Токенизация WordPiece

Ask a question   Open in Colab   Open Studio Lab

WordPiece - это алгоритм токенизации, разработанный Google для предварительного обучения BERT. Впоследствии он был повторно использован во многих моделях трансформеров, основанных на BERT, таких как DistilBERT, MobileBERT, Funnel Transformers и MPNET. Он очень похож на BPE в плане обучения, но фактическая токенизация выполняется по-другому.

> 💡 В этом разделе подробно рассматривается WordPiece, вплоть до демонстрации полной реализации. Вы можете пропустить его, если вам нужен только общий обзор алгоритма токенизации.

Алгоритм обучения

> ⚠️ Google никогда не предоставлял открытый доступ к своей реализации алгоритма обучения WordPiece, поэтому все вышесказанное - это наши предположения, основанные на опубликованных материалах. Возможно, они точны не на 100 %.

Как и BPE, WordPiece начинает работу с небольшого словаря, включающего специальные токены, используемые моделью, и начальный алфавит. Поскольку модель идентифицирует подслова путем добавления префикса (как `##` для BERT), каждое слово первоначально разбивается на части путем добавления этого префикса ко всем символам внутри слова. Так, например, `"word"` разбивается на части следующим образом:

```
w ##o ##r ##d
```

Таким образом, начальный алфавит содержит все символы, присутствующие в начале слова, и символы, присутствующие внутри слова, которым предшествует префикс WordPiece.

Затем, как и в случае с BPE, WordPiece изучает правила слияния. Основное отличие заключается в способе выбора пары для слияния. Вместо того чтобы выбирать наиболее частую пару, WordPiece рассчитывает оценку для каждой пары по следующей формуле:

$$\mathrm{score} = (\mathrm{freq\_of\_pair})/(\mathrm{freq\_of\_first\_element} \times \mathrm{freq\_of\_second\_element})$$

Деля частоту пары на произведение частот каждой из ее частей, алгоритм отдает предпочтение слиянию пар, отдельные части которых встречаются в словаре реже. Например, он не обязательно объединит `("un", "##able")`, даже если эта пара встречается в словаре очень часто, потому что две пары `"un"` и `"##able"`, скорее всего, встречаются в большом количестве других слов и имеют высокую частоту. Напротив, такая пара, как `("hu", "##gging")`, вероятно, будет объединена быстрее (при условии, что слово “hugging” часто встречается в словаре), поскольку `"hu"` и `"##gging"` по отдельности, скорее всего, встречаются реже.

Давайте рассмотрим тот же словарь, который мы использовали в учебном примере BPE:

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

Рабиение здесь будет следующим:

```
("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##g
```

поэтому исходный словарь будет иметь вид `["b", "h", "p", "##g", "##n", "##s", "##u"]` (если мы пока забудем о специальных токенах). Самая частая пара - `("##u", "##g")` (встречается 20 раз), но индивидуальная частота `"##u"` очень высока, поэтому ее оценка не самая высокая (она составляет 1/36). Все пары с `"##u"` фактически имеют такую же оценку (1/36), поэтому лучшую оценку получает пара `("##g", "##s")` - единственная, в которой нет `"##u"` - с оценкой 1/20, и первым выученным слиянием будет `("##g", "##s") -> ("##gs")`.

Обратите внимание, что при слиянии мы удаляем `##` между двумя токенами, поэтому мы добавляем `"##gs"` в словарь и применяем слияние в словах корпуса:

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs"]
Corpus: ("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "#
```

В этот момент `"##u"` находится во всех возможных парах, поэтому все они получают одинаковый балл. Допустим, в этом случае первая пара объединяется, так что `("h", "##u") -> "hu"`. Это приводит нас к:

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu"]
Corpus: ("hu" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("hu" "##gs'
```

Затем следующую лучшую оценку разделяют `("hu", "##g")` и `("hu", "##gs")` (1/15, по сравнению с 1/21 для всех остальных пар), поэтому первая пара с наибольшей оценкой объединяется:

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu", "hug"]
Corpus: ("hug", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("hu" "##gs", 5)
```

и мы продолжаем так до тех пор, пока не достигнем необходимого размера словаря.

> ✏️ **Теперь ваша очередь!** Каким будет следующее правило слияния?

Алгоритм токенизации

Токенизация в WordPiece и BPE отличается тем, что WordPiece сохраняет только конечный словарь, а не выученные правила слияния. Начиная со слова, которое нужно токенизировать, WordPiece находит самое длинное подслово, которое есть в словаре, а затем разбивает его на части. Например, если мы используем словарь, изученный в примере выше, для слова `" hugs"` самым длинным подсловом, начиная с начала, которое находится в словаре, является `"hug"`, поэтому мы делим его на части и получаем `["hug", "##s"]`. Затем мы продолжаем с `"##s"`, которое находится в словаре, поэтому токенизация `"hugs"` будет `["hug", "##s"]`.

В BPE мы бы применили слияния, выученные по порядку, и токенизировали это как `["hu", "##gs"]`, поэтому кодировка отличается.

В качестве другого примера посмотрим, как будет токенизировано слово `"bugs"`. `"b"` - самое длинное подслово, начинающееся с начала слова, которое есть в словаре, поэтому мы делим его на части и получаем `["b", "##ugs"]`. Затем `"##u"` - самое длинное подслово, начинающееся в начале `"##ugs"`, которое есть в словаре, поэтому мы делим его на части и получаем `["b", "##u", "##gs"]`. Наконец, `"##gs"` находится в словаре, так что этот последний список является токеном `"bugs"`.

Когда токенизация доходит до стадии, когда невозможно найти подслово в словаре, все слово токенизируется как неизвестное - так, например, `"mug"` будет токенизировано как `["[UNK]"]`, как и `"bum"` (даже если мы можем начать с `"b"` и `"##u"`, `"##m"` не входит в словарь, и результирующий токен будет просто `["[UNK]"]`, а не `["b", "##u", "[UNK]"]`). Это еще одно отличие от BPE, который классифицирует как неизвестные только отдельные символы, отсутствующие в словаре.

> ✏️ **Теперь ваша очередь!** Как будет токенизировано слово `"pugs"`?

Реализация WordPiece

Теперь давайте посмотрим на реализацию алгоритма WordPiece. Как и в случае с BPE, это всего лишь учебный пример, и вы не сможете использовать его на большом корпусе.

Мы будем использовать тот же корпус, что и в примере с BPE:

```
corpus = [
    "This is the Hugging Face Course.",
    "This chapter is about tokenization.",
    "This section shows several tokenizer algorithms.",
    "Hopefully, you will be able to understand how they are trained and generate tokens.",
]
```

Во-первых, нам нужно предварительно токенизировать корпус в слова. Поскольку мы воспроизводим токенизатор WordPiece (например, BERT), для предварительной токенизации мы будем использовать токенизатор `bert-base-cased`:

```
from transformers import AutoTokenizer

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

Затем мы вычисляем частоту каждого слова в корпусе, как и при предварительной токенизации:

```
from collections import defaultdict

word_freqs = defaultdict(int)
for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

word_freqs
```

```
defaultdict(
    int, {'This': 3, 'is': 2, 'the': 1, 'Hugging': 1, 'Face': 1, 'Course': 1, '.': 4, 'chapter': 1, 'abou
    'tokenization': 1, 'section': 1, 'shows': 1, 'several': 1, 'tokenizer': 1, 'algorithms': 1, 'Hopefull
    ',': 1, 'you': 1, 'will': 1, 'be': 1, 'able': 1, 'to': 1, 'understand': 1, 'how': 1, 'they': 1, 'are'
    'trained': 1, 'and': 1, 'generate': 1, 'tokens': 1})
```

Как мы уже видели, алфавит - это уникальное множество, состоящее из всех первых букв слов и всех остальных букв, которые встречаются в словах с префиксом `##`:

```
alphabet = []
for word in word_freqs.keys():
    if word[0] not in alphabet:
        alphabet.append(word[0])
    for letter in word[1:]:
        if f"##{letter}" not in alphabet:
            alphabet.append(f"##{letter}")

alphabet.sort()
alphabet

print(alphabet)
```

```
['##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '##i', '##k', '##l', '##m', '##n', '##o', '##p',
 '##t', '##u', '##v', '##w', '##y', '##z', ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'g', 'h', 'i', 's
 'w', 'y']
```

Мы также добавляем специальные токены, используемые моделью, в начало этого словаря. В случае BERT это список `["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"]`:

```
vocab = ["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"] + alphabet.copy()
```

Далее нам нужно разделить каждое слово на части, при этом все буквы, которые не являются первыми, должны иметь префикс `##`:

```
splits = {
    word: [c if i == 0 else f"##{c}" for i, c in enumerate(word)]
    for word in word_freqs.keys()
}
```

Теперь, когда мы готовы к обучению, давайте напишем функцию, которая вычисляет оценку каждой пары. Нам нужно будет использовать ее на каждом шаге обучения:

```
def compute_pair_scores(splits):
    letter_freqs = defaultdict(int)
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            letter_freqs[split[0]] += freq
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            letter_freqs[split[i]] += freq
            pair_freqs[pair] += freq
        letter_freqs[split[-1]] += freq

    scores = {
        pair: freq / (letter_freqs[pair[0]] * letter_freqs[pair[1]])
        for pair, freq in pair_freqs.items()
    }
    return scores
```

Давайте посмотрим на часть этого словаря после первых разделений:

```
pair_scores = compute_pair_scores(splits)
for i, key in enumerate(pair_scores.keys()):
    print(f"{key}: {pair_scores[key]}")
    if i >= 5:
        break
```

```
('T', '##h'): 0.125
('##h', '##i'): 0.03409090909090909
('##i', '##s'): 0.02727272727272727
('i', '##s'): 0.1
('t', '##h'): 0.03571428571428571
('##h', '##e'): 0.011904761904761904
```

Теперь для того, чтобы найти пару с наилучшим результатом, нужно всего лишь сделать быстрый цикл:

```
best_pair = ""
max_score = None
for pair, score in pair_scores.items():
    if max_score is None or max_score < score:
        best_pair = pair
        max_score = score

print(best_pair, max_score)
```

```
('a', '##b') 0.2
```

Итак, первое слияние, которое нужно выучить, это `('a', '##b') -> 'ab'`, и мы добавляем `'ab'` в словарь:

```
vocab.append("ab")
```

Чтобы продолжить, нам нужно применить это слияние в нашем словаре `splits`. Давайте напишем для этого еще одну функцию:

```
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue
        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                merge = a + b[2:] if b.startswith("##") else a + b
                split = split[:i] + [merge] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits
```

И мы можем посмотреть на результат первого слияния:

```
splits = merge_pair("a", "##b", splits)
splits["about"]
```

```
['ab', '##o', '##u', '##t']
```

Теперь у нас есть все, что нужно, чтобы зацикливать процесс до тех пор, пока мы не выучим все слияния, которые нам нужны. Давайте нацелимся на размер словаря равный 70:

```
vocab_size = 70
while len(vocab) < vocab_size:
    scores = compute_pair_scores(splits)
    best_pair, max_score = "", None
    for pair, score in scores.items():
        if max_score is None or max_score < score:
            best_pair = pair
            max_score = score
    splits = merge_pair(*best_pair, splits)
    new_token = (
        best_pair[0] + best_pair[1][2:]
        if best_pair[1].startswith("##")
        else best_pair[0] + best_pair[1]
    )
    vocab.append(new_token)
```

Затем мы можем просмотреть созданный словарь:

```
print(vocab)
```

```
['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', '##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '#
 '##l', '##m', '##n', '##o', '##p', '##r', '##s', '##t', '##u', '##v', '##w', '##y', '##z', ',', '.', 'C'
 'T', 'a', 'b', 'c', 'g', 'h', 'i', 's', 't', 'u', 'w', 'y', 'ab', '##fu', 'Fa', 'Fac', '##ct', '##ful',
 'Th', 'ch', '##hm', 'cha', 'chap', 'chapt', '##thm', 'Hu', 'Hug', 'Hugg', 'sh', 'th', 'is', '##thms', '#
 '##ut']
```

Как мы видим, по сравнению с BPE этот токенизатор быстрее выучивает части слов как токены.

> 💡 Использование `train_new_from_iterator()` на одном и том же корпусе не приведет к точно такому же словарю. Это происходит потому, что библиотека 🤗 Tokenizers не реализует WordPiece для обучения (поскольку мы не полностью уверены в его внутреннем устройстве), а использует вместо него BPE.

Чтобы токенизировать новый текст, мы предварительно токенизируем его, разбиваем на части, а затем применяем алгоритм токенизации к каждому слову. То есть начиная с первого слова мы ищем самое большое подслово и разбиваем его на части, затем мы повторяем процесс для второй части, и так далее для оставшейся части этого слова и следующих слов в тексте:

```
def encode_word(word):
    tokens = []
    while len(word) > 0:
        i = len(word)
        while i > 0 and word[:i] not in vocab:
            i -= 1
        if i == 0:
            return ["[UNK]"]
        tokens.append(word[:i])
        word = word[i:]
        if len(word) > 0:
            word = f"##{word}"
    return tokens
```

Давайте проверим алгоритм на одном слове, которое есть в словаре, и на другом, которого нет:

```
print(encode_word("Hugging"))
print(encode_word("HOgging"))
```

```
['Hugg', '##i', '##n', '##g']
['[UNK]']
```

Теперь давайте напишем функцию, которая токенизирует текст:

```
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    encoded_words = [encode_word(word) for word in pre_tokenized_text]
    return sum(encoded_words, [])
```

Мы можем попробовать его на любом тексте:

```
tokenize("This is the Hugging Face course!")
```

```
['Th', '##i', '##s', 'is', 'th', '##e', 'Hugg', '##i', '##n', '##g', 'Fac', '##e', 'c', '##o', '##u', '##
 '##e', '[UNK]']
```

Вот и все об алгоритме WordPiece! Теперь давайте посмотрим на Unigram.

<> Update on GitHub