Токенизация Unigram

Ask a question Open in Colab Open Studio Lab

Алгоритм Unigram часто используется в SentencePiece, который является алгоритмом токенизации, применяемым в таких моделях, как AlBERT, T5, mBART, Big Bird и XLNet.

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

Алгоритм обучения

По сравнению с BPE и WordPiece, Unigram работает в другом направлении: он начинает с большого словарного запаса и удаляет из него токены, пока не достигнет желаемого размера словаря. Существует несколько вариантов создания базового словаря: например, мы можем взять наиболее часто встречающиеся подстроки в предварительно токенизированных словах или применить BPE к исходному корпусу с большим объемом словаря.

На каждом шаге обучения алгоритм Unigram рассчитывает потери по корпусу с учетом текущего словарного запаса. Затем для каждого символа в словаре алгоритм вычисляет, насколько увеличится общая потеря, если этот символ будет удален, и ищет символы, которые увеличат ее меньше всего. Эти символы оказывают меньшее влияние на общую потерю по корпусу, поэтому в некотором смысле они “менее нужны” и являются лучшими кандидатами на удаление.

Это очень дорогостоящая операция, поэтому мы удаляем не просто один символ, связанный с наименьшим увеличением потерь, аp (\(p\)) - гиперпараметр, которым вы можете управлять, обычно 10 или 20) процентов символов, связанных с наименьшим увеличением потерь. Этот процесс повторяется до тех пор, пока словарь не достигнет желаемого размера.

Обратите внимание, что мы никогда не удаляем базовые символы, чтобы убедиться, что любое слово может быть токенизировано.

Итак, все еще немного туманно: основная часть алгоритма заключается в том, чтобы вычислить потери по корпусу и посмотреть, как они изменяются при удалении некоторых токенов из словаря, но мы еще не объяснили, как это сделать. Этот шаг зависит от алгоритма токенизации модели Unigram, поэтому мы рассмотрим его далее.

Мы используем корпус текста из предыдущих примеров:

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

и для этого примера мы возьмем все подстроки из исходного словаря:

```
["h", "u", "g", "hu", "ug", "p", "pu", "n", "un", "b", "bu", "s", "hug", "gs", "ugs"]
```

Алгоритм токенизации

Модель Unigram - это тип языковой модели, в которой каждый токен рассматривается как независимый от предшествующих ему. Это самая простая языковая модель в том смысле, что вероятность появления токена X с учетом предыдущего контекста - это просто вероятность появления токена X. Таким образом, если бы мы использовали модель Unigram для генерации текста, мы бы всегда предсказывали наиболее часто встречающийся токен.

Вероятность данного токена - это его частота (количество раз, когда мы его находим) в исходном корпусе, деленная на сумму частот всех токенов в словаре (чтобы убедиться, что суммы вероятностей равны 1). Например, `"ug"` присутствует в `"hug"`, `"pug"` и `"hugs"`, поэтому его частота в нашем корпусе равна 20.

Здесь приведены частоты всех возможных подслов в словаре:

```
("h", 15) ("u", 36) ("g", 20) ("hu", 15) ("ug", 20) ("p", 17) ("pu", 17) ("n", 16)
("un", 16) ("b", 4) ("bu", 4) ("s", 5) ("hug", 15) ("gs", 5) ("ugs", 5)
```

Итак, сумма всех частот равна 210, а вероятность появления подслова `"ug"`, таким образом, составляет 20/210.

> ✏️ **Теперь ваша очередь!** Напишите код для вычисления вышеуказанных частот и дважды проверьте правильность приведенных результатов, а также общую сумму.

Теперь для токенизации данного слова мы рассматриваем все возможные сегментации на токены и вычисляем вероятность каждого из них в соответствии с моделью Unigram. Поскольку все токены считаются независимыми, эта вероятность равна произведению вероятностей появления каждого токена. Например, при токенизации `["p", "u", "g"]` слова `"pug"` вероятность составляет:

$$P([``p", ``u", ``g"]) = P(``p") \times P(``u") \times P(``g") = \frac{5}{210} \times \frac{36}{210} \times \frac{20}{210} = 0.000389$$

Для сравнения, токен `["pu", "g"]` имеет вероятность:

$$P([``pu", ``g"]) = P(``pu") \times P(``g") = \frac{5}{210} \times \frac{20}{210} = 0.0022676$$

так что один из них гораздо более вероятен. В целом, токенизации с наименьшим количеством токенов будут иметь наибольшую вероятность (из-за деления на 210, повторяющегося для каждого токена), что соответствует интуитивному желанию: разбить слово на наименьшее количество токенов.

Токенизация слова с помощью модели Unigram - это токенизация с наибольшей вероятностью. В примере с `"pug"` приведены вероятности, которые мы получили бы для каждой возможной сегментации:

```
["p", "u", "g"] : 0.000389
["p", "ug"] : 0.0022676
["pu", "g"] : 0.0022676
```

Так, `"pug"` будет токенизировано как `["p", "ug"]` или `["pu", "g"]`, в зависимости от того, какая из этих сегментаций встретится первой (отметим, что в большом корпусе подобные случаи равенства будут редки).

В данном случае было легко найти все возможные сегментации и вычислить их вероятности, но в общем случае это будет немного сложнее. Для этого используется классический алгоритм, который называется *алгоритм Витерби (Viterbi algorithm)*. По сути, мы можем построить граф для выявления возможных сегментаций данного слова, сказав, что существует ветвь от символа *a* до символа *b*, если подслово от *a* до *b* есть в словаре, и приписать этой ветви вероятность подслова.

Чтобы найти путь в этом графе, который будет иметь наилучшую оценку, алгоритм Витерби определяет для каждой позиции в слове сегментацию с наилучшей оценкой, которая заканчивается на этой позиции. Поскольку мы идем от начала к концу, этот лучший результат можно найти, перебирая все подслова, заканчивающиеся на текущей позиции, а затем используя лучший результат токенизации с позиции, на которой начинается это подслово. Затем нужно просто развернуть путь, чтобы прийти к концу.

Давайте рассмотрим пример с использованием нашего словаря и слова `"unhug"`. Для каждой позиции подслова с наилучшими оценками заканчиваются следующим образом:

```
Character 0 (u): "u" (score 0.171429)
Character 1 (n): "un" (score 0.076191)
Character 2 (h): "un" "h" (score 0.005442)
Character 3 (u): "un" "hu" (score 0.005442)
Character 4 (g): "un" "hug" (score 0.005442)
```

Таким образом, `"unhug"` будет токенизировано как `["un", "hug"]`.

> ✏️ **Теперь ваша очередь!** Определите токенизацию слова `" huggun"` и его оценку.

Назад к обучению

Теперь, когда мы увидели, как работает токенизация, мы можем немного глубже изучить потери, используемые во время обучения. На любом этапе эта потеря вычисляется путем токенизации каждого слова в корпусе с использованием текущего словаря и модели Unigram, определяемой частотами каждого токена в корпусе (как было показано ранее).

Каждое слово в корпусе имеет оценку, а потеря - это отрицательное логарифмическое правдоподобие этих оценок, то есть сумма для всех слов в корпусе всех `-log(P(word))`.

Давайте вернемся к нашему примеру со следующим корпусом:

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

Токенизация каждого слова с соответствующими оценками:

```
"hug": ["hug"] (score 0.071428)
"pug": ["pu", "g"] (score 0.007710)
"pun": ["pu", "n"] (score 0.006168)
"bun": ["bu", "n"] (score 0.001451)
"hugs": ["hug", "s"] (score 0.001701)
```

Таким образом, потери будут:

```
10 * (-log(0.071428)) + 5 * (-log(0.007710)) + 12 * (-log(0.006168)) + 4 * (-log(0.001451)) + 5 * (-log(0
```

Теперь нам нужно вычислить, как удаление каждого токена влияет на потери. Это довольно утомительно, поэтому мы просто сделаем это для двух токенов и оставим весь процесс на потом, когда у нас будет код, чтобы помочь нам. В этом (очень) конкретном случае у нас есть две эквивалентные токенизации всех слов: как мы видели ранее, например, `"pug"` может быть токенизировано `["p", "ug"]` с тем же результатом. Таким образом, удаление токена `"pu"` из словаря приведет к точно таким же потерям.

С другой стороны, удаление `" hug"` усугубит потери, потому что токенизация `"hug"` и `"hugs"` станет:

```
"hug": ["hu", "g"] (score 0.006802)
"hugs": ["hu", "gs"] (score 0.001701)
```

Эти изменения приведут к увеличению потерь:

```
- 10 * (-log(0.071428)) + 10 * (-log(0.006802)) = 23.5
```

Поэтому токен `"pu"`, вероятно, будет удален из словаря, но не `"hug"`.

Реализация Unigram

Теперь давайте реализуем все, что мы видели до сих пор, в коде. Как и в случае с BPE и WordPiece, это не эффективная реализация алгоритма Unigram (совсем наоборот), но она должна помочь вам понять его немного лучше.

В качестве примера мы будем использовать тот же корпус текста, что и раньше:

```
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.",
]
```

На этот раз в качестве модели мы будем использовать `xlnet-base-cased`:

```
from transformers import AutoTokenizer

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

Как и в случае с BPE и WordPiece, мы начинаем с подсчета количества вхождений каждого слова в корпус:

```
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
```

Затем нам нужно инициализировать наш словарь чем-то большим, чем размер словаря, который мы захотим получить в конце. Мы должны включить все основные символы (иначе мы не сможем токенизировать каждое слово), но для больших подстрок мы сохраним только самые распространенные, поэтому мы отсортируем их по частоте:

```
char_freqs = defaultdict(int)
subwords_freqs = defaultdict(int)
for word, freq in word_freqs.items():
    for i in range(len(word)):
        char_freqs[word[i]] += freq
        # Перебираем подслова длиной не менее 2
        for j in range(i + 2, len(word) + 1):
            subwords_freqs[word[i:j]] += freq

# Сортировка подслов по частоте
sorted_subwords = sorted(subwords_freqs.items(), key=lambda x: x[1], reverse=True)
sorted_subwords[:10]
```

```
[('_t', 7), ('is', 5), ('er', 5), ('_a', 5), ('_to', 4), ('to', 4), ('en', 4), ('_T', 3), ('_Th', 3), ('_
```

Мы группируем символы с лучшими подсловами, чтобы получить начальный словарь размером 300:

```
token_freqs = list(char_freqs.items()) + sorted_subwords[: 300 - len(char_freqs)]
token_freqs = {token: freq for token, freq in token_freqs}
```

> 💡 SentencePiece использует более эффективный алгоритм под названием Enhanced Suffix Array (ESA) для создания начального словаря.

Далее мы вычисляем сумму всех частот, чтобы преобразовать частоты в вероятности. Для нашей модели мы будем хранить логарифмы вероятностей, потому что численно стабильнее складывать логарифмы, чем перемножать маленькие числа, и это упростит вычисление потерь модели:

```
from math import log

total_sum = sum([freq for token, freq in token_freqs.items()])
model = {token: -log(freq / total_sum) for token, freq in token_freqs.items()}
```

Теперь основная функция - это функция токенизации слов с помощью алгоритма Витерби. Как мы уже видели, этот алгоритм вычисляет наилучшую сегментацию каждой подстроки слова, которую мы будем хранить в переменной с именем `best_segmentations`. Мы будем хранить по одному словарю на каждую позицию в слове (от 0 до его полной длины), с двумя ключами: индекс начала последнего токена в лучшей сегментации и оценка лучшей сегментации. По индексу начала последнего токена мы сможем получить полную сегментацию, когда список будет полностью заполнен.

Пополнение списка осуществляется с помощью двух циклов: основной цикл просматривает каждую начальную позицию, а второй цикл перебирает все подстроки, начинающиеся с этой начальной позиции. Если подстрока есть в словаре, мы получаем новую сегментацию слова до этой конечной позиции, которую сравниваем с той, что хранится в `best_segmentations`.

После завершения основного цикла мы просто начинаем с конца и переходим от одной начальной позиции к другой, записывая токены по мере продвижения, пока не достигнем начала слова:

```
def encode_word(word, model):
    best_segmentations = [{"start": 0, "score": 1}] + [
        {"start": None, "score": None} for _ in range(len(word))
    ]
    for start_idx in range(len(word)):
        # Это должно быть правильно заполнено предыдущими шагами цикла
        best_score_at_start = best_segmentations[start_idx]["score"]
        for end_idx in range(start_idx + 1, len(word) + 1):
            token = word[start_idx:end_idx]
            if token in model and best_score_at_start is not None:
                score = model[token] + best_score_at_start
                # Если мы нашли лучшую сегментацию, заканчивающуюся на end_idx, мы обновляем
                if (
                    best_segmentations[end_idx]["score"] is None
                    or best_segmentations[end_idx]["score"] > score
                ):
                    best_segmentations[end_idx] = {"start": start_idx, "score": score}

    segmentation = best_segmentations[-1]
    if segmentation["score"] is None:
        # Мы не нашли токенизацию слова -> возвращаем unknown
        return ["<unk>"], None

    score = segmentation["score"]
    start = segmentation["start"]
    end = len(word)
    tokens = []
    while start != 0:
        tokens.insert(0, word[start:end])
        next_start = best_segmentations[start]["start"]
        end = start
        start = next_start
    tokens.insert(0, word[start:end])
    return tokens, score
```

Мы уже можем опробовать нашу первоначальную модель на некоторых словах:

```
print(encode_word("Hopefully", model))
print(encode_word("This", model))
```

```
(['H', 'o', 'p', 'e', 'f', 'u', 'll', 'y'], 41.5157494601402)
(['This'], 6.288267030694535)
```

Теперь легко вычислить потери модели на корпусе!

```
def compute_loss(model):
    loss = 0
    for word, freq in word_freqs.items():
        _, word_loss = encode_word(word, model)
        loss += freq * word_loss
    return loss
```

Мы можем проверить его работу на имеющейся у нас модели:

```
compute_loss(model)
```

```
413.10377642940875
```

Вычисление оценок для каждого токена также не представляет особой сложности; нам просто нужно вычислить потери для модели, полученные при удалении каждого токена:

```
import copy


def compute_scores(model):
    scores = {}
    model_loss = compute_loss(model)
    for token, score in model.items():
        # Мы всегда храним токены длиной 1
        if len(token) == 1:
            continue
        model_without_token = copy.deepcopy(model)
        _ = model_without_token.pop(token)
        scores[token] = compute_loss(model_without_token) - model_loss
    return scores
```

Мы можем попробовать это на заданном токене:

```
scores = compute_scores(model)
print(scores["ll"])
print(scores["his"])
```

Поскольку `"ll"` используется в токенизации слова `"Hopefully"`, и его удаление, вероятно, заставит нас дважды использовать токен `"l"` вместо этого, мы ожидаем, что он будет иметь положительную потерю. `"his"` используется только внутри слова `"This"`, которое токенизируется само по себе, поэтому мы ожидаем, что потери будут нулевыми. Вот результаты:

```
6.376412403623874
0.0
```

> 💡 Такой подход очень неэффективен, поэтому SentencePiece использует приближенную оценку потерь модели без токена X: вместо того чтобы начинать с нуля, он просто заменяет токен X его сегментацией в оставшемся словаре. Таким образом, все оценки могут быть вычислены одновременно с потерями модели.

Когда этот процесс завершиться, останется только добавить в словарь специальные токены, используемые моделью, а затем итерироваться, пока мы не вычеркнем из словаря достаточно токенов, чтобы достичь желаемого размера:

```
percent_to_remove = 0.1
while len(model) > 100:
    scores = compute_scores(model)
    sorted_scores = sorted(scores.items(), key=lambda x: x[1])
    # Удалите токены percent_to_remove с наименьшими оценками
    for i in range(int(len(model) * percent_to_remove)):
        _ = token_freqs.pop(sorted_scores[i][0])

    total_sum = sum([freq for token, freq in token_freqs.items()])
    model = {token: -log(freq / total_sum) for token, freq in token_freqs.items()}
```

Затем, чтобы токенизировать некоторый текст, нам просто нужно применить предварительную токенизацию, а затем использовать нашу функцию `encode_word()`:

```
def tokenize(text, model):
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in words_with_offsets]
    encoded_words = [encode_word(word, model)[0] for word in pre_tokenized_text]
    return sum(encoded_words, [])


tokenize("This is the Hugging Face course.", model)
```

```
['_This', '_is', '_the', '_Hugging', '_Face', '_', 'c', 'ou', 'r', 's', 'e', '.']
```

Вот и все об Unigram! Надеемся, теперь вы чувствуете себя экспертом во всем, что касается токенизаторов. В следующем разделе мы рассмотрим блоки библиотеки 🤗 Tokenizers и покажем, как их можно использовать для создания собственного токенизатора.

<> Update on GitHub