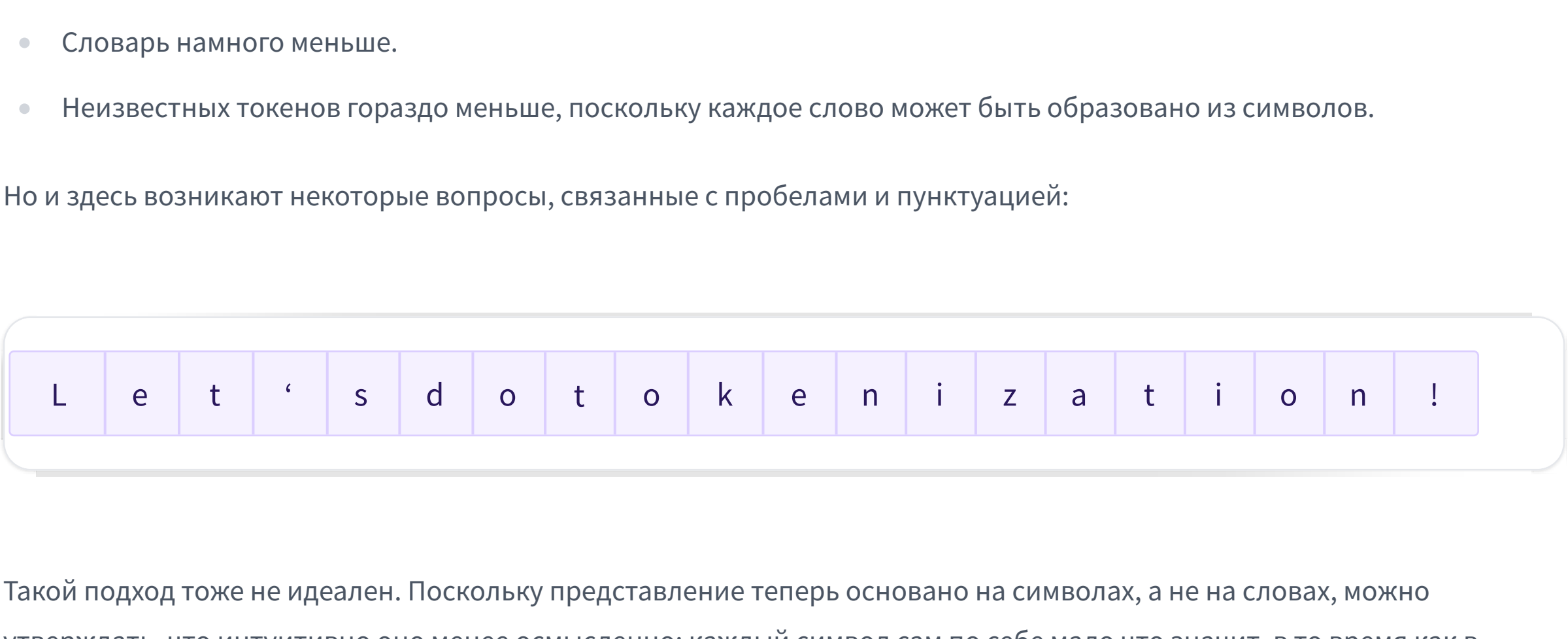
Каждому слову присваивается идентификатор, начиная с 0 и заканчивая размером словаря. Модель использует эти идентификаторы для идентификации каждого слова.

Если мы хотим полностью покрыть язык с помощью токенизатора, основанного на словах, нам понадобится идентификатор для каждого слова в языке, что приведет к созданию огромного количества токенов. Например, в английском языке более 500 000 слов, поэтому, чтобы построить карту соответствия каждого слова входному идентификатору, нам нужно будет отслеживать такое количество идентификаторов. Кроме того, такие слова, как “dog”, представляются иначе, чем слова типа “dogs”, и модель изначально не будет знать, что “dog” и “dogs” похожи: она определит эти два слова как несвязанные. То же самое относится и к другим похожим словам, например “run” и “running”, которые модель изначально не будет воспринимать как похожие.

Наконец, нам нужен специальный токен для обозначения слов, которых нет в нашем словаре. Это так называемый “unknown” токен, часто представляемый как ”[UNK]” или ”<unk>”. Обычно это плохой знак, если вы видите, что токенизатор выдает много таких токенов, поскольку он не смог получить разумное представление слова, и вы теряете информацию на этом этапе. При создании словаря целью является сделать это таким образом, чтобы токенизатор как можно меньше слов токенизировал как неизвестный токен.

Один из способов уменьшить количество неизвестных токенов - это пойти на один уровень глубже, используя *основанный на символах (character-based)* токенизатор.

На основе символов



Токенизаторы на основе символов (character-based) разбивают текст на символы, а не на слова. Это дает два основных преимущества:

- Словарь намного меньше.
- Неизвестных токенов гораздо меньше, поскольку каждое слово может быть образовано из символов.

Но и здесь возникают некоторые вопросы, связанные с пробелами и пунктуацией:

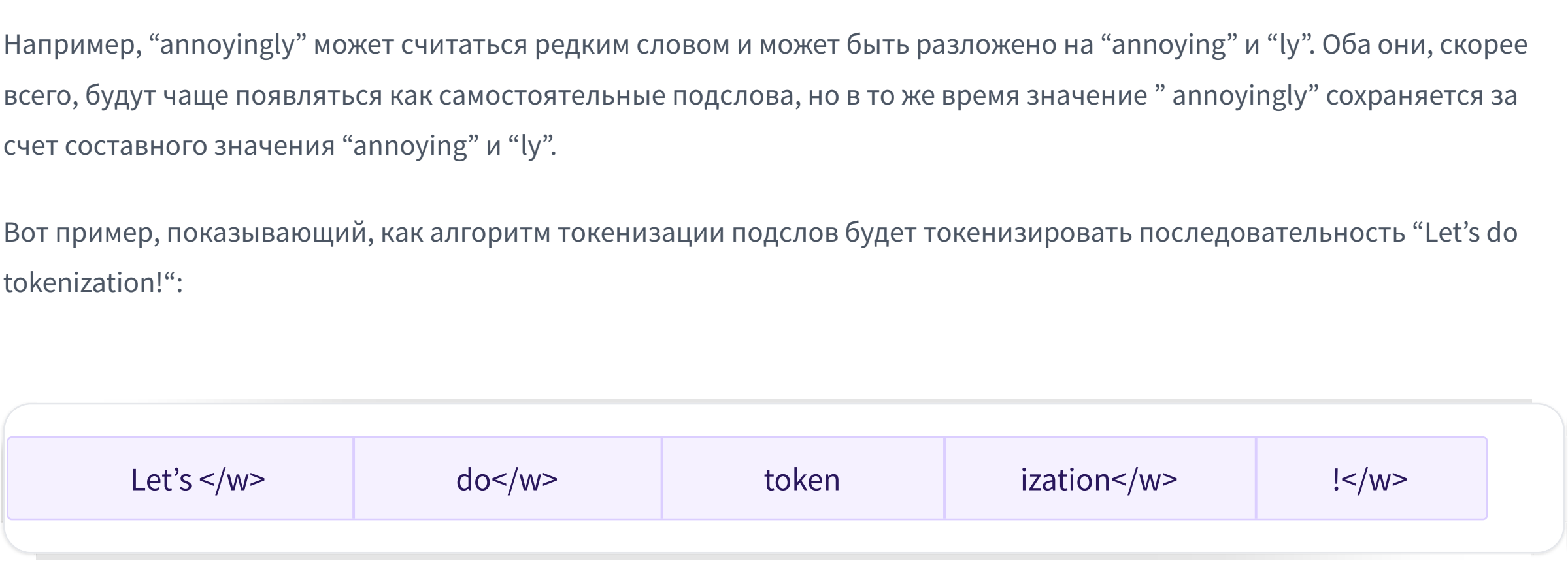
L	e	t	‘	s	d	o	t	o	k	e	n	i	z	a	t	i	o	n	!

Такой подход тоже не идеален. Поскольку представление теперь основано на символах, а не на словах, можно утверждать, что интуитивно оно менее осмысленно: каждый символ сам по себе мало что значит, в то время как в случае со словами это не так. Однако это опять же зависит от языка: например, в Китайском языке каждый символ несет больше информации, чем символ в латинском языке.

Еще один момент, который следует учитывать, - это то, что в итоге мы получим очень большое количество токенов для обработки нашей моделью: если при использовании токенизатора, основанного на словах, слово будет состоять только из одного токена, то при преобразовании в символы оно может легко превратиться в 10 или более токенов.

Чтобы получить лучшее из обоих миров, мы можем использовать третью технику, которая объединяет эти два подхода: *токенизацию по подсловам (subword tokenization)*.

Токенизация по подсловам



Алгоритмы токенизации подслов (subword tokenization) основываются на принципе, согласно которому часто используемые слова не должны разбиваться на более мелкие подслова, а редкие слова должны быть разложены на значимые подслова.

Например, “annoyingly” может считаться редким словом и может быть разложено на “annoying” и “ly”. Оба они, скорее всего, будут чаще появляться как самостоятельные подслова, но в то же время значение ” annoyingly” сохраняется за счет составного значения “annoying” и “ly”.

Вот пример, показывающий, как алгоритм токенизации подслов будет токенизировать последовательность “Let’s do tokenization!“:

Let’s </w>	do</w>	token	ization</w>	!</w>

Эти подслова в конечном итоге несут в себе большой семантический смысл: например, в приведенном выше примере “tokenization” было разделено на “token” и “ization” - два токена, которые несут в себе семантический смысл и при этом занимают мало места (для представления длинного слова требуется всего два токена). Это позволяет нам получить относительно хорошее покрытие при небольшом размере словаря и почти полном отсутствии неизвестных токенов.

Этот подход особенно полезен в агглютинативных языках, таких как турецкий, где вы можете образовывать (почти) произвольно длинные сложные слова, соединяя подслова.

И не только!

Неудивительно, что существует множество других техник. Вот лишь некоторые из них:

- Byte-level BPE, на уровне байтов, используется в GPT-2
- WordPiece, используемый в BERT
- SentencePiece или Unigram, используемый в нескольких многоязычных моделях

Теперь у вас должно быть достаточно знаний о том, как работают токенизаторы, чтобы приступить к работе с API.

Загрузка и сохранение

Загрузка и сохранение токенизаторов так же проста, как и в случае с моделями. Фактически, они основаны на тех же двух методах: `from_pretrained()` и `save_pretrained()`. Эти методы загружают или сохраняют алгоритм, используемый токенизатором (что-то вроде *архитектуры* модели), а также его словарь (что-то вроде *весов* модели).

Загрузка токенизатора BERT, обученного на той же контрольной точке, что и BERT, выполняется так же, как и загрузка модели, за исключением того, что мы используем класс `BertTokenizer`:

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-cased")
```

Подобно `AutoModel`, класс `AutoTokenizer` будет захватывать нужный класс токенизатора в библиотеке, основываясь на имени контрольной точки, и может быть использован непосредственно с любой контрольной точкой:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

Теперь мы можем использовать токенизатор, как показано в предыдущем разделе:

```
tokenizer("Using a Transformer network is simple")
```

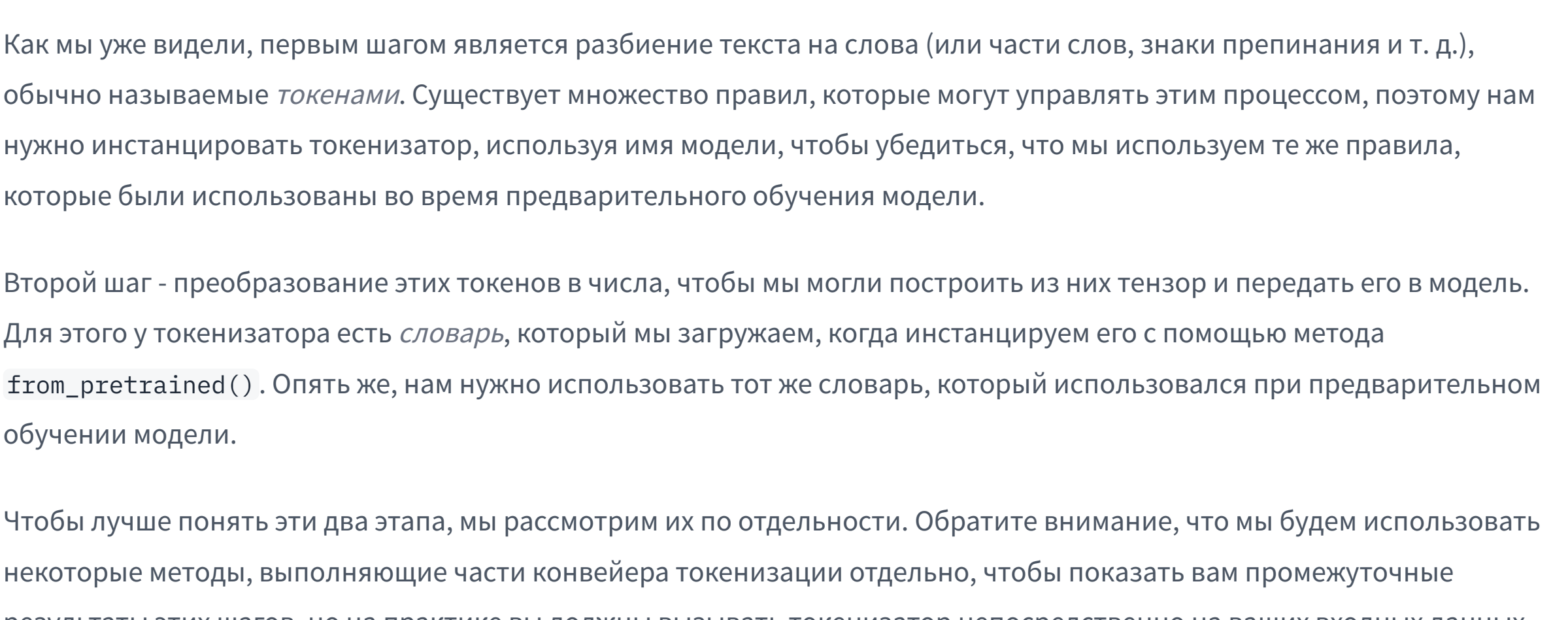
```
{'input_ids': [101, 7993, 170, 11303, 1200, 2443, 1110, 3014, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

Сохранение токенизатора идентично сохранению модели:

```
tokenizer.save_pretrained("directory_on_my_computer")
```

Подробнее о `token_type_ids` мы поговорим в Главе 3, а ключ `attention_mask` мы объясним чуть позже. Сначала давайте посмотрим, как генерируются `input_ids`. Для этого нам понадобится рассмотреть промежуточные методы токенизатора.

Кодирование



Перевод текста в числа называется *кодированием (encoding)*. Кодирование выполняется в два этапа: токенизация, а затем преобразование во входные идентификаторы.

Как мы уже видели, первым шагом является разбиение текста на слова (или части слов, знаки препинания и т. д.), обычно называемые *токенами*. Существует множество правил, которые могут управлять этим процессом, поэтому нам нужно инстанцировать токенизатор, используя имя модели, чтобы убедиться, что мы используем те же правила, которые были использованы во время предварительного обучения модели.

Второй шаг - преобразование этих токенов в числа, чтобы мы могли построить из них тензор и передать его в модель. Для этого у токенизатора есть *словарь*, который мы загружаем, когда инстанцируем его с помощью метода `from_pretrained()`. Опять же, нам нужно использовать тот же словарь, который использовался при предварительном обучении модели.

Чтобы лучше понять эти два этапа, мы рассмотрим их по отдельности. Обратите внимание, что мы будем использовать некоторые методы, выполняющие части конвейера токенизации отдельно, чтобы показать вам промежуточные результаты этих шагов, но на практике вы должны вызывать токенизатор непосредственно на ваших входных данных (как показано в разделе 2).

Токенизация

Процесс токенизации выполняется методом `tokenize()` токенизатора:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

sequence = "Using a Transformer network is simple"
tokens = tokenizer.tokenize(sequence)

print(tokens)
```

Результатом работы этого метода является список строк, или токенов:

```
['Using', 'a', 'transform', '##er', 'network', 'is', 'simple']
```

Этот токенизатор является токенизатором подслов: он разбивает слова до тех пор, пока не получит токены, которые могут быть представлены в его словаре. В данном случае слово `transformer` разбивается на два токена: `transform` и `##er`.

От токенов к идентификаторам входа

Преобразование во входные идентификаторы выполняется методом токенизатора `convert_tokens_to_ids()`:

```
ids = tokenizer.convert_tokens_to_ids(tokens)

print(ids)
```

```
[7993, 170, 11303, 1200, 2443, 1110, 3014]
```

Эти выходы, преобразованные в тензор соответствующего фреймворка, могут быть использованы в качестве входов в модель, как было показано ранее в этой главе.

> ✏️ **Попробуйте!** Повторите два последних шага (токенизацию и преобразование во входные идентификаторы) на входных предложениях, которые мы использовали в разделе 2 (“I’ve been waiting for a HuggingFace course my whole life.” и “I hate this so much!”). Убедитесь, что вы получили те же самые входные идентификаторы, которые мы получали ранее!

Декодирование

Декодирование происходит наоборот: из индексов словаря мы хотим получить строку. Это можно сделать с помощью метода `decode()` следующим образом:

```
decoded_string = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110, 3014])
print(decoded_string)
```

```
'Using a Transformer network is simple'
```

Обратите внимание, что метод `decode` не только преобразует индексы обратно в токены, но и группирует токены, которые были частью одних и тех же слов, чтобы создать читаемое предложение. Такое поведение будет очень полезно, когда мы будем использовать модели, прогнозирующие новый текст (либо текст, сгенерированный из подсказки (prompt), либо для решения задачи преобразования последовательности-в-последовательность (sequence-to-sequence), такой как перевод или резюмирование).

Теперь вы должны понимать, какие атомарные операции может выполнять токенизатор: токенизация, преобразование в идентификаторы и преобразование идентификаторов обратно в строку. Однако мы лишь пощупали верхушку айсберга. В следующем разделе мы рассмотрим ограничения нашего подхода и посмотрим, как их преодолеть.