**TokenIzacja**

Ta część wprowadza w różne typy tokenizacji służące do dzielenia słów na jeden lub więcej tokenów, ponieważ istnieje relacja jeden do wielu między słowami a tokenizacją. W pierwszej sekcji omówiono pretokenizację, koncepcję słowa oraz różnicę między pretokenizacją a tokenizacją. Ta sekcja zawiera również przykład kodu Pythona dotyczący pretokenizacji. W drugiej sekcji omówiono tokenizację opartą na słowach, znakach i podsłowach, a także kompromisy tych technik tokenizacji. Dowiesz się również o tokenizerze dla BERT i tokenizerze dla GPT-2, mimo że zostaną one przedstawione później , dzięki czemu zobaczysz, że ich tokenizery działają inaczej. Zobaczysz również przykłady kodu oparte na Pythonie, które pokazują rzeczywistą tokenizację generowaną przez te modele językowe. Zwróć uwagę, że przykłady kodu oparte na Pythonie w tym rozdziale wymagają obecnie Pythona 3.7, który możesz pobrać z Internetu, jeśli jeszcze go nie masz. W momencie oddania tej książki do druku możliwe będzie korzystanie z Pythona 3.9 (lub nowszego) do przykładów kodu zawartych tu.

**CZYM JEST PRETOKENIZACJA?**

Pretokenizacja to pierwsze z kilku zadań wykonywanych podczas przetwarzania korpusu tekstowego, który zazwyczaj obejmuje pewien podzbiór poniższych:

• Pretokenizacja

• Zadania tokenizacji

• Tokenizacja oparta na słowach

• Tokenizacja oparta na znakach

• Tokenizacja podsłów

Zanim zagłębimy się w temat pretokenizacji, w następnej sekcji omówimycoś fundamentalnego: definicję słowa.

**Czym jest słowo?**

Słowo to zasadniczo sekwencja symboli, które reprezentują coś, co może być namacalne lub abstrakcyjne (to jest to, co zawsze wiedzieliśmy o słowie). Słowo zawierające wyłącznie znaki alfabetu jest konsekwentnie traktowane jako słowo przez różne tokenizatory. Jednak tokenizatory mogą traktować skróty w języku angielskim inaczej, zwłaszcza te, które tokenizują na podstawie spacji. Na przykład słowo „I've" jest skrótem od „I" i „have" i jest traktowane jako jedno słowo przez tokenizatory, które dzielą na podstawie spacji, podczas gdy tokenizatory używające spacji i znaków interpunkcyjnych traktują „I've" jako dwa słowa. Dostępne są następujące cztery metody mapowania między tokenami, znakami i słowami:

•word_to_chars()

•token_to_chars()

•char_to_word()

•char_to_token()

Pretokenizacja a tokenizacja

Dokumentacja książki „Hugging Face" rozróżnia pretokenizację i tokenizację, podczas gdy wiele artykułów online używa jedynie terminu „tokenizacja". Pretokenizacja może być wdrażana na różne sposoby, indywidualnie lub w połączeniu, a tutaj przedstawiono trzy techniki:

•tokenizacja spacji (np. RoBERTa i GPT-2)

•tokenizacja spacji i znaków interpunkcyjnych

•tokenizacja oparta na regułach (np. FlauBERT i XLM)

Użycie spacji jako separatora słów (lub spacji i znaku interpunkcyjnego) w pretokenizacji pokrywa się ze „zwykłą" tokenizacją, którą można przeprowadzić w języku angielskim i większości języków europejskich. Jednak niektóre języki europejskie (np. francuski i niemiecki) używają znaków diakrytycznych, a ich usunięcie może zmienić znaczenie słowa, jak omówiono w dalszej części. W dalszej części w odniesieniu do studiów magisterskich (LLM) pojawią się terminy „uncased" i „cased". W przypadku LLM bez wielkich liter wszystkie słowa są pisane małymi literami, a wszystkie znaki diakrytyczne zostały usunięte. Znaki diakrytyczne występują w kilku językach indoeuropejskich, takich jak francuski, hiszpański, niemiecki i skandynawskie, ale nie w języku angielskim. Jednak w przypadku LLM z wielkimi literami zachowane są wielkie litery i znaki diakrytyczne (jeśli występują). Należy pamiętać, że w niektórych językach, takich jak mandaryński i japoński, nie ma koncepcji wielkich i małych liter. Wybór między LLM bez wielkich liter a LLM z wielkimi literami zależy bezpośrednio od konkretnego przypadku użycia. Język angielski jest prawdopodobnie najłatwiejszym językiem do pretokenizacji ze względu na następujące cechy:

•Brak znaków diakrytycznych

•Wszystkie słowa są rozdzielone spacją

•Brak deklinacji rodzajników i przymiotników (w przeciwieństwie do języków niemieckich/słowiańskich)

•Jedna forma dla każdego słowa w każdym przypadku gramatycznym (np. mianownik i biernik)

Co więcej, apostrof w angielskim słowie może odnosić się do posiadania lub skrótu z czasownikiem (który może być nieregularny), jak pokazano w poniższych zdaniach:

•Tak, to z pewnością prawda.

•Myślę, że taka jest jego prawdziwa natura.

•Tak, to prawda, że jego cena jest wysoka.

•Samochód Johna to kabriolet, jest nowy i ma czerwony dach.

•Dave kupuje nowy samochód.

•Sara nie będzie jadła krewetek.

•Sara jeszcze nie skończyła.

Zauważ, że słowo „its" odnosi się do posiadania, mimo że nie zawiera apostrofu, i stanowi jedyny wyjątek od reguły dotyczącej posiadania. Język angielski również może traktować rzeczowniki jako czasowniki w rozmowach. Gdy kontekst jest jasny, ludzie rozumieją ich znaczenie, nawet gdy słyszą taką konstrukcję po raz pierwszy, jak w poniższym zdaniu:

„I need to book for an hour and then I can meet up you".

Słowo „book” w poprzednim zdaniu to rzeczownik używany jako zamiennik czasownika „study”. W standardowym języku angielskim czasowniki w formie bezokolicznika są inicjowane przez przyimek „to” poprzedzający czasownik. Jednak poprzednie zdanie jest niestandardowe, ponieważ zawiera „to book”, a mimo to ludzie mogą łatwo wywnioskować zamierzone znaczenie. Przyjrzyjmy się teraz przykładowi kodu w Pythonie, który ilustruje pretokenizację, omówioną w następnej sekcji.

**Przykład kodu przed tokenizacją**

Listing 1 przedstawia zawartość pliku pretokenizer1.py, który pokazuje, jak tokenizatory realizują tokenizację dla różnych modeli.

```
from transformers import AutoTokenizer

# notice the multiple white spaces:
sentence  = "I love Chicago deep dish       pizza
don't you?"

# instantiate a BERT-based tokenizer:
tokenizer = AutoTokenizer.from_pretrained
("bert-base-uncased")

# invoke the pre-tokenizer:
result1 = tokenizer.backend_tokenizer.pre_
tokenizer.pre_tokenize_str(sentence)
print("-> tokenizer: BERT")
print("sentence:  ",sentence)
print("tokenized: ",result1)
print()

# instantiate a GPT2-based tokenizer:
tokenizer = AutoTokenizer.from_pretrained("gpt2")
result2 = tokenizer.backend_tokenizer.pre_
tokenizer.pre_tokenize_str(sentence)

print("-> tokenizer: GPT2")
print("sentence:  ",sentence)
print("tokenized: ",result2)
print()


# instantiate a T5-based tokenizer:
tokenizer = AutoTokenizer.from_pretrained("t5-
small")
result3 = tokenizer.backend_tokenizer.pre_
tokenizer.pre_tokenize_str(sentence)

# instantiate a XLNetTokenizer-based tokenizer:
from transformers import XLNetTokenizer

tokenizer = XLNetTokenizer.from_pretrained
("xlnet-base-cased")
result4 = tokenizer.tokenize(sentence)

print("-> tokenizer: XLNetTokenizer")
print("sentence:  ",sentence)
print("tokenized: ",result4)
```

Listing 1 rozpoczyna się od instrukcji importu oraz inicjalizacji zdania zmiennej ciągiem tekstowym. Pozostała część Listingu 2.1 składa się z czterech bloków kodu tokenizacji dla modeli BERT, GPT-2, T5 i

XLNetTokenizer. Każdy blok kodu inicjuje zmienną tokenizer jako instancję tokenizatora specyficznego dla modelu, tokenizuje tekst w zmiennej zdania, a następnie wyświetla wyniki.Na przykład, poniższy blok kodu ilustruje szczegóły poprzedniego zdania dla modelu BERT:

```
# instantiate a BERT-based tokenizer:
tokenizer = AutoTokenizer.from_pretrained
("bert-base-uncased")

# invoke the pre-tokenizer:
result1 = tokenizer.backend_tokenizer.pre_
tokenizer.pre_tokenize_str(sentence)
print("-> tokenizer: BERT")
print("sentence:  ",sentence)
print("tokenized: ",result1)
print()
```

Teraz uruchom kod z Listingu 1, a zobaczysz następujący wynik:

```
-> tokenizer: BERT
sentence:   I love Chicago deep dish       pizza,
don't you?
tokenized:  [('I', (0, 1)), ('love', (2, 6)),
('Chicago', (7, 14)), ('deep', (15, 19)), ('dish',
(20, 24)), ('pizza', (31, 36)), (',', (36, 37)),
('don', (38, 41)), ("'", (41, 42)), ('t', (42,
43)), ('you', (44, 47)), ('?', (47, 48))]

-> tokenizer: GPT2
sentence:   I love Chicago deep dish       pizza,
don't you?
tokenized:  [('I', (0, 1)), ('Ġlove', (1, 6)),
('ĠChicago', (6, 14)), ('Ġdeep', (14, 19)),
('Ġdish', (19, 24)), ('ĠĠĠĠĠĠ', (24, 30)),
('Ġpizza', (30, 36)), (',', (36, 37)), ('Ġdon',
(37, 41)), ("'t", (41, 43)), ('Ġyou', (43, 47)),
('?', (47, 48))]

-> tokenizer: T5-SMALL
sentence:   I love Chicago deep dish       pizza,
don't you?
tokenized:  [('_I', (0, 1)), ('_love', (2, 6)),
('_Chicago', (7, 14)), ('_deep', (15, 19)),
('_dish', (20, 24)), ('_pizza,', (31, 37)),
("_don't", (38, 43)), ('_you?', (44, 48))]

-> tokenizer: XLNetTokenizer
sentence:   I love Chicago deep dish       pizza,
don't you?
tokenized:  ['_I', '_love', '_Chicago', '_deep',
'_dish', '_pizza', ',', '_don', "'", 't', '_you',
'?']
```

**CZYM JEST TOKENIZACJA?**

W uproszczeniu, tokenizacja polega na podziale tekstu korpusu na słowa, które można odwzorować na liczby (np. word2vec), w celu przetworzenia tych liczbowych odpowiedników tych słów przez sieci neuronowe. Istnieje kilka rodzajów technik tokenizacji, które zostaną pokrótce omówione w dalszej części, wraz z niektórymi kompromisami związanymi z tymi technikami.

**Niuanse tokenizerów**

Tokenizery wykonują różne rodzaje tokenizacji, a ich charakterystyka zależy od modelu przetwarzania języka naturalnego (NLP) używanego do tworzenia instancji tokenizera. W związku z tym, w tej części pokrótce omówiono różnice między tokenizerami tworzonymi na podstawie modeli BERT, GPT2 i T5. Tokenizer oparty na BERT przechowuje indeks przesunięć słów. Ponadto, wstępna tokenizacja dzieli na odstępy i interpunkcję. Jednak tokenizer oparty na GPT2 dzieli na odstępy i znaki interpunkcyjne, a spacje zastępuje symbolem Ġ, dzięki czemu możliwe jest odzyskanie oryginalnych spacji w ciągu tekstowym. Ten tokenizer różni się od tokenizera BERT, ponieważ rozpoznaje podwójne spacje. Innym typem tokenizera jest tokenizer oparty na T5, który opiera się na algorytmie SentencePiece i jest również podobny do tokenizera GPT2-2. Jednak ten tokenizer zachowuje spacje i zastępuje je tokenem podkreślenia (_). Co więcej, tokenizer T5 dzieli tylko na odstępy, a nie na znaki interpunkcyjne, i dodaje również początkową spację. W rozdziale 4 przedstawiono przykłady modeli językowych opartych na Pythonie, które obejmują tworzenie instancji tokenizera specyficznego dla danego modelu.

**Ogólny algorytm mapowania tokenów na indeksy**

W tej sekcji przedstawiono prosty algorytm kojarzenia liczby całkowitej dla każdego słowa w korpusie, pokazany na Listingu 2.

```
token2idx = {}
curr_idx  = 0

docs = list()
docs.append("this is a simple document")
docs.append("this is a second document")
docs.append("the third short document")

# initialize word/integer pairs:
for doc in docs:


   for token in doc.split(" "):
     if token not in token2idx:
       token2idx[token] = curr_idx
       curr_idx += 1

 # display token/integer pairs:
 for token in token2idx.keys():
   print(f"token: {token:10} value:
 {token2idx[token]:4}")
 print()

 # create an integer-based counterpart
 int_docs = []
 for doc in docs:
   for doc in docs:
     tokens = [token2idx[word] for word in doc.split
 (" ")]
     int_docs.append(tokens)

 print("int_docs:",int_docs)
 print()

 # create integer-to-token mapping:
 idx2token = {v: k for k,v in token2idx.items()}
 for k,v in idx2token.items():
   print(f"key: {k:4} value: {v:10}")
```

Listing 2 rozpoczyna się od zdefiniowania słownika Pythona token2idx i zmiennej skalarnej curr_idx, która początkowo ma wartość 0. Kolejny blok kodu inicjuje zmienną docs trzema dokumentami, z których każdy jest krótkim zdaniem. Kolejna część Listingu 2 iteruje po dokumentach w docs i używa funkcji split() do tokenizacji każdego dokumentu. Następnie logika warunkowa dodaje nową parę token/liczba całkowita do słownika token2idx, jeśli bieżący token nie znajduje się jeszcze w tym słowniku. Kolejny blok kodu wyświetla pary token/liczba całkowita w token2idx, a następnie blok kodu wypełnia zmienną int_docs wartościami całkowitymi powiązanymi z każdym tokenem w słowniku token2idx, a następnie wyświetla wszystkie wartości całkowite. Ostatnia część Listingu 2 tworzy mapowanie liczb całkowitych na tokeny o nazwie idx2token. Ten ostatni blok kodu jest istotny, gdy musimy przekonwertować predykcje (obejmujące wartości identyfikatorów tokenów) w sieciach neuronowych na powiązane tokeny. Uruchom kod z Listingu 2, a zobaczysz następujący wynik:

```
token: this       value:    0
token: is         value:    1
token: a          value:    2
token: simple     value:    3
token: document   value:    4
token: second     value:    5
token: the        value:    6
token: third      value:    7
token: short      value:    8

int_docs: [[0, 1, 2, 3, 4], [0, 1, 2, 5, 4], [6, 7,
8, 4], [0, 1, 2, 3, 4], [0, 1, 2, 5, 4], [6, 7, 8,
4], [0, 1, 2, 3, 4], [0, 1, 2, 5, 4], [6, 7, 8, 4]]

key:    0 value: this
key:    1 value: is
key:    2 value: a
key:    3 value: simple
key:    4 value: document
key:    5 value: second
key:    6 value: the
key:    7 value: third
key:    8 value: short
```

Powyższy przykład kodu ponownie przedstawia ogólny proces dla słów powiązanych z wartościami całkowitymi. Zanim omówimy techniki tokenizacji stosowane w modelach językowych, przyjrzyjmy się bliżej niektórym związanym z nimi wyzwaniom, opisanym w następnej sekcji.

**Zadania tokenizacji i związane z nimi wyzwania**

Tokenizacja polega na wyszukiwaniu tokenów w zdaniach i dokumentach, gdzie tokenami mogą być słowa, znaki lub części słów. Tokenizacja musi również uwzględniać potencjalne problemy, które mogą pojawić się podczas wykonywania następujących podzadań:

- Konwertuj tekst na małe litery (lub nie)
- Przetwarzaj znaki interpunkcyjne oddzielające zdania
- Obsługuj znaki diakrytyczne (takie jak francuski i niemiecki)
- Obsługuj skróty („won't" kontra „will not")
- Przetwarzaj nietypowe (rzadko występujące) słowa

Jeśli nie wykonywałeś wcześniej tokenizacji, może się to wydawać prostym zadaniem, ale istnieje kilka nietrywialnych aspektów tokenizacji. Oto lista punktów (w dowolnej kolejności), które należy uwzględnić podczas tokenizacji korpusu:

• Rzeczowniki pospolite a nazwy własne

• Opcjonalny separator słów

• Znaki diakrytyczne i znaczenia słów

• Różne techniki tokenizacji podwyrazów

• Rzeczowniki w liczbie pojedynczej a mnogiej

• Warianty pisowni słów

• Błędy typograficzne

• Fałszywe wyrazy pokrewne w zadaniach tłumaczeniowych

Po pierwsze, zamiana słów na małe litery powoduje utratę rozróżnienia między rzeczownikami pospolitymi a własnymi (np. „kamień" a „kamień"). Po drugie, użycie spacji do tokenizacji zdań na tokeny słów nie działa poprawnie w językach takich jak japoński, w których spacje są opcjonalne. W konsekwencji zdanie można zapisać jako pojedynczy, ciągły ciąg znaków innych niż spacja: Może to być uciążliwe i żmudne w każdym języku. Ten przykład pokazuje, że brak spacji zwiększa złożoność (i być może trzeba się było skupić, aby przeanalizować poprzedni ciąg bez spacji). Po trzecie, pominięcie znaków diakrytycznych (akcentów) może sprawić, że słowo stanie się wieloznaczne: słowo „peche" ma trzy możliwe znaczenia (tj. brzoskwinia, ryba lub grzech) w zależności od tego, który znak diakrytyczny występuje w słowie. Jako skrajny przykład, poniższe (rzadkie) francuskie wyrażenie „le peche du peche" ma wiele interpretacji, które byłyby jasne jedynie z kontekstu. Po czwarte, ważne jest, aby podczas tokenizacji uwzględnić interpunkcję, dzięki czemu modele nie będą musiały uczyć się różnych reprezentacji słowa w połączeniu z wieloma znakami interpunkcyjnymi. Po piąte, różne biblioteki i zestawy narzędzi realizują różne typy tokenizacji podwyrazów. Na przykład możliwe jest tokenizowanie słowa „Don't" jako [„Don", „'", „t"] lub jako [„Do", „n't"] (w tym przypadku biblioteka spaCy stosuje to drugie podejście). Po szóste, usunięcie końcowego „s" z angielskiego słowa może skutkować słowem o innym znaczeniu. Na przykład „birds" i „bird" to rzeczownik w liczbie mnogiej i pojedynczej, podczas gdy „new" i „news" mają różne znaczenia. Ogólnie rzecz biorąc, dodanie litery „s" do rzeczownika tworzy liczbę mnogą tego rzeczownika, podczas gdy dodanie „s" do przymiotnika może skutkować rzeczownikiem o innym znaczeniu niż przymiotnik. Chociaż niektóre z powyższych przykładów mogą wydawać się nieistotne (lub wręcz trywialne), takie szczegóły mogą wpływać na dokładność, a także płynność tekstu przetłumaczonego na język inny niż język tekstu wejściowego. Tokenizacja musi również uwzględniać inne kwestie, takie jak obsługa alternatywnych sposobów zapisu słowa (np. „favor" zamiast „favour", „tire" zamiast „tire" i „color" zamiast „color"), różne znaczenia tego samego słowa (np. „to table" a discussion) oraz błędy typograficzne (np. „dependent" zamiast niepoprawnego „dependant"). Inne potencjalne problemy obejmują warianty pisowni i kapitalizacji, czasowniki nieregularne oraz problemy z tokenizacją poza słownikiem. Istnieje jeszcze jeden problem związany z różnicami w wymowie. Rozważ różne sposoby wymowy następujących słów:

•Aluminium: „uh-LOO-minum" lub „al-loo-MIN-ium"

•Prywatność: „PRIV-acy" kontra „PRY-vacy"

•Harmonogram: „SKEDjule" kontra „SHEDjule"

**Alternatywa dla tokenizacji: model ByT5**

Google wprowadził model ByT5, który działa bezpośrednio z bajtami UTF-8, zamiast korzystać z tokenizacji podsłownej. Zaletą tego podejścia jest to, że eliminuje ono jakąkolwiek formę wstępnego przetwarzania tekstu. Co więcej, podejście to można rozszerzyć na obsługę sekwencji bajtów bez generowania nadmiernych kosztów obliczeniowych. Sekwencje bajtów są jednak znacznie dłuższe niż sekwencje na poziomie słów. Model ByT5 wywodzi się z innego modelu Google o nazwie mT5, co jest akronimem od „Massively Multilingual Text-to-Text Transfer Transformer" (masowo wielojęzyczny transformator transferu tekstu na tekst). Chociaż mT5 jest modelem opartym na tokenach, można go uczynić wolnym od tokenów, wprowadzając do niego niewielki zestaw modyfikacji. Szczegóły dotyczące modelu ByT5 można znaleźć w artykule arxiv „Towards a Token-Free Future With Pre-Trained Byte-to-Byte Models", dostępnym pod adresem https://arxiv.org/abs/2105.13626.

**TOKENIZERY SŁÓW, ZNAKÓW I PODSŁÓW**

W istocie tokenizery przekształcają zdania i dokumenty do formatu odpowiedniego do przetwarzania przez modele językowe. Należy pamiętać, że LLM przetwarzają wyłącznie wartości liczbowe, co oznacza, że tokenizery muszą konwertować ciągi tekstowe na dane liczbowe. Zasadniczo cel tokenizera jest dwojaki: znalezienie najbardziej znaczącej reprezentacji oraz (jeśli to możliwe) znalezienie reprezentacji najmniejszej.

W NLP można spotkać kilka typów tokenizerów:

•tokenizery przedsłowne (omówione wcześniej)

•tokenizery słów

•tokenizery znaków

•tokenizery podsłowne

Każdy z powyższych typów tokenizerów zostanie omówiony w poniższych podrozdziałach wraz z niektórymi ich wadami i zaletami.

**Tokenizery oparte na słowach**

Ogólnie rzecz biorąc, tokenizery oparte na słowach są proste w użyciu, ponieważ obejmują ograniczoną liczbę reguł i mogą osiągać dość dobre rezultaty. Jednym ze sposobów programowej tokenizacji ciągu tekstowego na podstawie spacji jest użycie funkcji split() w Pythonie, jak pokazano tutaj:

```
tokenized_text = "I love Chicago pizza".split()

print(tokenized_text)
```

```
['I', 'love', 'Chicago', 'pizza']
```

Niektóre tokenizery słów określają dodatkowe reguły interpunkcji, co może skutkować rozbudowanym słownictwem (tj. liczbą tokenów w danym korpusie). Ponadto słowom przypisuje się identyfikator z zakresu od 0 do (N-1), gdzie N to liczba tokenów w słowniku. Model identyfikuje dane słowo poprzez przypisaną mu wartość identyfikatora. Niestety, blisko spokrewnione słowa są traktowane jako różne słowa, co oznacza, że będą im przypisywane różne wartości identyfikatora. Na przykład, poniższy zestaw słów jest powiązany znaczeniowo:

•sing

•sang

•sung

•singing

Co więcej, model będzie traktował formę liczby pojedynczej rzeczownika inaczej niż jego formę liczby mnogiej. Proces ten jest dodatkowo skomplikowany w językach (takich jak niemiecki i słowiańskie), w których rzeczowniki występują w rodzaju męskim, żeńskim, nijakim i mnogim, a wszystkie z nich traktowane są jako różne. Na przykład, w języku angielskim istnieje tylko jedna forma rodzajnika „the", podczas gdy poniższy przykład przedstawia deklinację rodzajnika określonego „der" w języku niemieckim, z kolumnami dla rodzaju męskiego, żeńskiego, nijakiego i mnogiego oraz wierszami dla mianownika, dopełniacza, celownika i biernika:

| | Masculine | Feminine | Neuter | Plural |
|---|---|---|---|---|
| Nominative | der | die | das | die |
| Genitive | des | der | des | der |
| Dative | dem | der | dem | den |
| Accusative | den | die | das | die |

Jak zobaczysz w rozdziale 5, BERT zawiera około 25 000 tokenów, które są słowami z języka angielskiego, składającego się z ponad pół miliona słów. BERT wykorzystuje mechanizm podziału tokenów na podtokeny, co umożliwia mu wykorzystanie tych „fragmentów słów" dla innych tokenów. Co więcej, BERT używa specjalnego tokenu UNK do reprezentowania nieznanych słów (i inne modele językowe również używają tokenu UNK).

**Ograniczenia tokenizatorów słów**

Model rozpoznaje tylko tokeny słów, które są częścią jego kroku uczenia, co oznacza, że słowa złożone nie zostaną rozpoznane, jeśli nie pojawią się w kroku uczenia. Na przykład, jeśli słowa „book" i „keeper" są częścią kroku uczenia, ale słowo „bookkeeper" nie było w kroku uczenia, to „bookkeeper" nie zostanie rozpoznane i w związku z tym będzie reprezentowane za pomocą tokenu UNK. Kolejnym wyzwaniem dla tokenizatorów są skróty. Na przykład angielskie słowa „its" i „it's" mają zupełnie inne znaczenie: „its" odnosi się do posiadania, podczas gdy „it's" oznacza „it is". W przypadku rzeczowników apostrof również odnosi się do posiadania (np. samochód Johna). Zatem zdanie „To prawda, że długopis Johna jest wart swojej wagi w złocie" może mieć tylko jedną interpretację. Jak zapewne się domyślasz, akronimy w połączeniu ze slangiem stanowią wyzwanie dla tokenizatorów. Rozważmy następujący zestaw terminów, których niektórzy ludzie używają w swoich wiadomościach tekstowych:

AFAIK

IIRC

IDK

SMH

AFAICS

LMK

DOPE

BOMB

Tokenizacja dla języków z wieloma alfabetami

Jak już wcześniej zauważyliście, znak spacji między wyrazami jest opcjonalny w językach takich jak chiński i japoński. Ponadto język japoński ma trzy alfabety: hiragana, katakana (używany tylko do słów obcych) i romaji (do romanizacji słów japońskich). Japoński ma również kanji, czyli system reprezentacji wyrazów oparty na piktogramach. Japońskie zdania i znaki drogowe mogą zawierać kombinację hiragany i kanji, co jest jeszcze trudniejsze do zrozumienia dla osób nieznających języka japońskiego. Jednym z pomysłów jest reprezentacja wyrazów za pomocą systemu kodowania. Na przykład ASCII to schemat kodowania, który koduje wyrazy w języku angielskim; jednak ASCII nie może reprezentować języków europejskich (takich jak francuski) ze znakami diakrytycznymi. Innym schematem kodowania jest ISO-159, który obsługuje znaki diakrytyczne; jednak ten schemat kodowania jest ograniczony głównie do języków zachodnioeuropejskich. Innym schematem kodowania jest UTF-8, który obsługuje znacznie szerszy zestaw znaków. Prawdopodobnie najbardziej rozbudowanym schematem kodowania jest schemat kodowania Unicode, który obsługuje wszystkie litery, znaki interpunkcyjne i alfabety wszystkich języków ludzkich, w tym arabskiego, mandaryńskiego i japońskiego. Wyszukaj w internecie więcej informacji na temat schematów kodowania, a znajdziesz wiele darmowych artykułów.

**KOMPROMISY Z TOKENIZERAMI ZNAKOWYMI**

Tokenizatory znakowe dzielą korpus na znaki zamiast na pojedyncze słowa, co ma dwie główne zalety:

• Słownictwo jest mniejsze niż w przypadku tokenizacji opartej na słowach.

• Mniejsza liczba tokenów pozasłownych (każde słowo można zbudować ze znaków).

Istnieją jednak również dwa ograniczenia tokenizacji opartej na znakach. Po pierwsze, zestaw znaków zapewnia ograniczone znaczenie rzeczowe. Słowa są podstawowymi elementami składowymi zdań i dlatego niosą ze sobą znaczenie (a czasami wieloznaczne). Chociaż powyższe stwierdzenie jest prawdziwe w przypadku języków opartych na alfabecie (takich jak języki indoeuropejskie), niektóre języki Azji Południowo-Wschodniej opierają się na piktogramach, które mogą przekazywać złożone koncepcje wymagające zdania wyjaśniającego. Po drugie, tokenizacja oparta na znakach generuje znacznie większą liczbę tokenów, która może być nawet pięciokrotnie większa niż zbiór tokenów opartych na słowach, co skutkuje znacznie dłuższym czasem przetwarzania podczas trenowania modelu w porównaniu z czasem przetwarzania tokenów opartych na słowach. Jeśli porównamy właściwości znaków z właściwościami słów, zobaczymy, że znaki nie mają znaczenia rzeczowego dla wielu języków (z wyłączeniem języków azjatyckich). Jako bardzo prosty przykład, litera „s” jest niejednoznaczna ze względu na brak kontekstu, podczas gdy kontekst słowa „jabłka” jest zasadniczo jednoznaczny. Słowa mają jednak odwrotną właściwość: słowo może być przeciążone wieloma znaczeniami, które zależą od ich kontekstu w zdaniach.

**Ograniczenia tokenizatorów znakowych**

Jak dowiedzieliście się w poprzedniej sekcji, tokeny znakowe wymagają więcej czasu obliczeniowego i zasobów do trenowania modelu. Jeśli średnia liczba liter w słowie wynosi 5, to zdanie 10-wyrazowe zawiera 50 tokenów zamiast 10. Dodatkowo pojawia się problem generowania poprawnej pisowni słowa, co jest trudniejsze, gdy tokeny są znakami, a nie słowami. Co więcej, tokenizacja oparta na słowach lepiej nadaje się do różnych zadań związanych z przetwarzaniem języka naturalnego, takich jak NER i klasyfikacja tekstu, a mniej do tokenizacji opartej na znakach. Chociaż poprzednie punkty przemawiają za tokenizacją opartą na słowach, wiąże się to z utworzeniem słownictwa o umiarkowanej objętości, co z kolei zwiększa prawdopodobieństwo występowania słów OOV (poza słownikiem) w korpusie. Jak zobaczycie w przykładach kodu opartego na BERT, słowa OOV są reprezentowane za pomocą tokenu UNK („nieznany”). Na szczęście istnieje trzecia, hybrydowa technika, która pozwala

wykorzystać zalety tokenizacji opartej na znakach i słowach, unikając jednocześnie ich problemów. Technika ta nazywa się tokenizacją podsłowną i jest tematem następnej sekcji.

**TOKENIZACJA PODWYRAZÓW**

Tokenizacja podwyrazów zazwyczaj opiera się na algorytmach, regułach statystycznych i ważnej heurystyce: tokenizacja rzadko występujących lub rzadko występujących słów w podwyrazy i brak podziału na słowa często występujące. Taką tokenizację można łatwo przeprowadzić dla angielskich przysłówków z przyrostkiem „ly": zastąp przysłówek dwoma tokenami, gdzie drugim tokenem jest kombinacja „ly". W ten sposób „slowly" dzieli się na „slow" i „ly", a „quickly" zastępuje się „quick" i „ly" itd. Podobnie, przymiotnik „lonely" można rozdzielić na „lone" i „ly". W niektórych przypadkach ten rozkład na dwa tokeny daje faktyczne słowo dla pierwszego tokenu, co ma miejsce w powyższych przykładach. Co więcej, tokenizacja podwyrazów może również prowadzić do tokenów o znaczeniu, na przykład tokenizacja słowa „internationalization" na „international" i „ization". Oprócz powyższego typu tokenizacji, służącego do obsługi podzbioru słów angielskich, istnieją inne algorytmy tokenizacji, z których niektóre wymieniono tutaj:

• BPE na poziomie bajtów (w GPT-2)

• WordPiece (w BERT)

• SentencePiece lub Unigram (w kilku modelach wielojęzycznych)

**Przykład kodu tokenizatora podsłownego**

Listing 3 przedstawia zawartość pliku hf_subword_tokenizer.py, która ilustruje sposób przeprowadzania tokenizacji za pomocą transformatora Hugging Face.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained
("bert-base-cased")
sequence = "I love Chicago always deeply dished
pizzas!"
tokens = tokenizer.tokenize(sequence)

print("sequence:",sequence)
print("tokens:  ",tokens)

ids = tokenizer.convert_tokens_to_ids(tokens)
print("token ids:", ids)

# ids = [146, 1567, 2290, 1579, 5585, 10478, 1174,
13473, 1116, 106]
decoded_ids = tokenizer.decode(ids)
print("decoded:  ",decoded_ids)
```

Listing 3 rozpoczyna się od instrukcji import, a następnie inicjuje zmienną tokenizer jako instancję AutoTokenizer. Następnie sekwencja zmiennych jest inicjowana jako ciąg tekstowy, a zmienna tokens jest inicjowana jako zestaw tokenów w sekwencji zmiennych. Uruchom kod z Listingu 2.3 za pomocą następującego polecenia:

python3 hf_tokenizer.py

Poprzednie polecenie wyświetli następujące dane wyjściowe:

```
sequence: I love Chicago always deeply dished
pizzas!
tokens:   ['I', 'love', 'Chicago', 'always',
'deeply', 'dish', '##ed', 'pizza', '##s', '!']
token ids: [146, 1567, 2290, 1579, 5585, 10478,
1174, 13473, 1116, 106]
decoded:   I love Chicago always deeply dished
pizzas!
```

Należy zauważyć, że metoda decode() z Listingu 3 wykonuje dwie czynności: po pierwsze, indeksy są konwertowane na tokeny; po drugie, tokeny, które były częścią tych samych słów, są grupowane, tworząc w ten sposób czytelne zdanie. To zachowanie jest niezwykle przydatne w przypadku modeli przewidujących nowy tekst (tekst generowany z monitu lub w przypadku problemów z sekwencją do sekwencji, takich jak tłumaczenie lub podsumowanie).

**Kluczowe kwestie dotyczące tokenizacji BERT**

Chociaż BERT jest omówiony w rozdziale 5, ta sekcja została tutaj uwzględniona ze względu na informacje dotyczące tokenów, które ten rozdział zawiera na temat BERT. Niezależnie od tego, czy zdecydujesz się przeczytać tę sekcję, czy nie, zawsze możesz do niej powrócić po zapoznaniu się z materiałem dotyczącym kodowania par bajtów (BPE) i WordPiece w rozdziale 5. BERT ma stały słownik tokenów wraz z powiązanymi osadzeniami, co oznacza, że BERT udostępnia własny tokenizer. Co więcej, nie można zastąpić osadzeń BERT innym zestawem osadzeń (swoim lub cudzym). Inne kluczowe punkty dotyczące BERT to:

•obejmuje BPE i WordPiece (podobnie jak BPE)

•zawiera warstwę obliczeń prawdopodobieństwa

•modele oparte na BERT wykorzystują te same tokeny specjalne

•zasadniczo trenowane na surowym tekście za pomocą WordPiece

•nie wymaga stemmingu i lematyzacji

•model małych liter konwertuje zdania na małe litery

•model wielkich liter nie wprowadza zmian w kapitalizacji zdań

•wykorzystuje model transformatora

Ponieważ BERT wykorzystuje komponent kodera architektury transformatora, etap dostrajania analizuje tylko słowa, które mają wpływ na dane wyjściowe, ignorując słowa wspólne dla wszystkich danych. Należy pamiętać, że każdy model BERT używa własnego typu danych wejściowych: na przykład, w przeciwieństwie do BERT, DistilBERT nie używa identyfikatorów token_type_id. Posiadasz już wystarczającą wiedzę na temat tokenizatorów, aby poznać interfejsy API Hugging Face, które zostaną omówione w następnej sekcji. Modele takie jak BERT lub GPT-2, a także ich warianty, tokenizują tekst za pomocą jakiejś wersji algorytmu BPE lub modelu unigram.

**ALGORYTMY TOKENIZACJI PODWIERDZI**

Tokenizacja podwyrazów polega na dzieleniu tokenów OOV na mniejsze fragmenty. Przypomnijmy, że poprzednia sekcja pokrótce opisuje następujące typy tokenizatorów:

•tokenizatory słów

•tokenizatory znaków

•tokenizatory podwyrazów

Algorytmy tokenizacji podwyrazów opierają się na heurystyce, co oznacza, że opierają się na intuicyjnym rozumowaniu, które „ma sens” i może dać poprawną odpowiedź. W szczególności słowom, które występują częściej, przypisywane są unikalne identyfikatory. Natomiast słowa o niższej częstotliwości występowania są dzielone na podwyrazy, które zachowują znaczenie słów o niższej częstotliwości występowania. Poniższa lista zawiera cztery ważne algorytmy tokenizacji podwyrazów:

•BPE

•SentencePiece (używany w ALBERT)

•unigramowy model języka

•WordPiece (używany w BERT i DistilBERT)

Biblioteka transformatorów Hugging Face obsługuje wymagane tokenizatory dla większości wariantów BERT. W rzeczywistości model BERT w tej bibliotece obsługuje 11 typów problemów NLP, które można również łączyć, tworząc potok. Wszystkie cztery algorytmy obejmują tokenizację podsłowną tekstu, choć różnią się tym, że implementują różne algorytmy przetwarzania podsłow. Autorzy BERT wytrenowali wariant BERT, wykorzystując technikę zwaną maskowaniem całych słów, która opiera się na następującej zasadzie: jeśli jedno z podsłów z „osadzeń” zostało wybrane, wszystkie tokeny podsłów zostałyby zastąpione.

**Czym jest BPE?**

BPE (kodowanie par bajtów) to wszechstronne narzędzie, które początkowo służyło potrzebom kompresji danych, ale stało się kluczowe we współczesnym przetwarzaniu języka naturalnego (NLP). Umożliwiając tokenizację podsłownictwa, BPE pozwala modelom na wydajne i adaptacyjne przetwarzanie obszernego słownictwa, co czyni je podstawą wielu nowatorskich modeli językowych. BPE to algorytm tokenizacji podsłownictwa oddolnego, który uczy się słownictwa podsłownictwa o określonej wielkości (wielkość słownictwa jest hiperparametrem). Główną ideą BPE jest iteracyjne scalanie najczęściej występującej pary bajtów (lub znaków) w zbiorze danych, aż do osiągnięcia określonej liczby scaleń. Pierwszy krok nazywa się inicjalizacją i polega na tokenizacji tekstu na znaki, a następnie przypisaniu każdemu znakowi unikalnego tokena. Drugi krok polega na iteracyjnym scalaniu wyników. Podczas każdej iteracji należy znaleźć najczęściej występującą parę sąsiadujących tokenów, a następnie scalić tokeny, aby utworzyć nowy token. Powtarzaj ten proces przez predefiniowaną liczbę iteracji lub do momentu, aż scalanie nie będzie już możliwe. Powstałe tokeny reprezentują częste kombinacje znaków lub podsłów w danych. Przyjrzyjmy się słowu „lowers” i zobaczmy, jak zastosować BPE do tego słowa. W oparciu o poprzedni akapit, pierwszy krok polega na tokenizacji słowa na znaki, a zatem początkowe tokeny to l, o, w, e, r i s. Następnie, jeśli „lo” jest najczęściej występującą parą w zbiorze danych, może to być pierwsze scalenie, skutkujące nowym zestawem tokenów: lo, w, e, r i s. Dodatkowe scalenie może skutkować tokenami low, e i rs. Końcowe jednostki podsłów zależą od częstości par w zbiorze danych i liczby wykonanych operacji scalenia. Istnieje pewna wada stosowania częstotliwości w BPE: wynik iteracyjnego etapu scalania może skutkować niejednoznacznymi kodowaniami końcowymi, które mogą być nieprzydatne. Jako drugi przykład rozważmy ciąg znaków aaabdaaabac. Ponieważ para bajtów aa występuje najczęściej, zastąpimy ją znakiem, który nie występuje w ciągu, takim jak litera Z. Wykonaj zamianę, co da następujący ciąg tekstowy:

ZabdZabac (gdzie Z=aa)

Powtórz krok podstawiania, tym razem z parą ab, i zastąp tę parę literą Y, co da ciąg ZYdZYac (gdzie Y=ab Z=aa). W tym momencie możemy kontynuować poprzednią procedurę, wybierając ZY (która pojawia się dwukrotnie) i zastępując ten ciąg literą X, co da XdXac (gdzie X=ZY Y=ab Z=aa).

**Czym jest WordPiece?**

WordPiece to algorytm tokenizacji podwyrazów oparty na zasadzie zachłanności, który został udostępniony wraz z wydaniem BERT w 2018 roku i który łączy w sobie cechy algorytmu BPE (tj. parowania znaków) oraz algorytmu unigram (tj. wybierania par do scalenia). Dokładniej rzecz ujmując, WordPiece używa wartości prawdopodobieństwa zamiast częstotliwości występowania, aby połączyć najlepszą parę znaków, i używa częstotliwości występowania do wyboru znaków do sparowania. Główna różnica dotyczy konkretnego sposobu wybierania bigramów na etapie scalania. Co ciekawe, RoBERTa (oparta na BERT) również wykorzystuje WordPiece. Oto kilka przykładów tokenizacji podsłów w BERT:

1. toppings jest dzielone na topping i ##s.

2. trimmings jest dzielone na trim, ##ming i ##s.

3. misspelled jest dzielone na mis, ##spel i ##led.

Jednak BERT nie zapewnia mechanizmu rekonstrukcji oryginalnego słowa z jego fragmentów. Natomiast ELMo zapewnia reprezentację kontekstową na poziomie słowa (a nie podsłowa), co różni się od BERT. W dalszej części tego rozdziału poznasz przykłady kodu, które tworzą tokeny BERT ze zdań w języku angielskim.

**Czym jest SentencePiece?**

SentencePiece to kolejny tokenizator podwyrazów i detokenizator dla NLP, który wykonuje segmentację podwyrazów, obsługując modele językowe BPE i unigram. SentencePiece oferuje ujednolicone, oparte na danych podejście do tokenizacji i detokenizacji tekstu. Jego zdolność do pracy z surowym tekstem, generowania jednostek podwyrazów i niezależności od języka sprawia, że jest popularnym wyborem do zadań przetwarzania tekstu opartych na sieciach neuronowych, szczególnie w scenariuszach obejmujących wiele języków lub wymagających spójnego przetwarzania danych tekstowych. SentencePiece ma dwie główne cechy. Po pierwsze, SentencePiece traktuje ciąg tekstowy jako sekwencję znaków Unicode i zastępuje spacje znakiem _. Ponadto SentencePiece obsługuje języki, w których spacja między wyrazami jest opcjonalna (takie jak chiński i japoński). Po drugie, SentencePiece obsługuje odwracalną tokenizację, co oznacza, że oryginalny tekst można skonstruować poprzez łączenie słów i zastępowanie wszystkich wystąpień znaku „_” spacją. Z kolei BERT nie obsługuje odwracalnej tokenizacji, ponieważ tokenizer BERT usuwa wiele spacji. Po trzecie, tradycyjne tokenizery są często projektowane dla konkretnych języków, a ich wydajność może ulec pogorszeniu w przypadku zastosowania w innych. Z kolei SentencePiece zapewnia spójne podejście do tokenizacji niezależnie od języka, dzięki czemu nadaje się do modeli wielojęzycznych lub języków z mniejszym wsparciem dla przetwarzania języka naturalnego (NLP). Po czwarte, SentencePiece hermetyzuje tokenizację i detokenizację, oferując kompleksowy przepływ pracy do przetwarzania tekstu, co upraszcza proces dla programistów. Po piąte, SentencePiece może efektywnie obsługiwać rzadkie słowa i terminy spoza słownika, reprezentując je jako kombinację znanych tokenów podsłownych. Ta możliwość jest niezbędna, aby modele mogły dobrze generalizować dane niewidoczne.

**TOKENIZERY I MODELE HUGGING FACE**

Hugging Face udostępnia liczne klasy dla tokenizatorów i modeli. W szczególności klasy Hugging Face zawierające prefiks Auto (np. AutoModel i AutoTokenizer) wykonują różne zadania, takie jak dostęp do konfiguracji i wstępnie wytrenowanych wag modelu, który jest zawarty jako ciąg znaków w metodzie from_pretrained(). Na przykład, poniższy fragment kodu inicjuje zmienną tokenizer jako instancję klasy AutoTokenizer i z modelu bert-base-uncased:

tokenizer = AutoTokenizer.from_pretrained

("bert-base-uncased")

Poniższy fragment kodu inicjuje zmienną model jako instancję modelu bert-base-uncased:

mymodel = TFAutoModel.from_pretrained

("bert-base-uncased")

Dla wygody, poniższa lista zawiera zestaw tokenizerów, do których odwołują się przykłady kodu Pythona w tej książce, wyświetlane jako lista instrukcji importu:

•from transformers import AutoTokenizer

•from transformers import BartTokenizer

•z transformers import BertTokenizer

•z transformers import BertTokenizerFast

•z transformers import FlaubertTokenizer

•z transformers import GPT2Tokenizer

•z transformers import RobertaTokenizer

•# tokenizery dall_e_tok:

•z dall_e_tok import DALLETokenizer

•z dall_e_tok import DallEEncoder

•z dalle_pytorch.simple_tokenizer import tokenizer, HugTokenizer

•# tokenizery pytorch:

•z pytorch_transformers import GPT2Tokenizer, GPT2LMHeadModel

•# tokenizer BPE:

•from tokenizery import ByteLevelBPETokenizer

Poniższa lista zawiera zestaw klas związanych z modelem, do których odwołują się przykłady kodu Pythona , wyświetlanych jako instrukcje importu:

•from transformers import AutoModel

•from transformers import AutoModelForSeq2SeqLM

•from transformers import AutoModelForSequenceClassification

•from transformers import BartForConditionalGeneration

•from transformers import BertForQuestionAnswering

•from transformers import BertForSequenceClassification

•from transformers import BertModel

•from transformers import FlaubertModel

•from transformers import GPT2LMHeadModel

•from transformers import TFAutoModel

•from transformers import TFAutoModelForSequenceClassification

Poniższa lista zawiera klasy związane z konfiguracją, do których odwołują się przykłady kodu Pythona , wyświetlane jako instrukcje import:

•z transformatorów import BertConfig

•z transformatorów import RobertaConfig

Poniższe podsekcje omawiają, jak zapisywać i ładować tokenizery, a także jak tokenizować ciągi znaków za pomocą transformatora Hugging Face.

**Ładowanie i zapisywanie tokenizerów**

Zadanie ładowania i zapisywania tokenizerów jest tak samo proste, jak odpowiadające mu zadanie dla modeli i opiera się na tych samych dwóch metodach: from_pretrained() i save_pretrained(). Metody te ładują lub zapisują algorytm używany przez tokenizer (podobnie jak architektura modelu), a także jego słownik (podobnie jak wagi modelu). Musimy załadować tokenizer BERT, który jest trenowany z tym samym punktem kontrolnym co BERT dla kroku ładowania modelu, z tą różnicą, że używamy klasy BertTokenizer:

from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained

("bert-base-cased")

Podobnie jak AutoModel, klasa AutoTokenizer pobiera odpowiednią klasę tokenizera z biblioteki na podstawie nazwy punktu kontrolnego i może być używana bezpośrednio z dowolnym punktem kontrolnym:

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained

("bert-base-cased")

Możemy teraz użyć tokenizera, jak pokazano w poprzedniej sekcji:

tokenizer("Korzystanie z sieci Transformer jest proste")

{'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]}

Zapisywanie tokenizera jest identyczne z zapisywaniem modelu:

tokenizer.save_pretrained("directoryname")

**AutoTokenizer, BERTTokenizer i GPT2Tokenizer**

Chociaż BERT i GPT są omówione odpowiednio w rozdziałach od 5 do 8, ich klasy tokenizera dla BERT i GPT2 są przedstawione tutaj:

AutoTokenizer

BERTTokenizer

GPT2Tokenizer

Na przykład, rozważ poniższe fragmenty kodu do utworzenia instancji tokenizera opartego na BERT:

tokenizer = transformers.BertTokenizer.from_

pretrained('bert-base-uncased', do_lower_case=True)

tokenizer = AutoTokenizer.from_pretrained

("bert-base-uncased")

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

W trzech poprzednich fragmentach kodu pożądaną architekturę można wywnioskować z nazwy wstępnie wytrenowanego modelu, którą przekazujesz do metody from_pretrained(). Zgodnie z konwencją, fragment kodu „uncased" oznacza, że wszystkie słowa w modelu są pisane małymi literami.

**Czym są AutoClass?**

Klasy z prefiksem AutoClass automatycznie pobierają odpowiedni model, podając nazwę/ścieżkę do wstępnie wytrenowanych wag/konfiguracji/słownika. Ponadto utworzenie instancji AutoModel, AutoConfig lub AutoTokenizer tworzy klasę odpowiedniej architektury. Na przykład poniższy fragment kodu tworzy instancję BertModel:

model = AutoModel.from_pretrained('bert-base-cased')

Zatem, jeśli ciąg określony w metodzie from_pretrained() jest punktem kontrolnym BERT (takim jak bert-base-uncased), to poniższe

fragmenty kodu mają ten sam efekt:

AutoTokenizer.from_pretrained("bert-base-uncased")

BertTokenizer.from_pretrained("bert-base-uncased")

Klasy Auto* umożliwiają również określenie dowolnego punktu kontrolnego, a

załadowany zostanie poprawny model. Przykład znajduje się tutaj:

# zwraca poprawny egzemplarz GPT2Tokenizer:

AutoTokenizer.from_pretrained("gpt2")

# to się nie powiedzie (niezgodność modelu):

BertTokenizer.from_pretrained("gpt2")

**TOKENIZERY HUGGING FACE**

W tej sekcji opisano wyniki treningu z następującymi tokenizerami przy użyciu Hugging Face:

• Tokenizery wolne i szybkie

• Model BPE (model oparty na częstotliwości)

• Tokenizery WordPiece

• Tokenizery Unigram (model oparty na prawdopodobieństwie)

Porównamy również tokeny wygenerowane za pomocą Hugging Face z algorytmami tokenizacji SOTA (najnowocześniejszymi). Podstawowa różnica między tymi algorytmami dotyczy wyboru par znaków do scalenia, a także polityki scalania, której te algorytmy używają do generowania tokenów końcowych.

**Tokenizery wolne i szybkie**

Istnieje różnica między tokenizerami wolnymi i szybkimi. Tokenizery wolne są napisane w Pythonie i znajdują się w bibliotece Hugging Face Transformers. Natomiast wersje szybkie znajdują się w tokenizerach Hugging Face i są napisane w języku programowania Rust. Należy również pamiętać o następującej kwestii dotyczącej wolnych i szybkich tokenizerów: różnica w szybkości tokenizacji jest najbardziej widoczna, gdy znaczna ilość tekstu jest przetwarzana równolegle. W rzeczywistości wolne tokenizery mogą być szybsze niż szybkie, przetwarzając niewielką ilość tekstu. Szybkie tokenizery obsługują dwie inne ważne funkcje: paralelizację i mapowanie przesunięć, które odnosi się do rejestrowania pozycji indeksu tokenów. Ta ostatnia funkcjonalność obsługuje mapowanie słów na wygenerowane tokeny, a także możliwość mapowania znaków tekstowych na token, w którym są osadzone.

**Protokoły klasyfikacji tokenów**

Listing 4 przedstawia zawartość pliku tokens_classify1.py, który pokazuje, jak przeprowadzić klasyfikację tokenów.

```
from transformers import pipeline

sentence = "I Love Chicago Deep Dish Pizza"
token_classifier = pipeline("token-classification")


tokens = token_classifier(sentence)
print("sentence:",sentence)
print()
print("tokens:  ",tokens)
```

Listing 4 rozpoczyna się od instrukcji import i inicjuje sekwencję zmiennych jako ciąg tekstowy. Następnie zmienna token_classifier jest inicjowana jako instancja klasy potoku, która również określa

token-classification jako typ zadania. Następnie zmienna token jest inicjowana wynikiem wywołania metody token_classifier() ze zmienną sentence. Ostatnia część Listingu 4 wyświetla zawartość zmiennej sentence oraz tokeny zmiennych. Uruchom kod z Listingu 4, a zobaczysz następujący wynik:

```
sentence: I Love Chicago Deep Dish Pizza

tokens:   [{'entity': 'I-MISC', 'score':
0.59600025, 'index': 1, 'word': 'I', 'start': 0,
'end': 1}, {'entity': 'I-MISC', 'score': 0.500077,
'index': 2, 'word': 'Love', 'start': 2, 'end':
6}, {'entity': 'I-ORG', 'score': 0.8103038,
'index': 3, 'word': 'Chicago', 'start': 7, 'end':
14}, {'entity': 'I-ORG', 'score': 0.56864274,
'index': 4, 'word': 'Deep', 'start': 15, 'end':
19}, {'entity': 'I-ORG', 'score': 0.71021026,
'index': 5, 'word': 'Di', 'start': 20, 'end': 22},
{'entity': 'I-ORG', 'score': 0.771432, 'index': 6,
'word': '##sh', 'start': 22, 'end': 24}, {'entity':
'I-ORG', 'score': 0.8432511, 'index': 7, 'word':
'Pizza', 'start': 25, 'end': 30}]
```

Poprzednie dane wyjściowe zostały przeformułowane i zmniejszone, aby ułatwić przeglądanie typów encji i różnych wartości powiązanych z tokenami:

```
[
{'entity': 'I-MISC','score': 0.59600025,'index':
1,'word':'I',    'start': 0, 'end': 1},
{'entity': 'I-MISC','score': 0.500077,  'index':
2,'word':'Love', 'start': 2, 'end': 6},
{'entity': 'I-ORG', 'score': 0.8103038, 'index':
3,'word':'Chicago','start':7,'end': 14},


{'entity': 'I-ORG', 'score': 0.56864274,'index':
4,'word':'Deep','start': 15, 'end': 19},
{'entity': 'I-ORG', 'score': 0.71021026,'index':
5,'word':'Di',  'start': 20, 'end': 22},
{'entity': 'I-ORG', 'score': 0.771432,  'index':
6,'word':'##sh', 'start':22, 'end': 24},
{'entity': 'I-ORG', 'score': 0.8432511, 'index':
7,'word':'Pizza','start':25, 'end': 30}
]
```

Uruchom kod z Listingu 4 w Pythonie 3.7. Jeśli uruchomisz kod w Pythonie 3.9, zobaczysz następujący komunikat o błędzie, który wyświetla wszystkie dostępne zadania:

```
KeyError: "Unknown task token-classification,
available tasks are ['feature-extraction',
'sentiment-analysis', 'ner', 'question-
answering', 'table-question-answering', 'fill-
mask', 'summarization', 'translation', 'text2text-
generation', 'text-generation', 'zero-shot-
classification', 'conversational', 'translation_XX_
to_YY']"
```

**TOKENIZACJA DLA MODELU DISTILBERT**

Pisząc kod Pythona dla różnych modeli opartych na BERT, należy łączyć tokenizery i modele należące do tego samego modelu. Na przykład, jeśli odwołujesz się do tokenizera z klasy BertTokenizer, musisz również utworzyć instancję modelu opartego na BERT zamiast modelu DistilBERT. Chociaż to drugie jest oczywiste, nie zobaczysz komunikatu o błędzie, jeśli je wymieszasz; zamiast tego zobaczysz komunikaty ostrzegawcze o „nieoczekiwanej tokenizacji”. Listing 5 przedstawia zawartość pliku distilbert_tokenizer1.py, który pokazuje, co się dzieje, gdy „wymieszasz” tokenizery i modele.

```
# notice the mismatch in this code snippet:
from transformers import DistilBertTokenizer,
BertModel

tokenizer = DistilBertTokenizer.from_pretrained
('bert-base-uncased')
print("tokenizer size:   ",tokenizer.vocab_size)
print("model max length: ",tokenizer.model_max_
length)
print("model input names:",tokenizer.model_input_
names)
print()

text1 = "Pizza with four toppings and trimmings."
marked_text1 = "[CLS] " + text1 + " [SEP]"
tokenized_text1 = tokenizer.tokenize(marked_text1)

print("input sentence #1:")
print(text1)
print()

print("Tokens from input sentence #1:")
print(tokenized_text1)
print()

print("Some tokens in BERT:")
print(list(tokenizer.vocab.keys())[1000:1020])
print()
```

Listing 5 rozpoczyna się od instrukcji importu, a następnie inicjalizuje zmienną tokenizer jako instancję tokenizera powiązanego ze zbiorem danych bert-base-uncased, który składa się wyłącznie z małych liter i nie zawiera znaków diakrytycznych. Ta zmienna umożliwia wyświetlenie maksymalnej długości modelu, a także nazw wejściowych modelu. Kolejna część Listingu 2.5 inicjuje zmienną tekstową text1, a następnie zmienna marked_text1 ma początkowy i końcowy token specjalny. Zmienna tokenized_text1 jest inicjalizowana wynikiem tokenizacji tekstu w ciągu marked_text1. Następnie wyświetlana jest zawartość zmiennych text1 i tokenized_text1, a następnie lista istniejących tokenów. Uruchom kod z Listingu 5, a zobaczysz następujący wynik. Zwróć uwagę na niepoprawną wartość maksymalnej długości, która jest pogrubiona, a także na tokenizację podwyrazów dla dodatków i dodatków:

```
tokenizer size:    30522
model max length:  100000000000000000198846248386 56
model input names: ['input_ids', 'attention_mask']

input sentence #1:
Pizza with four toppings and trimmings.

Tokens from input sentence #1:
['[CLS]', 'pizza', 'with', 'four', 'topping',
'##s', 'and', 'trim', '##ming', '##s', '.',
'[SEP]']

Some tokens in BERT:
['"', '#', '$', '%', '&', "'", '(', ')', '*', '+',
',', '-', '.', '/', '0', '1', '2', '3', '4', '5']
```

Dodatkowo zobaczysz następujące komunikaty ostrzegawcze:

```
The tokenizer class you load from this checkpoint
is not the same type as the class this function
is called from. It may result in unexpected
tokenization.
The tokenizer class you load from this checkpoint
is 'BertTokenizer'.
The class this function is called from is
'DistilBertTokenizer'.
```

Należy zauważyć, że komunikat ostrzegawczy nie oznacza, że błędy koniecznie wystąpią; należy jednak sprawdzić kod, jeśli zauważysz jakiekolwiek nietypowe lub nieoczekiwane rezultaty. Listing 6 przedstawia zawartość pliku distilbert_tokenizer2.py, która pokazuje, jak poprawić niedopasowane klasy i modele z Listingu 5.

```
# notice the corrections in this code sample
from transformers import DistilBertTokenizer,
DistilBertModel
tokenizer = DistilBertTokenizer.from_pretrained
('distilbert-base-uncased')
print("tokenizer size:   ",tokenizer.vocab_size)

print("model max length: ",tokenizer.model_max_
length)
print("model input names:",tokenizer.model_input_
names)
print()

text1 = "Pizza with four toppings and trimmings."
marked_text1 = "[CLS] " + text1 + " [SEP]"
tokenized_text1 = tokenizer.tokenize(marked_text1)

print("input sentence #1:")
print(text1)
print()

print("Tokens from input sentence #1:")
print(tokenized_text1)
print()

print("Some tokens in BERT:")
print(list(tokenizer.vocab.keys())[1000:1020])
print()
```

Listing 6 jest niemal identyczny z Listingiem 5; jedyne różnice dotyczą pierwszych dwóch wierszy, które są pogrubione. Uruchom kod z Listingu 6, a zobaczysz niemal taki sam wynik jak w Listingu 2.5, ale tym razem z poprawną wartością maksymalnej długości i bez komunikatów ostrzegawczych.

```
tokenizer size:    30522
model max length:  512
model input names: ['input_ids', 'attention_mask']

input sentence #1:
Pizza with four toppings and trimmings.

Tokens from input sentence #1:
['[CLS]', 'pizza', 'with', 'four', 'topping',
'##s', 'and', 'trim', '##ming', '##s', '.',
'[SEP]']
```

```
Some tokens in BERT:
['"', '#', '$', '%', '&', "'", '(', ')', '*', '+',
',', '-', '.', '/', '0', '1', '2', '3', '4', '5']
```

**TECHNIKI SELEKCJI TOKENÓW W LLMS**

LLM i sieci neuronowe mają kilka typów parametrów. Na przykład parametry w sieciach neuronowych to krawędzie łączące neurony w dwóch różnych warstwach. Parametrom przypisuje się wagi numeryczne, które są aktualizowane na etapie treningu LLM lub sieci neuronowej. Po drugie, LLM i sieci neuronowe zazwyczaj mają hiperparametry, których wartości są ustawiane przed etapem treningu LLM lub sieci neurowej. Po trzecie, wiele modeli LLM, takich jak modele GPT-x i inne modele LLM oparte na dekoderach, zapewnia regulowane parametry wnioskowania, które wpływają na wynik modelu. Oto niektóre z nich:

- Maksymalna liczba nowych tokenów
- Długość tokena
- Tokeny stopu
- Wartość szczytowa próbki K
- Wartość szczytowa próbki P
- Temperatura

Ponieważ parametry wnioskowania będą bardziej zrozumiałe po zapoznaniu się z precyzyjnym dostrajaniem modelu LLM, omówienie powyższej listy parametrów wnioskowania zostało odłożone na potem.

**PODSUMOWANIE**

Ta część rozpoczęła się od wprowadzenia do koncepcji pretokenizacji, a także różnic między pretokenizacją a tokenizacją, wraz z przykładowym kodem w Pythonie. Następnie zapoznano się z niuansami tokenizerów i ogólnym algorytmem mapowania tokenów na indeks. Ponadto poznano tokenizery słów, znaków i podsłów. Następnie poznano ograniczenia tokenizerów opartych na słowach i językach o wielu alfabetach (takich jak japoński), a także ograniczenia tokenizerów opartych na znakach. Ponadto poznano algorytmy tokenizacji podsłów, takie jak BPE, a także algorytmy WordPiece i SentencePiece. Zapoznano się z tokenizerami Hugging Face i różnicami między wolnymi i szybkimi tokenizerami. Następnie poznano techniki selekcji tokenów.