


```

| 0 Tesla T4 Off | 00000000:00:04.0 Off |
0 |
| N/A 35C P8 9W / 70W | 0MiB / 15360MiB |
0% Default |
|
N/A |
+-----+
+-----+

+-----+
-----+
| Processes:
|
| GPU GI CI PID Type Process name
GPU Memory |
| ID ID
Usage |
|
=====
=====|
| No running processes found
|
+-----+
-----+

```

Jeśli akcelerator jest niedostępny (polecenie skończyło się błędem), to zmieniamy środowisko wykonawcze wybierając z menu "Środowisko wykonawcze" -> "Zmień typ środowiska wykonawczego" -> GPU.

Podpięcie dysku Google

Kolejnym elementem przygotowań, który jest opcjonalny, jest dołączenie własnego dysku Google Drive do środowiska Colab. Dzięki temu możliwe jest zapisywanie wytrenowanych modeli, w trakcie procesu treningu, na "zewnętrznym" dysku. Jeśli Google Colab doprowadzi do przerwania procesu treningu, to mimo wszystko pliki, które udało się zapisać w trakcie treningu nie przepadną. Możliwe będzie wznowienie treningu już na częściowo wytrenowanym modelu.

W tym celu montujemy dysk Google w Colabie. Wymaga to autoryzacji narzędzia Colab w Google Drive.

```

from google.colab import drive
drive.mount('/content/gdrive')

Mounted at /content/gdrive

```

Po podmontowaniu dysku mamy dostęp do całej zawartości Google Drive. Wskazując miejsce zapisywania danych w trakcie treningu należy wskazać ścieżkę zaczynającą się od `/content/gdrive`, ale należy wskazać jakiś podkatalog w ramach naszej przestrzeni dyskowej.

Pełna ścieżka może mieć postać `/content/gdrive/MyDrive/output`. Przed uruchomieniem treningu warto sprawdzić, czy dane zapisują się na dysku.

Instalacja bibliotek Pythona

Następnie zainstalujemy wszystkie niezbędne biblioteki. Poza samą biblioteką `transformers`, instalujemy również biblioteki do zarządzania zbiorami danych `datasets`, bibliotekę definiującą wiele metryk wykorzystywanych w algorytmach AI `evaluate` oraz dodatkowe narzędzia takie jak `sacremoses` oraz `sentencepiece`.

```
!pip install transformers==4.35.2 sacremoses==0.1.1 datasets==2.15.0
evaluate==0.4.1 sentencepiece==0.1.99 accelerate==0.24.1
```

```
Requirement already satisfied: transformers==4.35.2 in
/usr/local/lib/python3.10/dist-packages (4.35.2)
```

```
Collecting sacremoses==0.1.1
```

```
  Downloading sacremoses-0.1.1-py3-none-any.whl (897 kB)
```

```
0:00:00 897.5/897.5 kB 4.7 MB/s eta
```

```
0:00:00 521.2/521.2 kB 34.3 MB/s eta
```

```
0:00:00 84.1/84.1 kB 11.6 MB/s eta
```

```
anylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.3 MB)
```

```
0:00:00 1.3/1.3 MB 25.4 MB/s eta
```

```
0:00:00 261.4/261.4 kB 24.9 MB/s eta
```

```
0:00:00 ent already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from transformers==4.35.2) (3.13.1)
```

```
Requirement already satisfied: huggingface-hub<1.0,>=0.16.4 in
/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(0.19.4)
```

```
Requirement already satisfied: numpy>=1.17 in
/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(1.23.5)
```

```
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(23.2)
```

```
Requirement already satisfied: pyyaml>=5.1 in
/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(6.0.1)
```

```
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(2023.6.3)
```

```
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(2.31.0)
```

```
Requirement already satisfied: tokenizers<0.19,>=0.14 in
```

```

/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(0.15.0)
Requirement already satisfied: safetensors>=0.3.1 in
/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(0.4.1)
Requirement already satisfied: tqdm>=4.27 in
/usr/local/lib/python3.10/dist-packages (from transformers==4.35.2)
(4.66.1)
Requirement already satisfied: click in
/usr/local/lib/python3.10/dist-packages (from sacremoses==0.1.1)
(8.1.7)
Requirement already satisfied: joblib in
/usr/local/lib/python3.10/dist-packages (from sacremoses==0.1.1)
(1.3.2)
Requirement already satisfied: pyarrow>=8.0.0 in
/usr/local/lib/python3.10/dist-packages (from datasets==2.15.0)
(10.0.1)
Collecting pyarrow-hotfix (from datasets==2.15.0)
  Downloading pyarrow_hotfix-0.6-py3-none-any.whl (7.9 kB)
Collecting dill<0.3.8,>=0.3.0 (from datasets==2.15.0)
  Downloading dill-0.3.7-py3-none-any.whl (115 kB)


---


115.3/115.3 kB 12.9 MB/s eta
0:00:00
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-
packages (from datasets==2.15.0) (1.5.3)
Requirement already satisfied: xxhash in
/usr/local/lib/python3.10/dist-packages (from datasets==2.15.0)
(3.4.1)
Collecting multiprocessing (from datasets==2.15.0)
  Downloading multiprocessing-0.70.15-py310-none-any.whl (134 kB)


---


134.8/134.8 kB 12.3 MB/s eta
0:00:00
Requirement already satisfied: fsspec[http]<=2023.10.0,>=2023.1.0 in
/usr/local/lib/python3.10/dist-packages (from datasets==2.15.0)
(2023.6.0)
Requirement already satisfied: aiohttp in
/usr/local/lib/python3.10/dist-packages (from datasets==2.15.0)
(3.9.1)
Collecting responses<0.19 (from evaluate==0.4.1)
  Downloading responses-0.18.0-py3-none-any.whl (38 kB)
Requirement already satisfied: psutil in
/usr/local/lib/python3.10/dist-packages (from accelerate==0.24.1)
(5.9.5)
Requirement already satisfied: torch>=1.10.0 in
/usr/local/lib/python3.10/dist-packages (from accelerate==0.24.1)
(2.1.0+cu121)
Requirement already satisfied: attrs>=17.3.0 in
/usr/local/lib/python3.10/dist-packages (from aiohttp-
>datasets==2.15.0) (23.1.0)

```

Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets==2.15.0) (6.0.4)

Requirement already satisfied: yarll<2.0,>=1.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets==2.15.0) (1.9.4)

Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets==2.15.0) (1.4.0)

Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets==2.15.0) (1.3.1)

Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets==2.15.0) (4.0.3)

Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.16.4->transformers==4.35.2) (4.5.0)

Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->transformers==4.35.2) (3.3.2)

Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->transformers==4.35.2) (3.6)

Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->transformers==4.35.2) (2.0.7)

Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->transformers==4.35.2) (2023.11.17)

Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate==0.24.1) (1.12)

Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate==0.24.1) (3.2.1)

Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate==0.24.1) (3.1.2)

Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate==0.24.1) (2.1.0)

Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas->datasets==2.15.0) (2.8.2)

Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->datasets==2.15.0) (2023.3.post1)

Requirement already satisfied: six>=1.5 in

```

/usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1-
>pandas->datasets==2.15.0) (1.16.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.10.0-
>accelerate==0.24.1) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in
/usr/local/lib/python3.10/dist-packages (from sympy->torch>=1.10.0-
>accelerate==0.24.1) (1.3.0)
Installing collected packages: sentencepiece, sacremoses, pyarrow-
hotfix, dill, responses, multiprocessing, accelerate, datasets, evaluate
Successfully installed accelerate-0.24.1 datasets-2.15.0 dill-0.3.7
evaluate-0.4.1 multiprocessing-0.70.15 pyarrow-hotfix-0.6 responses-
0.18.0 sacremoses-0.1.1 sentencepiece-0.1.99

```

Mając zainstalowane niezbędne biblioteki, możemy skorzystać z wszystkich modeli i zbiorów danych zarejestrowanych w katalogu.

Typowym sposobem użycia dostępnych modeli jest:

- *wykorzystanie gotowego modelu*, który realizuje określone zadanie, np. [analizę senetymentu w języku angielskim](#) - model tego rodzaju nie musi być trenowany, wystarczy go uruchomić aby uzyskać wynik klasyfikacji (można to zobaczyć w demo pod wskazanym linkiem),
- *wykorzystanie modelu bazowego*, który jest dotrenowywany do określonego zadania; przykładem takiego modelu jest [HerBERT base](#), który uczony był jako maskowany model języka. Żeby wykorzystać go do konkretnego zadania, musimy wybrać dla niego "głowę klasyfikacyjną" oraz dotrenować na własnym zbiorze danych.

Modele tego rodzaju różnią się od siebie, można je załadować za pomocą wspólnego interfejsu, ale najlepiej jest wykorzystać jedną ze specjalizowanych klas, dostosowanych do zadania, które chcemy zrealizować. Zaczniemy od załadowania modelu BERT base - jednego z najbardziej popularnych modeli, dla języka angielskiego. Za jego pomocą będziemy odgadywać brakujące wyrazy w tekście. Wykorzystamy do tego wywołanie `AutoModelForMaskedLM`.

```

from transformers import AutoModelForMaskedLM, AutoTokenizer

model = AutoModelForMaskedLM.from_pretrained("bert-base-cased")

{"model_id": "4afbbe5282884c438a72755fa105f0ad", "version_major": 2, "version_minor": 0}

{"model_id": "f66f42433b6243c6a71d18d38fe39255", "version_major": 2, "version_minor": 0}

```

Some weights of the model checkpoint at bert-base-cased were not used when initializing BertForMaskedLM: ['bert.pooler.dense.bias', 'cls.seq_relationship.bias', 'cls.seq_relationship.weight', 'bert.pooler.dense.weight']

- This IS expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another

architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).

- This IS NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).

Załadowany model jest modulem PyTorch. Możemy zatem korzystać z API tej biblioteki. Możemy np. sprawdzić ile parametrów ma model BERT base:

```
count = sum(p.numel() for p in model.parameters() if p.requires_grad)
'{:,}'.format(count).replace(',', ' ')
{"type": "string"}
```

Widzimy zatem, że nasz model jest bardzo duży - zawiera ponad 100 milionów parametrów, a jest to tzw. model bazowy. Modele obecnie wykorzystywane mają jeszcze więcej parametrów - duże modele językowe, takie jak ChatGPT posiadają więcej niż 100 miliardów parametrów.

Możemy również podejrzeć samą strukturę modelu.

```
model
BertForMaskedLM(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(28996, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768,
bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
```

```

bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
    )
    )
    (intermediate): BertIntermediate(
        (dense): Linear(in_features=768, out_features=3072,
bias=True)
        (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
        (dense): Linear(in_features=3072, out_features=768,
bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
    )
    )
    )
    )
    )
    (cls): BertOnlyMLMHead(
        (predictions): BertLMPredictionHead(
            (transform): BertPredictionHeadTransform(
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (transform_act_fn): GELUActivation()
                (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
            )
            (decoder): Linear(in_features=768, out_features=28996,
bias=True)
        )
    )
    )
)

```

Tokenizacja tekstu

Załadowanie samego modelu nie jest jednak wystarczające, żeby zacząć go wykorzystywać. Musimy mieć mechanizm zamiany tekstu (łańcucha znaków), na ciąg tokenów, należących do określonego słownika. W trakcie treningu modelu, słownik ten jest określany (wybierany w sposób algorytmiczny) przed właściwym treningiem sieci neuronowej. Choć możliwe jest jego późniejsze rozszerzenie (douczenie na danych treningowych, pozwala również uzyskać reprezentację brakujących tokenów), to zwykle wykorzystuje się słownik w postaci, która została określona przed treningiem sieci neuronowej. Dlatego tak istotne jest wskazanie właściwego słownika dla tokenizera dokonującego podziału tekstu.

Biblioteka posiada klasę `AutoTokenizer`, która akceptuje nazwę modelu, co pozwala automatycznie załadować słownik korespondujący z wybranym modelem sieci neuronowej. Trzeba jednak pamiętać, że jeśli używamy 2 modeli, to każdy z nich najpewniej będzie miał inny słownik, a co za tym idzie muszą one mieć własne instancje klasy `Tokenizer`.

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

{"model_id": "dae374999bc5422896522ed19c5aa4cb", "version_major": 2, "version_minor": 0}

{"model_id": "ba1ab8cb3adf41ec947f32f661139a16", "version_major": 2, "version_minor": 0}

{"model_id": "fbbb26bba4c840098a7e0bd2de6b5c0c", "version_major": 2, "version_minor": 0}

BertTokenizerFast(name_or_path='bert-base-cased', vocab_size=28996,
model_max_length=512, is_fast=True, padding_side='right',
truncation_side='right', special_tokens={'unk_token': '[UNK]',
'sep_token': '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]',
'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True),
added_tokens_decoder={
  0: AddedToken("[PAD]", rstrip=False, lstrip=False,
single_word=False, normalized=False, special=True),
  100: AddedToken("[UNK]", rstrip=False, lstrip=False,
single_word=False, normalized=False, special=True),
  101: AddedToken("[CLS]", rstrip=False, lstrip=False,
single_word=False, normalized=False, special=True),
  102: AddedToken("[SEP]", rstrip=False, lstrip=False,
single_word=False, normalized=False, special=True),
  103: AddedToken("[MASK]", rstrip=False, lstrip=False,
single_word=False, normalized=False, special=True),
}
```

Tokenizer posługuje się słownikiem o stałym rozmiarze. Podowuje to oczywiście, że nie wszystkie wyrazy występujące w tekście, będą się w nim znajdowały. Co więcej, jeśli użyjemy tokenizera do podziału tekstu w innym języku, niż ten dla którego został on stworzony, to taki tekst będzie dzielony na większą liczbę tokenów.

```
sentence1 = tokenizer.encode(
    "The quick brown fox jumps over the lazy dog.",
    return_tensors="pt"
)
print(sentence1)
print(sentence1.shape)

sentence2 = tokenizer.encode("Zażółć gęślą jaźń.",
    return_tensors="pt")
```

```

print(sentence2)
print(sentence2.shape)

tensor([[ 101, 1109, 3613, 3058, 17594, 15457, 1166, 1103,
          16688, 3676,
           119, 102]])
torch.Size([1, 12])
tensor([[ 101, 163, 1161, 28259, 7774, 20671, 7128, 176,
          28221, 28244,
          1233, 28213, 179, 1161, 28257, 19339, 119, 102]])
torch.Size([1, 18])

```

Korzystając z tokenizera dla języka angielskiego do podziału polskiego zdania, widzimy, że otrzymujemy znacznie większą liczbę tokenów. Żeby zobaczyć, w jaki sposób tokenizer dokonał podziału tekstu, możemy wywołać `convert_ids_to_tokens`:

```

print("|".join(tokenizer.convert_ids_to_tokens(list(sentence1[0]))))
print("|".join(tokenizer.convert_ids_to_tokens(list(sentence2[0]))))

[CLS]|The|quick|brown|fox|jumps|over|the|lazy|dog|.|[SEP]
[CLS]|Z|##a|##ż|##ó|##ł|##ć|g|##ę|##ś|##ł|##ą|j|##a|##ż|##ń|.|[SEP]

```

Widzimy, że dla języka angielskiego wszystkie wyrazy w zdaniu zostały przekształcone w pojedyncze tokeny. W przypadku zdania w języku polskim, zawierającego szereg znaków diakrytycznych sytuacja jest zupełnie inna - każdy znak został wyodrębniony do osobnego sub-tokenu. To, że mamy do czynienia z sub-tokenami sygnalizowane jest przez dwa krzyżyki poprzedzające dany sub-token. Oznaczają one, że ten sub-token musi być sklejony z poprzedzającym go tokenem, aby uzyskać właściwy łańcuch znaków.

Zadanie 1 (0.5 punkt)

Wykorzystaj tokenizer dla modelu `allegro/herbert-base-cased`, aby dokonać tokenizacji tych samych zdań. Jakie wnioski można wyciągnąć przyglądając się sposobowi tokenizacji za pomocą różnych słowników?

```

# your_code
polishTokenizer=AutoTokenizer.from_pretrained("allegro/herbert-base-cased")
sentence1 = polishTokenizer.encode(
    "The quick brown fox jumps over the lazy dog.",
    return_tensors="pt"
)
print(sentence1)
print(sentence1.shape)

sentence2 = polishTokenizer.encode("Zażółć gęślą jaźń.",
    return_tensors="pt")
print(sentence2)
print(sentence2.shape)

```

```

print("|".join(polishTokenizer.convert_ids_to_tokens(list(sentence1[0]
))))
print("|".join(polishTokenizer.convert_ids_to_tokens(list(sentence2[0]
))))

{"model_id":"ef3518deb159464e992178ebe7a2f014","version_major":2,"version_minor":0}

{"model_id":"4da361aba34b41d384ad5e23c7d415d8","version_major":2,"version_minor":0}

{"model_id":"2bc3484e6978467d8b5b7aeaf9da9f54","version_major":2,"version_minor":0}

{"model_id":"98155e6ac58c4f8197b4f306ac295616","version_major":2,"version_minor":0}

{"model_id":"74e740b3229346b4863f740518d6a8cc","version_major":2,"version_minor":0}

tensor([[ 0, 7117, 22991, 4879, 25015, 1016, 3435, 1055,
2202, 4952,
1010, 83, 10259, 6854, 2050, 3852, 2065, 1031,
1899, 2]])
torch.Size([1, 20])
tensor([[ 0, 2237, 7227, 1048, 7029, 46389, 2059, 272,
1059, 1899,
2]])
torch.Size([1, 11])
<s>|The</w>|qui</w>|brow</w>|fo</w>|jump</w>|o</w>|
the</w>|la</w>|do</w>|. </w>|</s>
<s>|Za</w>|żół</w>|ć</w>|gę</w>|ślą</w>|ja</w>|ż</w>|. </w>|</s>

```

W wynikach tokenizacji poza wyrazami/tokenami występującymi w oryginalnym tekście pojawiają się jeszcze dodatkowe znaczniki [CLS] oraz [SEP] (albo inne znaczniki - w zależności od użytego słownika). Mają one specjalne znaczenie i mogą być wykorzystywane do realizacji specyficznych funkcji związanych z analizą tekstu. Np. reprezentacja tokenu [CLS] wykorzystywana jest w zadaniach klasyfikacji zdań. Z kolei token [SEP] wykorzystywany jest do odróżnienia zdań, w zadaniach wymagających na wejściu dwóch zdań (np. określenia, na ile zdania te są podobne do siebie).

Modelowanie języka

Modele pretrenowane w reżimie self-supervised learning (SSL) nie posiadają specjalnych zdolności w zakresie rozwiązywania konkretnych zadań z zakresu przetwarzania języka naturalnego, takich jak odpowiadanie na pytania, czy klasyfikacja tekstu (z wyjątkiem bardzo dużych modeli, takich jak np. GPT-3, których model językowy zdolny jest do predykcji np. sensownych odpowiedzi na pytania). Można je jednak wykorzystać do określania

prawdopodobieństwa wyrazów w tekście, a tym samym do sprawdzenia, jaką wiedzę posiada określony model w zakresie znajomości języka, czy też ogólną wiedzę o świecie.

Aby sprawdzić jak model radzi sobie w tych zadaniach, możemy dokonać inferencji na danych wejściowych, w których niektóre wyrazy zostaną zastąpione specjalnymi symbolami maskującymi, wykorzystywanymi w trakcie pre-treningu modelu.

Należy mieć na uwadze, że różne modele mogą korzystać z różnych specjalnych sekwencji w trakcie pretreningu. Np. Bert korzysta z sekwencji [MASK]. Wygląd tokenu maskującego lub jego identyfikator możemy sprawdzić w [pliku konfiguracji tokenizera](#) dystrybuowanym razem z modelem, albo odczytać wprost z instancji tokenizera.

W pierwszej kolejności, spróbujemy uzupełnić brakujący wyraz w angielskim zdaniu.

```
sentence_en = tokenizer.encode(
    "The quick brown [MASK] jumps over the lazy dog.",
    return_tensors="pt"
)
print("|".join(tokenizer.convert_ids_to_tokens(list(sentence_en[0]))))
target = model(sentence_en)
print(target.logits[0][4])

[CLS]|The|quick|brown|[MASK]|jumps|over|the|lazy|dog|.|[SEP]
tensor([-5.3489, -5.6063, -5.1303, ..., -5.9625, -4.1559, -4.5403],
      grad_fn=<SelectBackward0>)
```

Ponieważ zdanie po stokenizowaniu uzupełniane jest znacznikiem [CLS], to zamaskowane słowo znajduje się na 4 pozycji. Wywołanie `target.logits[0][4]` pokazuje tensor z rozkładem prawdopodobieństwa poszczególnych wyrazów, które zostało określone na podstawie parametrów modelu. Możemy wybrać wyrazy, które posiadają największe prawdopodobieństwo, korzystając z wywołania `torch.topk`:

```
import torch

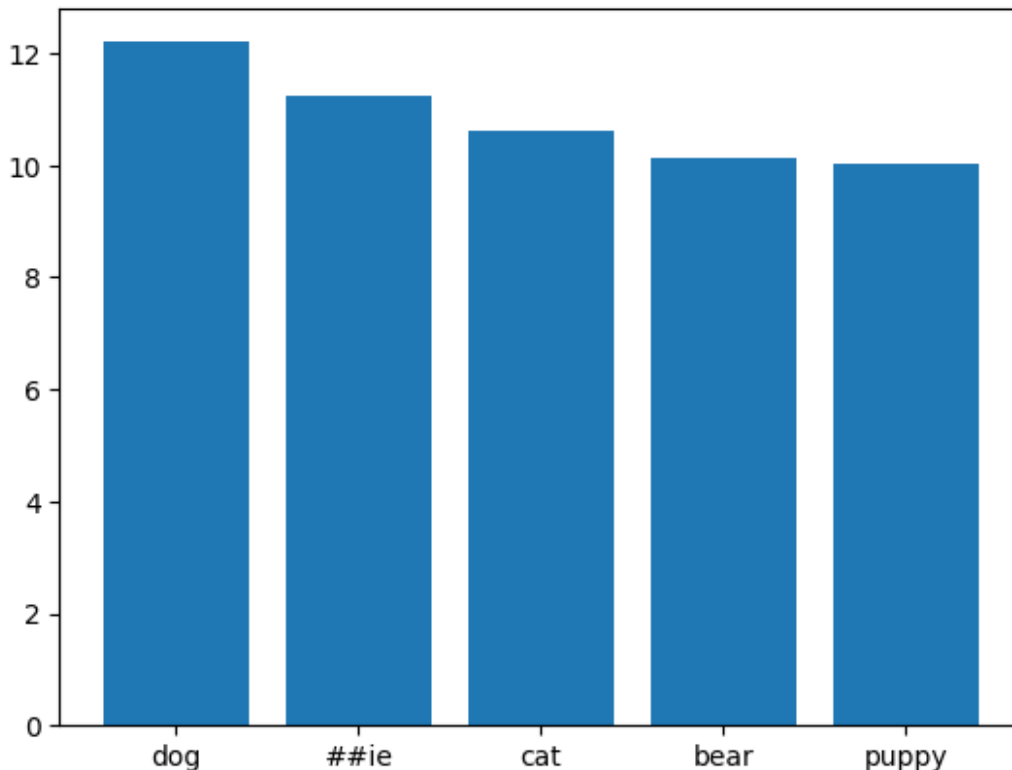
top = torch.topk(target.logits[0][4], 5)
top

torch.return_types.topk(
  values=tensor([12.1982, 11.2289, 10.6009, 10.1278, 10.0120],
    grad_fn=<TopkBackward0>),
  indices=tensor([ 3676,  1663,  5855,  4965, 21566]))
```

Otrzymaliśmy dwa wektory - `values` zawierający składowe wektora wyjściowego sieci neuronowej (nieznormalizowane) oraz `indices` zawierający indeksy tych składowych. Na tej podstawie możemy wyświetlić wyraz, które według modelu są najbardziej prawdopodobnymi uzupełnieniami zamaskowanego wyrazu:

```
words = tokenizer.convert_ids_to_tokens(top.indices)
```

```
import matplotlib.pyplot as plt
plt.bar(words, top.values.detach().numpy())
<BarContainer object of 5 artists>
```



Według modelu najbardziej prawdopodobnym uzupełnieniem brakującego wyrazu jest `dog` (a nie `fox`). Nieco zaskakujący może być drugi wyraz `##ie`, ale po dodaniu go do istniejącego tekstu otrzymamy zdanie: "The quick brownie jumps over the lazy dog", które również wydaje się sensowne (choć nieco zaskakujące).

Zadanie 2 (1.5 punkty)

Wykorzystując model `allegro/herbert-base-cased` zaproponuj zdania z jednym brakującym wyrazem, weryfikując zdolność tego modelu do:

- odmiany przez polskie przypadki,
- uwzględniania długodystansowych związków w tekście,
- reprezentowania wiedzy o świecie.

Dla każdego problemu wymyśl po 3 zdania sprawdzające i wyświetl predykcję dla 5 najbardziej prawdopodobnych wyrazów.

Możesz wykorzystać kod z funkcji `plot_words`, który ułatwi Ci wyświetlanie wyników. Zweryfikuj również jaki token maskujący wykorzystywany jest w tym modelu. Pamiętaj również o załadowaniu modelu `allegro/herbert-base-cased`.

Oceń zdolności modelu w zakresie wskazanych zadań.

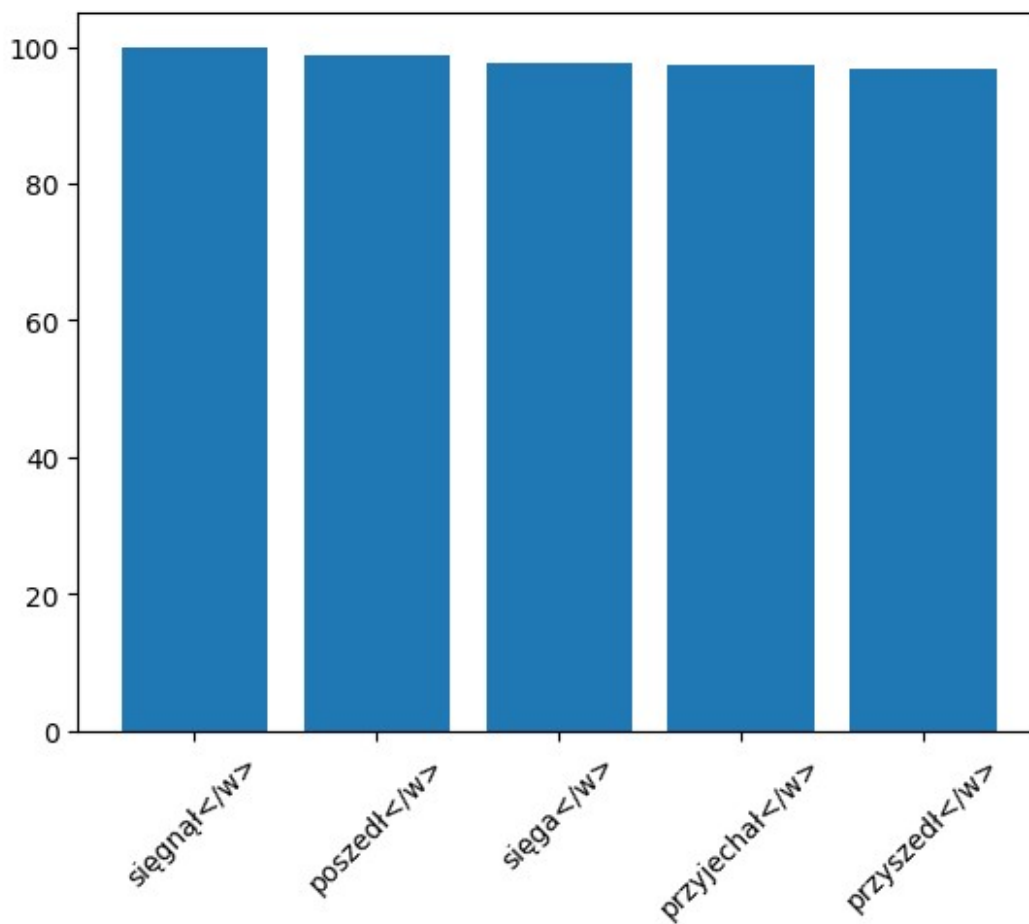
```
def plot_words(sentence, word_model, word_tokenizer, mask="<mask>"):
    sentence = word_tokenizer.encode(sentence, return_tensors="pt")
    tokens = word_tokenizer.convert_ids_to_tokens(list(sentence[0]))
    print("|".join(tokens))
    target = word_model(sentence)
    top = torch.topk(target.logits[0][tokens.index(mask)], 5)
    words = word_tokenizer.convert_ids_to_tokens(top.indices)
    plt.xticks(rotation=45)
    plt.bar(words, top.values.detach().numpy())
    plt.show()

herbert=AutoModelForMaskedLM.from_pretrained("allegro/herbert-base-
cased")
tokenizer=AutoTokenizer.from_pretrained("allegro/herbert-base-cased")

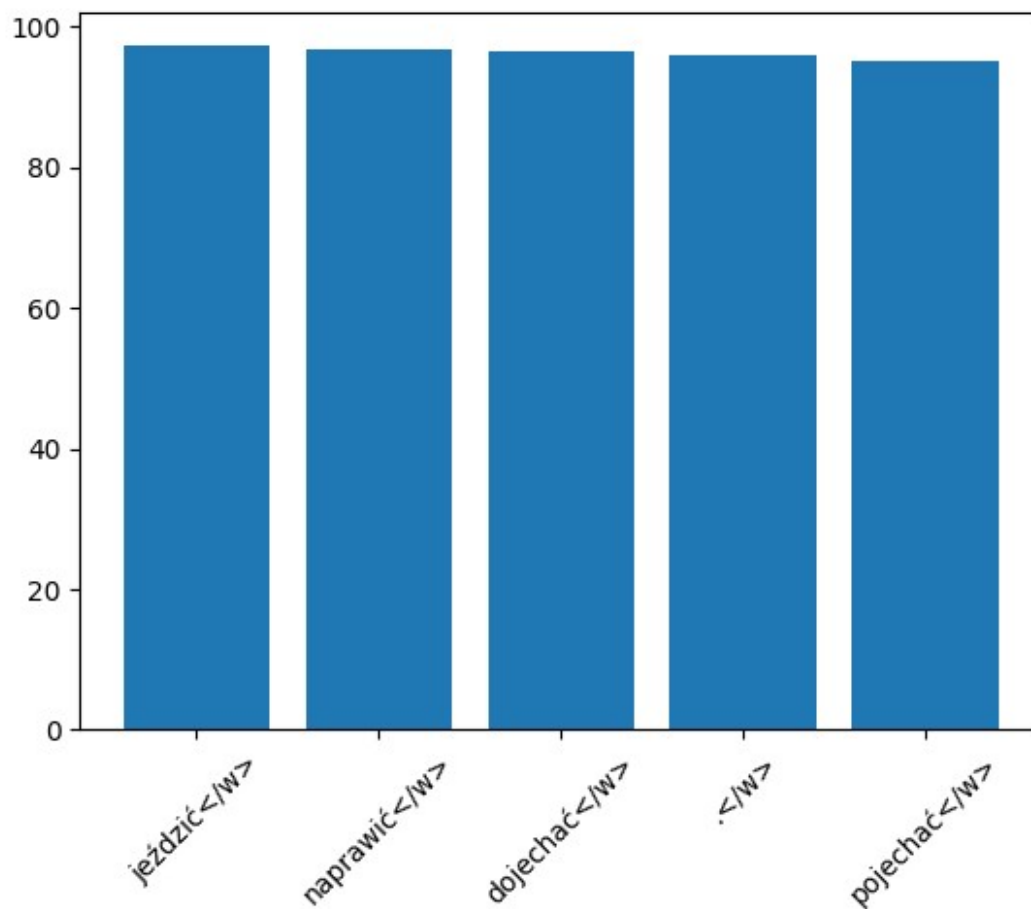
plot_words("Ojciec <mask> po mleko.",herbert,tokenizer)
plot_words("Mam zepsuty samochód, więc nie dam rady <mask>.",
,herbert,tokenizer)
plot_words("Aktualny prezydent USA ma na nazwisko
<mask>.",herbert,tokenizer)
# your_code

{"model_id":"50eeaf75798f4239a17625a7b861726c","version_major":2,"vers
ion_minor":0}

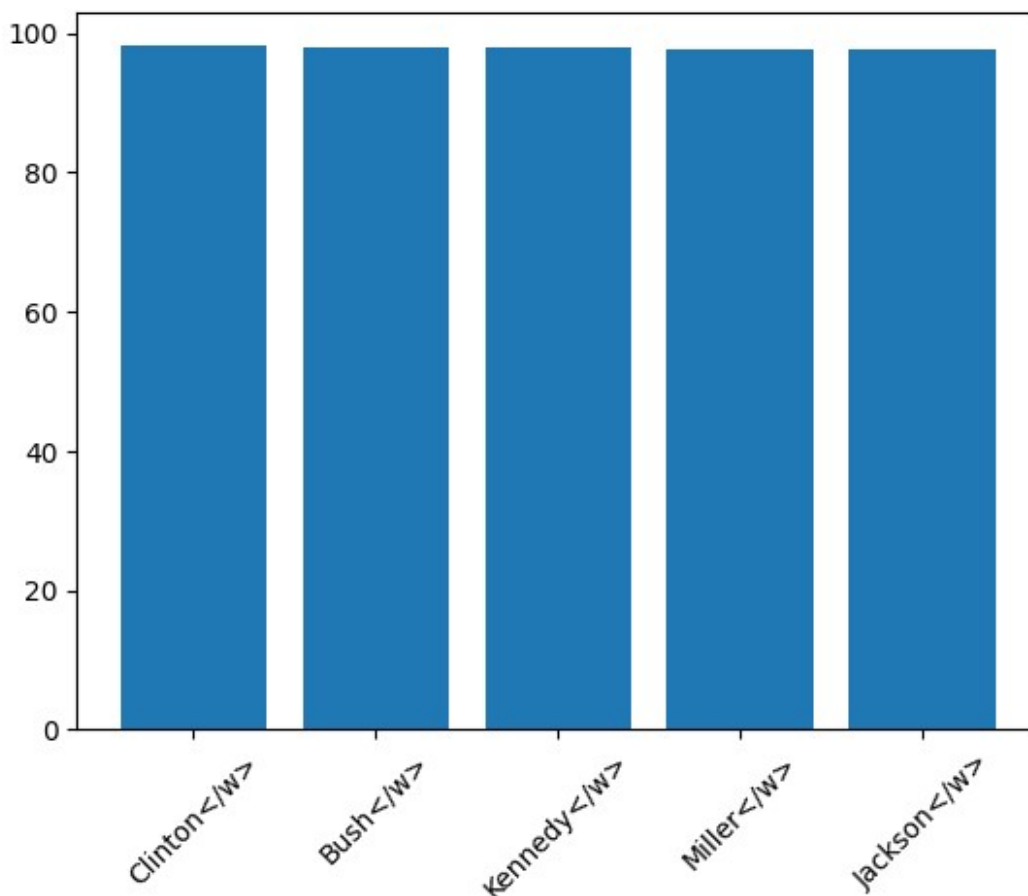
<s>|Ojciec</w>|<mask>|po</w>|mleko</w>|. </w>|</s>
```



<s>|Mam</w>|zepsu|ty</w>|samochód</w>|,</w>|więc</w>|nie</w>|dam</w>|
rady</w>|<mask>|. </w>|</s>



<s>|Aktu|alny</w>|prezydent</w>|USA</w>|ma</w>|na</w>|nazwisko</w>|
<mask>|. </w>|</s>



Dla wszystkich przykładów zdań przewidywane słowa pasują do kontekstu całego zdania. Jednak w ostatnim zdaniu odpowiedź jest błędna.

Klasyfikacja tekstu

Pierwszym zadaniem, które zrealizujemy korzystając z modelu HerBERT będzie klasyfikacja tekstu. Będzie to jednak dość nietypowe zadanie. O ile oczekiwanym wynikiem jest klasyfikacja binarna, czyli dość popularny typ klasyfikacji, o tyle dane wejściowe są nietypowe, gdyż są to pary: (pytanie, kontekst). Celem algorytmu jest określenie, czy na zadane pytanie można odpowiedzieć na podstawie informacji znajdujących się w kontekście.

Model tego rodzaju jest nietypowy, ponieważ jest to zadanie z zakresu klasyfikacji par tekstów, ale my potraktujemy je jak zadanie klasyfikacji jednego tekstu, oznaczając jedynie fragmenty tekstu jako **Pytanie:** oraz **Kontekst:**. Wykorzystamy tutaj zdolność modeli transformacyjnych do automatycznego nauczania się tego rodzaju znaczników, przez co proces przygotowania danych będzie bardzo uproszczony.

Zbiorem danych, który wykorzystamy do treningu i ewaluacji modelu będzie PoQUAD - zbiór inspirowany angielskim [SQuADem](#), czyli zbiorem zawierającym ponad 100 tys. pytań i odpowiadających im odpowiedzi. Zbiór ten powstał niedawno i jest jeszcze rozbudowywany.

Zawiera on pytania, odpowiedzi oraz konteksty, na podstawie których można udzielić odpowiedzi.

W dalszej części laboratorium skoncentrujemy się na problemie odpowiadania na pytania.

Przygotowanie danych do klasyfikacji

Przygotowanie danych rozpoczniemy od sklonowania repozytorium zawierającego pytania i odpowiedzi.

```
from datasets import load_dataset

dataset = load_dataset("clarin-pl/poquad")

{"model_id": "9f9463e57833446fadd90edd0ed0a16c", "version_major": 2, "version_minor": 0}

{"model_id": "46e7aa93f79547bda37011035c6b1608", "version_major": 2, "version_minor": 0}

{"model_id": "8781c361641d4961a57909ea44da0cd1", "version_major": 2, "version_minor": 0}

{"model_id": "63436394d2fb454497a69d58d3174514", "version_major": 2, "version_minor": 0}

{"model_id": "560cc79ad779422eadcf25e833fd646a", "version_major": 2, "version_minor": 0}

{"model_id": "02e0147f7d72487a9758a193e52326eb", "version_major": 2, "version_minor": 0}

{"model_id": "67d5fe37b8d648c09785c63b70eff195", "version_major": 2, "version_minor": 0}

{"model_id": "a4cbbc6b76d5452b9b8fb84884f6b76d", "version_major": 2, "version_minor": 0}
```

Sprawdźmy co znajduje się w zbiorze danych.

```
dataset

DatasetDict({
  train: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 46187
  })
  validation: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 5764
  })
})
```

```
} )  
})
```

Zbiór danych jest podzielony na dwie części: treningową i walidacyjną. Rozmiar części treningowej to ponad 46 tysięcy pytań i odpowiedzi, natomiast części walidacyjnej to ponad 5 tysięcy pytań i odpowiedzi.

Dane zbioru przechowywane są w plikach `poquad_train.json` oraz `poquad_dev.json`. Dostarczenie podziału na te grupy danych jest bardzo częstą praktyką w przypadku publicznych, dużych zbiorów danych, gdyż umożliwia porównywanie różnych modeli, korzystając z dokładnie takiego samego zestawu danych. Prawdopodobnie istnieje również zbiór `poquad_test.json`, który jednak nie jest udostępniany publicznie. Tak jest w przypadku SQuADu - twórcy zbioru automatycznie ewaluują dostarczane modele, ale nie udostępniają zbioru testowego. Dzięki temu trudniej jest nadmiernie dopasować model do danych testowych.

Struktura każdej z dostępnych części jest taka sama. Zgodnie z powyższą informacją zawiera ona następujące elementy:

- `id` - identyfikator pary: pytanie - odpowiedź,
- `title` - tytuł artykułu z Wikipedii, na podstawie którego utworzono parę,
- `context` - fragment treści artykułu z Wikipedii, zawierający odpowiedź na pytanie,
- `question` - pytanie,
- `answers` - odpowiedzi.

Możemy wyświetlić kilka początkowych wpisów części treningowej:

```
dataset['train']['question'][:5]  
  
['Co było powodem powrócenia konceptu porozumieniu monachijskiego?',  
 'Pomiędzy jakimi stronami odbyło się zgromadzenie w sierpniu 1942  
 roku?',  
 'O co ubiegali się polscy przedstawiciele podczas spotkania z  
 sierpnia 1942 roku?',  
 "Który z dyplomatów sprzeciwił się konceptowi konfederacji w  
 listopadzie '42?",  
 'Kiedy oficjalnie doszło do zawarcia porozumienia?']  
  
dataset['train']['answers'][:5]  
  
[{'text': ['wymianą listów Ripka – Stroński'], 'answer_start': [117]},  
 {'text': ['E. Beneša i J. Masaryka z jednej a Wł. Sikorskiego i E.  
 Raczyńskiego'],  
   'answer_start': [197]},  
 {'text': ['podpisanie układu konfederacyjnego'], 'answer_start':  
 [315]},  
 {'text': ['E. Beneš'], 'answer_start': [558]},  
 {'text': ['20 listopada 1942'], 'answer_start': [691]}]
```

Niestety, autorzy zbioru danych, pomimo tego, że dane te znajdują się w źródłowym zbiorze danych, nie udostępniają dwóch ważnych informacji: o tym, czy można odpowiedzieć na dane

pytanie oraz jak brzmi generatywna odpowiedź na pytanie. Dlatego póki nie zostanie to naprawione, będziemy dalej pracować z oryginalnymi plikami zbioru danych, które dostępne są na stronie opisującej zbiór danych: <https://huggingface.co/datasets/clarin-pl/poquad/tree/main>

Pobierz manualnie zbiory `poquad-dev.json` oraz `poquad-train.json`.

```
# !wget
https://huggingface.co/datasets/clarin-pl/poquad/raw/main/poquad-
dev.json
# !wget
https://huggingface.co/datasets/clarin-pl/poquad/resolve/main/poquad-
train.json
```

Dla bezpieczeństwa, jeśli korzystamy z Google drive, to przeniesiemy pliki do naszego dysku:

```
# !mkdir gdrive/MyDrive/poquad
# !mv poquad-dev.json gdrive/MyDrive/poquad
# !mv poquad-train.json gdrive/MyDrive/poquad

# !head -30 gdrive/MyDrive/poquad/poquad-dev.json
```

Struktura pliku odpowiada strukturze danych w zbiorze SQuAD. Dane umieszczone są w kluczu `data` i podzielone na krotki odpowiadające pojedynczym artykułom Wikipedii. W ramach artykułu może być wybranych jeden lub więcej paragrafów, dla których w kluczu `qas` pojawiają się pytania (`question`), flaga `is_impossible`, wskazujące czy można odpowiedzieć na pytanie oraz odpowiedzi (o ile nie jest ustawiona flaga `is_impossible`). Odpowiedzi może być wiele i składają się one z treści odpowiedzi (`text`) traktowanej jako fragment kontekstu, a także naturalnej odpowiedzi na pytanie (`generative_answer`).

Taki podział może wydawać się dziwny, ale zbiór SQuAD zawiera tylko odpowiedzi pierwszego rodzaju. Wynika to z faktu, że w języku angielskim fragment tekstu będzie często stanowił dobrą odpowiedź na pytanie (oczywiście z wyjątkiem pytań dla których odpowiedź to `tak` lub `nie`).

Natomiast ten drugi typ odpowiedzi jest szczególnie przydatny dla języka polskiego, ponieważ często odpowiedź chcemy syntaktycznie dostosować do pytania, co jest niemożliwe, jeśli odpowiedź wskazywana jest jako fragment kontekstu. W sytuacji, w której odpowiedzi były określane w sposób automatyczny, są one oznaczone jako `plausible_answers`.

Zacniemy od wczytania danych i wyświetlenia podstawowych statystyk dotyczących ilości artykułów oraz przypisanych do nich pytań.

```
import json

# Adjust for your needs
path = 'gdrive/MyDrive/poquad'

with open(path + "/poquad-train.json") as input:
    train_data = json.loads(input.read())["data"]
```

```

print(f"Train data articles: {len(train_data)}")

with open(path + "/poquad-dev.json") as input:
    dev_data = json.loads(input.read())["data"]

print(f"Dev data articles: {len(dev_data)}")

print(f"Train questions: {sum([len(e['paragraphs'][0]['qas']) for e in
train_data])}")
print(f"Dev questions: {sum([len(e['paragraphs'][0]['qas']) for e in
dev_data])}")

Train data articles: 8553
Dev data articles: 1402
Train questions: 41577
Dev questions: 6809

```

Ponieważ w pierwszym problemie chcemy stwierdzić, czy na pytanie można udzielić odpowiedzi na podstawie kontekstu, połączymy wszystkie konteksty w jedną tablicę, aby móc losować z niej dane negatywne, gdyż liczba pytań nie posiadających odpowiedzi jest stosunkowo mała, co prowadziłoby utworzenia niezbalansowanego zbioru.

```

all_contexts = [e["paragraphs"][0]["context"] for e in train_data] + [
    e["paragraphs"][0]["context"] for e in dev_data
]

```

W kolejnym kroku zamieniamy dane w formacie JSON na reprezentację zgodną z przyjętym założeniem. Chcemy by kontekst oraz pytanie występowały obok siebie i każdy z elementów był sygnalizowany wyrażeniem: **Pytanie:** i **Kontekst:**. Treść klasyfikowanego tekstu przyporządkowujemy do klucza `text`, natomiast klasę do klucza `label`, gdyż takie są oczekiwania biblioteki Transformer.

Pytania, które mają ustawioną flagę `is_impossible` na `True` trafiają wprost do przekształconego zbioru. Dla pytań, które posiadają odpowiedź, dodatkowo losowany jest jeden kontekst, który stanowi negatywny przykład. Weryfikujemy tylko, czy kontekst ten nie pokrywa się z kontekstem, który przypisany był do pytania. Nie przeprowadzamy bardziej zaawansowanych analiz, które pomogłyby wykuczyć sytuację, w której inny kontekst również zawiera odpowiedź na pytanie, gdyż prawdopodobieństwo wylosowania takiego kontekstu jest bardzo małe.

Na końcu wyświetlamy statystyki utworzonego zbioru danych.

```

import random

tuples = [], []

for idx, dataset in enumerate([train_data, dev_data]):
    for data in dataset:

```

```

context = data["paragraphs"][0]["context"]
for question_answers in data["paragraphs"][0]["qas"]:
    question = question_answers["question"]
    if question_answers["is_impossible"]:
        tuples[idx].append(
            {
                "text": f"Pytanie: {question} Kontekst:
{context}",
                "label": 0,
            }
        )
    else:
        tuples[idx].append(
            {
                "text": f"Pytanie: {question} Kontekst:
{context}",
                "label": 1,
            }
        )
    while True:
        negative_context = random.choice(all_contexts)
        if negative_context != context:
            tuples[idx].append(
                {
                    "text": f"Pytanie: {question}
Kontekst: {negative_context}",
                    "label": 0,
                }
            )
            break

train_tuples, dev_tuples = tuples
print(f"Total count in train/dev:
{len(train_tuples)}/{len(dev_tuples)}")
print(
    f"Positive count in train/dev: {sum([e['label'] for e in
train_tuples])}/{sum([e['label'] for e in dev_tuples])}"
)

Total count in train/dev: 75605/12372
Positive count in train/dev: 34028/5563

```

Widzimy, że uzyskane zbiory danych cechują się dość dobrym zbalansowaniem.

Dobłą praktyką po wprowadzeniu zmian w zbiorze danych, jest wyświetlenie kilku przykładowych punktów danych, w celu wykrycia ewentualnych błędów, które powstały na etapie konwersji zbioru. Pozwala to uniknąć nieprzyjemnych niespodzianek, np. stworzenie identycznego zbioru danych testowych i treningowych.

```
print(train_tuples[0:1])
print(dev_tuples[0:1])
```

```
[{'text': 'Pytanie: Co było powodem powrócenia konceptu porozumieniu  
monachijskiego? Kontekst: Projekty konfederacji zaczęły się załamywać  
5 sierpnia 1942. Ponownie wróciła kwestia monachijska, co uaktywniło  
się wymianą listów Ripka – Stroński. Natomiast 17 sierpnia 1942 doszło  
do spotkania E. Beneša i J. Masaryka z jednej a Wł. Sikorskiego i E.  
Raczyńskiego z drugiej strony. Polscy dyplomaci zaproponowali  
podpisanie układu konfederacyjnego. W następnym miesiącu, tj. 24  
września, strona polska przesłała na ręce J. Masaryka projekt  
deklaracji o przyszłej konfederacji obu państw. Strona czechosłowacka  
projekt przyjęła, lecz już w listopadzie 1942 E. Beneš podważył ideę  
konfederacji. W zamian zaproponowano zawarcie układu sojuszniczego z  
Polską na 20 lat (formalnie nastąpiło to 20 listopada 1942).',  
'label': 1}]
```

```
[{'text': 'Pytanie: Czym są pisma rabiniczne? Kontekst: Pisma  
rabiniczne – w tym Miszna – stanowią kompilację poglądów różnych  
rabinów na określony temat. Zgodnie z wierzeniami judaizmu Mojżesz  
otrzymał od Boga całą Torę, ale w dwóch częściach: jedną część w  
formie pisanej, a drugą część w formie ustnej. Miszna – jako Tora  
ustna – była traktowana nie tylko jako uzupełnienie Tory spisanej, ale  
również jako jej interpretacja i wyjaśnienie w konkretnych sytuacjach  
życiowych. Tym samym Miszna stanowiąca kodeks Prawa religijnego  
zaczęła równocześnie służyć za jego ustnie przekazywany podręcznik.',  
'label': 1}]
```

Ponieważ mamy nowe zbiory danych, możemy opakować je w klasy ułatwiające manipulowanie nimi. Ma to szczególne znaczenie w kontekście szybkiej tokenizacji tych danych, czy późniejszego szybkiego wczytywania wcześniej utworzonych zbiorów danych.

W tym celu wykorzystamy bibliotekę `datasets`. Jej kluczowymi klasami są `Dataset` reprezentujący jeden z podzbiorów zbioru danych (np. podzbiór testowy) oraz `DatasetDict`, który łączy wszystkie podzbiory w jeden obiekt, którym możemy manipulować w całości. (Gdyby autorzy udostępnili odpowiedni skrypt ze zbiorem, moglibyśmy wykorzystać tę bibliotekę bez dodatkowej pracy).

Dodatkowo zapiszemy tak utworzony zbiór danych na dysku. Jeśli później chcielibyśmy wykorzystać stworzony zbiór danych, to możemy to zrobić za pomocą komendy `load_dataset`.

```
from datasets import Dataset, DatasetDict

train_dataset = Dataset.from_list(train_tuples)
dev_dataset = Dataset.from_list(dev_tuples)
datasets = DatasetDict({"train": train_dataset, "dev": dev_dataset})
datasets.save_to_disk(path + "/question-context-classification")

{"model_id": "bb92925b35784471a1f4a2c4d00567bb", "version_major": 2, "version_minor": 0}
```

```
{"model_id": "2bbfba05d958487ea3fe9e9b656b3fd9", "version_major": 2, "version_minor": 0}
```

Dane tekstowe przed przekazaniem do modelu wymagają tokenizacji (co widzieliśmy już wcześniej). Efektywne wykonanie tokenizacji na całym zbiorze danych ułatwione jest przez obiekt `DatasetDict`. Definiujemy funkcję `tokenize_function`, która korzystając z załadowanego tokenizera, zamienia tekst na identyfikatory.

W wywołaniu używamy opcji `padding` - uzupełniamy wszystkie teksty do długości najdłuższego tekstu. Dodatkowo, jeśli któryś tekst wykracza poza maksymalną długość obsługiwaną przez model, to jest on przycinany (`truncation=True`).

Tokenizację aplikujemy do zbioru z wykorzystaniem przetwarzania batchowego (`batched=True`), które pozwala na szybsze stokenizowanie dużego zbioru danych.

```
from transformers import AutoTokenizer

pl_tokenizer = AutoTokenizer.from_pretrained("allegro/herbert-base-cased")

def tokenize_function(examples):
    return pl_tokenizer(examples["text"], padding="max_length",
truncation=True)

tokenized_datasets = datasets.map(tokenize_function, batched=True)
tokenized_datasets["train"]

{"model_id": "98347565f4614ed699dc94f5fa9cee42", "version_major": 2, "version_minor": 0}

{"model_id": "401a9cd5fd9940ea9e93d1f39c054254", "version_major": 2, "version_minor": 0}

Dataset({
  features: ['text', 'label', 'input_ids', 'token_type_ids',
'attention_mask'],
  num_rows: 75605
})
```

Stokenizowane dane zawierają dodatkowe pola: `input_ids`, `token_type_ids` oraz `attention_mask`. Dla nas najważniejsze jest pole `input_ids`, które zawiera identyfikatory tokenów. Pozostałe dwa pola są ustawione na identyczne wartości (wszystkie tokeny mają ten sam typ, maska uwagi zawiera wszystkie niezerowe tokeny), więc nie są one dla nas zbyt interesujące. Zobaczmy pola `text`, `input_ids` oraz `attention_mask` dla pierwszego przykładu:

```
example = tokenized_datasets["train"][0]
print(example)
print(example["text"])
```



```
print(example["input_ids"])
print(example["attention_mask"])
```

[illegible]

[illegible]

Pytanie: Co było powodem powrócenia konceptu porozumieniu monachijskiego? Kontekst: Projekty konfederacji zaczęły się załamywać 5 sierpnia 1942. Ponownie wróciła kwestia monachijska, co uaktywniło się wymianą listów Ripka – Stroński. Natomiast 17 sierpnia 1942 doszło do spotkania E. Beneša i J. Masaryka z jednej a Wł. Sikorskiego i E. Raczyńskiego z drugiej strony. Polscy dyplomaci zaproponowali podpisanie układu konfederacyjnego. W następnym miesiącu, tj. 24 września, strona polska przesłała na ręce J. Masaryka projekt

deklaracji o przyszłej konfederacji obu państw. Strona czechosłowacka projekt przyjęła, lecz już w listopadzie 1942 E. Beneš podważył ideę konfederacji. W zamian zaproponowano zawarcie układu sojuszniczego z Polską na 20 lat (formalnie nastąpiło to 20 listopada 1942).

[illegible][illegible]

[illegible]

Możem też sprawdzić, jak został stokenizowany pierwszy przykład:

```
print("|".join(pl_tokenizer.convert_ids_to_tokens(list(example["input_
ids"]))))
```

[illegible]

[illegible]

Widzimy, że wyrazy podzielone są sensownie, a na końcu tekstu pojawiają się tokeny wypełnienia (PAD). Oznacza to, że zdanie zostało poprawnie skonwertowane.

Możemy sprawdzić, że liczba tokenów w polu `inut_ids`, które są różne od tokenu wypełnienia (`[PAD] = 1`) oraz maska uwagi, mają tę samą długość:

```
print(len([e for e in example["input_ids"] if e != 1]))
print(len([e for e in example["attention_mask"] if e == 1]))
```

169
169

Mając pewność, że przygotowane przez nas dane są prawidłowe, możemy przystąpić do procesu uczenia modelu.

Trening z użyciem transformersów

Biblioteka Transformers pozwala na załadowanie tego samego modelu dostosowanego do różnych zadań. Wcześniej używaliśmy modelu HerBERT do predykcji brakującego wyrazu. Teraz ładujemy ten sam model, ale z inną "głową". Zostanie użyta warstwa, która pozwala na klasyfikację całego tekstu do jednej z n-klas. Wystarczy podmienić klasę, za pomocą której ładujemy model na `AutoModelForSequenceClassification`:

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(
    "allegro/herbert-base-cased", num_labels=2
)
```

model

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at allegro/herbert-base-cased and are newly initialized: ['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(50000, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768,
bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072,
bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768,
bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
```

```

        (dropout): Dropout(p=0.1, inplace=False)
    )
)
)
)
(pooler): BertPooler(
  (dense): Linear(in_features=768, out_features=768, bias=True)
  (activation): Tanh()
)
(dropout): Dropout(p=0.1, inplace=False)
(classifier): Linear(in_features=768, out_features=2, bias=True)
)

```

Komunikat diagnostyczny, który pojawia się przy ładowaniu modelu jest zgodny z naszymi oczekiwaniami. Model HerBERT był trenowany do predykcji tokenów, a nie klasyfikacji tekstu. Dlatego też ostatnia warstwa (`classifier.weight` oraz `classifier.bias`) jest inicjowana losowo. Wagi zostaną ustalone w trakcie procesu fine-tuningu modelu.

Jeśli porównamy wersje modeli załadowane za pomocą różnych klas, to zauważymy, że różnią się one tylko na samym końcu. Jest to zgodne z założeniami procesu pre-treningu i fine-tuningu. W pierwszy etapie model uczy się zależności w języku, korzystając z zadania maskowanego modelowania języka (Masked Language Modeling). W drugim etapie model dostosowywany jest do konkretnego zadania, np. klasyfikacji binarnej tekstu.

Korzystanie z biblioteki Transformers uwalnia nas od manualnego definiowania pętli uczącej, czy wywoływania algorytmu wstecznej propagacji błędu. Trening realizowany jest z wykorzystaniem klasy `Trainer` (i jej specjlizacji). Argumenty treningu określane są natomiast w klasie `TrainingArguments`. Klasy te są [bardzo dobrze udokumentowane](#), więc nie będziemy omawiać wszystkich możliwych opcji.

Najważniejsze opcje są następujące:

- `output_dir` - katalog do którego zapisujemy wyniki,
- `do_train` - wymagamy aby przeprowadzony był trening,
- `do_eval` - wymagamy aby przeprowadzona była ewaluacja modelu,
- `evaluation_strategy` - określenie momentu, w którym realizowana jest ewaluacja,
- `evaluation_steps` - określenie co ile kroków (krok = przetworzenie 1 batcha) ma być realizowana ewaluacja,
- `per_device_train/evaluation_batch_size` - rozmiar batcha w trakcie treningu/ewaluacji,
- `learning_rate` - szybkość uczenia,
- `num_train_epochs` - liczba epok uczenia,
- `logging...` - parametry logowania postępów uczenia,
- `save_strategy` - jak często należy zapisywać wytrenowany model,
- `fp16/bf16` - użycie arytmetyki o zmniejszonej dokładności, przyspieszającej proces uczenia. **UWAGA:** użycie niekompatybilnej arytmetyki skutkuje niemożnością nauczenia modelu, co jednak nie daje żadnych innych błędów lub komunikatów ostrzegawczych.

```

from transformers import TrainingArguments
import numpy as np

arguments = TrainingArguments(
    output_dir=path + "/output",
    do_train=True,
    do_eval=True,
    evaluation_strategy="steps",
    eval_steps=300,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    learning_rate=5e-05,
    num_train_epochs=1,
    logging_first_step=True,
    logging_strategy="steps",
    logging_steps=50,
    save_strategy="epoch",
    # fp16=True,
)

```

W trakcie treningu będziemy chcieli zobaczyć, czy model poprawnie radzi sobie z postawionym mu problemem. Najlepszym sposobem na podglądanie tego procesu jest obserwowanie wykresów. Model może raportować szereg metryk, ale najważniejsze dla nas będą następujące wartości:

- wartość funkcji straty na danych treningowych - jeśli nie spada w trakcie uczenia, znaczy to, że nasz model nie jest poprawnie skonstruowany lub dane uczące są niepoprawne,
- wartość jednej lub wielu metryk uzyskiwanych na zbiorze walidacyjnym - możemy śledzić wartość funkcji straty na zbiorze ewaluacyjnym, ale warto również wyświetlać metryki, które da się łatwiej zinterpretować; dla klasyfikacji zbalansowanego zbioru danych może to być dokładność (accuracy).

Biblioteka Transformers pozwala w zasadzie na wykorzystanie dowolnej metryki, ale szczególnie dobrze współpracuje z metrykami zdefiniowanymi w bibliotece `evaluate` (również autorstwa Huggingface).

Wykorzystanie metryki wymaga od nas zdefiniowania metody, która akceptuje batch danych, który zawiera predykcje (wektory zwrócone na wyjściu modelu) oraz referencyjne wartości - wartości przechowywane w kluczu `label`. Przed obliczeniem metryki konieczne jest "odcyfrowanie" zwróconych wartości. W przypadku klasyfikacji oznacza to po prostu wybranie najbardziej prawdopodobnej klasy i porównanie jej z klasą referencyjną.

Użycie konkretnej metryki realizowane jest za pomocą wywołania `metric.compute`, która akceptuje predykcje (`predictions`) oraz wartości referencyjne (`references`).

```

import evaluate

metric = evaluate.load("accuracy")

```



```
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)

{"model_id": "24f7c5451f7041f8a67a62662669d293", "version_major": 2, "version_minor": 0}
```

Ostatnim krokiem w procesie treningu jest stworzenie obiektu klasy `Trainer`. Akceptuje ona m.in. model, który wykorzystywany jest w treningu, przygotowane argumenty treningu, zbiory do treningu, ewaluacji, czy testowania oraz wcześniej określoną metodę do obliczania metryki na danych ewaluacyjnych.

W przetwarzaniu języka naturalnego dominującym podejściem jest obecnie rozdzielenie procesu treningu na dwa etapy: pre-treining oraz fine-tuning. W pierwszym etapie model trenowany jest w reżimie self-supervised learning (SSL). Wybierane jest zadanie związane najczęściej z modelowaniem języka - może to być kauzalne lub maskowane modelowanie języka.

W *kauzalnym modelowaniu języka* model językowy, na podstawie poprzedzających wyrazów określa prawdopodobieństwo wystąpienia kolejnego wyrazu. W *maskowanym modelowaniu języka* model językowy odgaduje w tekście część wyrazów, która została z niego usunięta.

W obu przypadkach dane, na których trenowany jest model nie wymagają ręcznego oznakowania (tagowania). Wystarczy jedynie posiadać duży korpus danych językowych, aby wytrenować model, który dobrze radzi sobie z jednym z tych zadań. Model tego rodzaju był pokazany na początku laboratorium.

W drugim etapie - fine-tuningu (dostrajaniu modelu) - następuje modyfikacja parametrów modelu, w celu rozwiązania konkretnego zadania. W naszym przypadku pierwszym zadaniem tego rodzaju jest klasyfikacja. Dostroimy zatem model `herbert-base-cased` do zadania klasyfikacji par: pytanie - kontekst.

Wykorzystamy wcześniej utworzone zbiory danych i dodatkowo zmienimy kolejność danych, tak aby uniknąć potencjalnego problemu z korelacją danych w ramach batcha. Wykorzystujemy do tego wywołanie `shuffle`.

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=arguments,
    train_dataset=tokenized_datasets["train"].shuffle(seed=42),
    eval_dataset=tokenized_datasets["dev"].shuffle(seed=42),
    compute_metrics=compute_metrics,
)
```

Zanim uruchomimy trening, załadujemy jeszcze moduł `TensorBoard`. Nie jest to krok niezbędny. `TensorBoard` to biblioteka, która pozwala na wyświetlanie w trakcie procesu trening wartości, które wskazują nam, czy model trenuje się poprawnie. W naszym przypadku będzie to `loss` na danych treningowych, `loss` na danych ewaluacyjnych oraz wartość metryki `accuracy`, którą

zdefiniowaliśmy wcześniej. Wywołanie tej komórki na początku nie da żadnego efektu, ale można ją odświeżać, za pomocą ikony w menu TensorBoard (ewentualnie włączyć automatyczne odświeżanie). Wtedy w miarę upływu treningu będziemy mieli podgląd, na przebieg procesu oraz osiągnięte wartości interesujących nas parametrów.

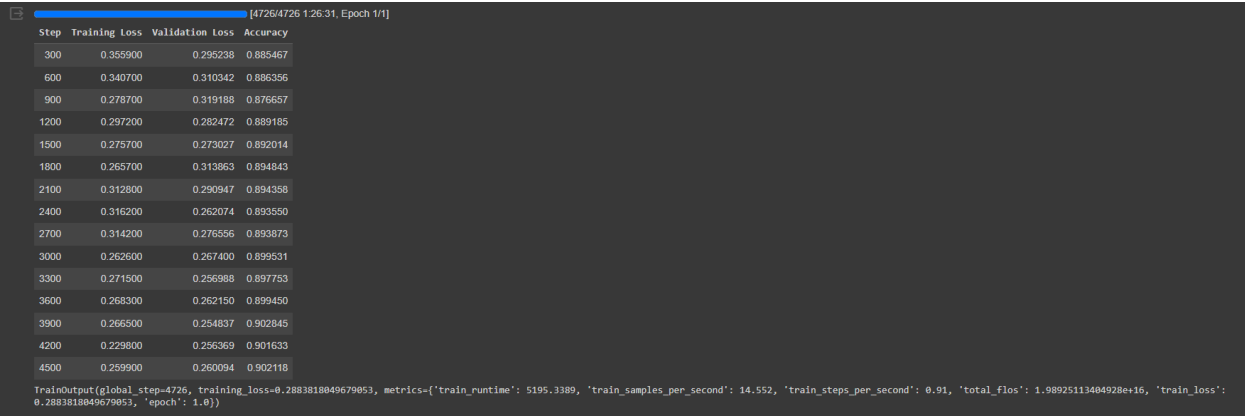
Warto zauważyć, że istnieje szereg innych narzędzi do monitorowania eksperymentów z treningiem sieci. Wśród nich dużą popularnością cieszą się [WanDB](#) oraz [Neptune.AI](#). Ich zaletą jest m.in. to, że możemy łatwo archiwizować przeprowadzone eksperymenty, porównywać je ze sobą, analizować wpływ hiperparametrów na uzyskane wyniki, itp.

```
%load_ext tensorboard
%tensorboard --logdir gdrive/MyDrive/poquad/output/runs

<IPython.core.display.Javascript object>
```

Uruchomienie procesu treningu jest już bardzo proste, po tym jak przygotowaliśmy wszystkie niezbędne szczegóły. Wystarczy wywołać metodę `trainer.train()`. Warto mieć na uwadze, że proces ten będzie jednak długotrwały - jedna epoka treningu na przygotowanych danych będzie trwała ponad 1 godzinę. Na szczęście, dzięki ustawieniu ewaluacji co 300 kroków, będziemy mogli obserwować jak model radzi sobie z postawionym przed nim problemem na danych ewaluacyjnych.

```
# trainer.train()
```



Step	Training Loss	Validation Loss	Accuracy
300	0.35990	0.295238	0.885467
600	0.340700	0.310342	0.886356
900	0.278700	0.319188	0.876657
1200	0.297200	0.282472	0.889185
1500	0.275700	0.273027	0.892014
1800	0.265700	0.313863	0.894843
2100	0.312800	0.290947	0.894358
2400	0.316200	0.262074	0.893550
2700	0.314200	0.276556	0.893873
3000	0.262600	0.267400	0.899531
3300	0.271500	0.256988	0.897753
3600	0.268300	0.262150	0.899450
3900	0.266500	0.254837	0.902845
4200	0.229800	0.256369	0.901633
4500	0.259900	0.260094	0.902118

TrainOutput(global_step=4726, training_loss=0.2883818849679853, metrics={'train_runtime': 5195.3389, 'train_samples_per_second': 14.552, 'train_steps_per_second': 0.91, 'total_flos': 1.98925113484928e+16, 'train_loss': 0.2883818849679853, 'epoch': 1.0})

Zadanie 3 (1 punkt)

Wybierz losową stronę z Wikipedii i skopiuj fragment tekstu do Notebook. Zadać 3 pytania, na które można udzielić odpowiedzi na podstawie tego fragmentu tekstu oraz 3 pytania, na które nie można udzielić odpowiedzi. Oceń jakość predykcji udzielanych przez model.

```
context="""W 2001 roku Małysz po raz pierwszy w karierze wygrał w  
Letniej Grand Prix. Już w pierwszym konkursie w Hinterzarten Polak  
odniósł zwycięstwo wyprzedzając o 4,5 punktów Martina Schmitta. W  
drugim konkursie na skoczni Adlerschanze w Hinterzarten Małysz był  
dziewiąty. W kolejnym konkursie, rozegranym w Courchevel na skoczni  
Tremplin Le Praz, skoczek z Wisły stanął na drugim stopniu podium,
```

przegrywając z Andreasem Widhölzlem o 1,1 punktu. 18 sierpnia w Stams Małysz uplasował się na dziewiątej pozycji. 5 września na skoczni w Sapporo był tuż za podium, na czwartym miejscu. 8 września w Hakubie był drugi, wygrał Stefan Horngacher. Dzień później, także w Hakubie, po raz trzeci zajął ósme miejsce, ale nie przeszkodziło mu to w zajęciu pierwszego miejsca w klasyfikacji generalnej."""

```
questions=("Kto wygrał pierwszy konkurs Letniej Grand Prix w 2001 roku  
w Hinterzarten",
```

```
            "Jaką pozycję zajął Małysz w konkursie na  
skoczni Tremplin Le Praz w Courchevel w 2001 roku?"
```

```
            , "W jaki dzień konkurs Letniej Grand Prix w  
2001 odbył się w Sapporo?",
```

```
            "Ile Małysz ma medali olimpijskich?", "W którym roku Małysz  
skończył karierę?", "Kto wygrał Letnie Grand Prix w 1999  
roku?")#without answer
```

```
import torch.nn.functional as F
```

```
import torch
```

```
from transformers import AutoModelForSequenceClassification
```

```
model = AutoModelForSequenceClassification.from_pretrained(  
    "/content/gdrive/MyDrive/poquad/output/checkpoint-4726",  
    num_labels=2  
)
```

```
def generate_question_context(questions, context):
```

```
    tuples=[]
```

```
    for question in questions:
```

```
        tuples.append({"text": f"Pytanie: {question} Kontekst:  
{context}"})
```

```
    return tuples
```

```
pl_tokenizer = AutoTokenizer.from_pretrained("allegro/herbert-base-  
cased")
```

```
def tokenize_function(examples):
```

```
    return pl_tokenizer(examples["text"], padding="max_length",  
    truncation=True)
```

```
tuples=generate_question_context(questions,context)
```

```
test_dataset = Dataset.from_list(tuples)
```

```
tokenized_test_dataset=test_dataset.map(tokenize_function,batched=True  
)
```

```
input_ids = tokenized_test_dataset["input_ids"]
```

```

attention_mask = tokenized_test_dataset["attention_mask"]

formatted_dataset = {"input_ids": torch.tensor(input_ids),
"attention_mask": torch.tensor(attention_mask)}

print(formatted_dataset["input_ids"].device)

print(model.device)
output=model(**formatted_dataset)
logits=output.logits

probabilities = F.softmax(logits, dim=1)

# Znajdź indeks klasy o najwyższym prawdopodobieństwie dla każdego przykładu
predicted_classes = torch.argmax(probabilities, dim=1)

print("Logits:")
print(logits)
print("\nProbabilities:")
print(probabilities)
print("\nPredicted Classes:")
print(predicted_classes)


{"model_id": "6b2ea8cae5374551a0cd59c7b18beef8", "version_major": 2, "version_minor": 0}

cpu
cpu
Logits:
tensor([[ -0.3006,  0.9464],
        [ -1.1559,  1.5161],
        [ -0.5173,  1.1176],
        [  0.7044, -0.7714],
        [  1.2171, -1.6116],
        [ -0.7631,  1.1946]], grad_fn=<AddmmBackward0>)

Probabilities:
tensor([[0.2232, 0.7768],
        [0.0646, 0.9354],
        [0.1631, 0.8369],
        [0.8139, 0.1861],
        [0.9442, 0.0558],
        [0.1237, 0.8763]], grad_fn=<SoftmaxBackward0>)

```

```
Predicted Classes:  
tensor([1, 1, 1, 0, 0, 1])
```

Model dla pytań z istniejącą odpowiedzią dał poprawny wynik. Dla pytań z innego kontekstu (4 i 5) nie potrafił znaleźć odpowiedzi, jednak ostatnie pytanie, na które nie dało się odpowiedzieć z kontekstu, ale było skonstruowane w specyficzny sposób, podobny do kontekstu, model zaklasyfikował jako takie, dla którego zna odpowiedź.

Odpowiadanie na pytania

Drugim problemem, którym zajmie się w tym laboratorium jest odpowiadanie na pytania. Zmierzymy się z wariantem tego problemu, w którym model sam formułuje odpowiedź, na podstawie pytania i kontekstu, w których znajduje się odpowiedź na pytanie (w przeciwieństwie do wariantu, w którym model wskazuje lokalizację odpowiedzi na pytanie).

Zadanie 4 (1 punkt)

Rozpocznij od przygotowania danych. Wybierzemy tylko te pytania, które posiadają odpowiedź (`is_impossible=False`). Uwzględnij zarówno pytania *pewne* (pole `answers`) jak i *prawdopodobne* (pole `plausible_answers`). Wynikowy zbiór danych powinien mieć identyczną strukturę, jak w przypadku zadania z klasyfikacją, ale etykiety zamiast wartości 0 i 1, powinny zawierać odpowiedź na pytanie, a sama nazwa etykiety powinna być zmieniona z `label` na `labels`, w celu odzwierciedlenia faktu, że teraz zwracane jest wiele etykiet.

Wyświetl liczbę danych (par: pytanie - odpowiedź) w zbiorze treningowym i zbiorze ewaluacyjnym.

Opakuj również zbiory w klasy z biblioteki `datasets` i zapisz je na dysku.

```
import random  
from datasets import Dataset, DatasetDict  
import random  
  
import json  
  
path = 'gdrive/MyDrive/poquad'  
  
with open(path + "/poquad-train.json") as input:  
    train_data = json.loads(input.read())["data"]  
  
# print(f"Train data articles: {len(train_data)}")  
  
with open(path + "/poquad-dev.json") as input:  
    dev_data = json.loads(input.read())["data"]
```

```

# print(f"Dev data articles: {len(dev_data)}")

# print(f"Train questions: {sum([len(e['paragraphs'])[0]['qas']) for e
in train_data])}")
# print(f"Dev questions: {sum([len(e['paragraphs'])[0]['qas']) for e in
dev_data])}")

tuples = [], []

for idx, dataset in enumerate([train_data, dev_data]):
    for data in dataset:
        context = data["paragraphs"][0]["context"]
        for question_answers in data["paragraphs"][0]["qas"]:
            question = question_answers["question"]
            if not question_answers["is_impossible"]:
                tuples[idx].append(
                    {
                        "text": f"Pytanie: {question} Kontekst:
{context}",
                        "labels": question_answers['answers'][0]
['generative_answer']
                    }
                )

train_tuples, dev_tuples = tuples
print(f"Total count in train/dev:
{len(train_tuples)}/{len(dev_tuples)}")
# print([e['labels'] for e in train_tuples])

train_dataset = Dataset.from_list(train_tuples)
dev_dataset = Dataset.from_list(dev_tuples)
datasets = DatasetDict({"train": train_dataset, "dev": dev_dataset})
datasets.save_to_disk(path + "/question-context-answers")

Total count in train/dev: 34028/5563

{"model_id": "cc12d789764d4a64a51b84549400b2f4", "version_major": 2, "vers
ion_minor": 0}

{"model_id": "b2691a3e32ec490ebbaec6b23df7504c", "version_major": 2, "vers
ion_minor": 0}

```

Zanim przejdziemy do dalszej części, sprawdźmy, czy dane zostały poprawnie utworzone. Zweryfikujmy przede wszystkim, czy klucze `text` oraz `label` zawierają odpowiednie wartości:

```
print(datasets["train"][0]["text"])
print(datasets["train"][0]["labels"])
print(datasets["dev"][0]["text"])
print(datasets["dev"][0]["labels"])
```

Pytanie: Co było powodem powrócenia konceptu porozumienia monachijskiego? Kontekst: Projekty konfederacji zaczęły się załamywać 5 sierpnia 1942. Ponownie wróciła kwestia monachijska, co uaktywniło się wymianą listów Ripka – Stroński. Natomiast 17 sierpnia 1942 doszło do spotkania E. Beneša i J. Masaryka z jednej a Wł. Sikorskiego i E. Raczyńskiego z drugiej strony. Polscy dyplomaci zaproponowali podpisanie układu konfederacyjnego. W następnym miesiącu, tj. 24 września, strona polska przesłała na ręce J. Masaryka projekt deklaracji o przyszłej konfederacji obu państw. Strona czechosłowacka projekt przyjęła, lecz już w listopadzie 1942 E. Beneš podważył ideę konfederacji. W zamian zaproponowano zawarcie układu sojuszniczego z Polską na 20 lat (formalnie nastąpiło to 20 listopada 1942).

wymiana listów Ripka – Stroński

Pytanie: Czym są pisma rabiniczne? Kontekst: Pisma rabiniczne – w tym Miszna – stanowią kompilację poglądów różnych rabinów na określony temat. Zgodnie z wierzeniami judaizmu Mojżesz otrzymał od Boga całą Torę, ale w dwóch częściach: jedną część w formie pisanej, a drugą część w formie ustnej. Miszna – jako Tora ustna – była traktowana nie tylko jako uzupełnienie Tory spisanej, ale również jako jej interpretacja i wyjaśnienie w konkretnych sytuacjach życiowych. Tym samym Miszna stanowiąca kodeks Prawa religijnego zaczęła równocześnie służyć za jego ustnie przekazywany podręcznik. kompilacją poglądów różnych rabinów na określony temat

Tokenizacja danych dla problemu odpowiadania na pytania jest nieco bardziej problematyczna. W pierwszej kolejności trzeba wziąć pod uwagę, że dane wynikowe (etykiety), też muszą podlegać tokenizacji. Realizowane jest to poprzez wywołanie tokenizera, z opcją `text_target` ustawioną na łańcuch, który ma być stokenizowany.

Ponadto wcześniej nie przejmowaliśmy się za bardzo tym, czy wykorzystywany model obsługuje teksty o założonej długości. Teraz jednak ma to duże znaczenie. Jeśli użyjemy modelu, który nie jest w stanie wygenerować odpowiedzi o oczekiwanej długości, to nie możemy oczekiwać, że model ten będzie dawał dobre rezultaty dla danych w zbiorze treningowym i testowym.

W pierwszej kolejności dokonamy więc tokenizacji bez ograniczeń co do długości tekstu. Ponadto, stokenizowane odpowiedzi przypiszemy do klucza `label`. Do tokenizacji użyjemy tokenizera stowarzyszonego z modelem `allegro/plt5-base`.

```
!pip install sentencepiece
```

```
Requirement already satisfied: sentencepiece in
/usr/local/lib/python3.10/dist-packages (0.1.99)
```

```

from transformers import AutoTokenizer

plt5_tokenizer = AutoTokenizer.from_pretrained("allegro/plt5-base")

def preprocess_function(examples):
    model_inputs = plt5_tokenizer(examples["text"])
    labels = plt5_tokenizer(text_target=examples["labels"])
    model_inputs["labels"] = labels["input_ids"]
    return model_inputs

tokenized_datasets = datasets.map(preprocess_function, batched=True)

{"model_id": "7294828576c34272bbb812e3385cebe", "version_major": 2, "version_minor": 0}

{"model_id": "65df41578fca4ae19b7995557c2613ae", "version_major": 2, "version_minor": 0}

{"model_id": "872e86d811a24a9f9ae34f3c971780cd", "version_major": 2, "version_minor": 0}

{"model_id": "f5e1194984d549289d9aec8685071250", "version_major": 2, "version_minor": 0}

You are using the default legacy behaviour of the <class
'transformers.models.t5.tokenization_t5.T5Tokenizer'>. This is
expected, and simply means that the `legacy` (previous) behavior will
be used so nothing changes for you. If you want to use the new
behaviour, set `legacy=False`. This should only be set if you
understand what it means, and thoroughly read the reason why this was
added as explained in
https://github.com/huggingface/transformers/pull/24565

{"model_id": "099fdcdfb5de48688bdbfa3c38d634ef", "version_major": 2, "version_minor": 0}

{"model_id": "02bleedb88d6472aa6c7951f9fc11d56", "version_major": 2, "version_minor": 0}

```

Sprawdźmy jak dane wyglądają po tokenizacji:

```

print(tokenized_datasets["train"][0].keys())
print(tokenized_datasets["train"][0]["input_ids"])
print(tokenized_datasets["train"][0]["labels"])
print(len(tokenized_datasets["train"][0]["input_ids"]))
print(len(tokenized_datasets["train"][0]["labels"]))
example = tokenized_datasets["train"][0]

print("|".join(plt5_tokenizer.convert_ids_to_tokens(list(example["input_ids"]))))

```



```

t_ids"])))
print("|".join(plt5_tokenizer.convert_ids_to_tokens(list(example["labels"]))))

dict_keys(['text', 'labels', 'input_ids', 'attention_mask'])
[21584, 291, 639, 402, 11586, 292, 23822, 267, 1269, 8741, 280, 24310,
42404, 305, 373, 1525, 15643, 291, 2958, 273, 19605, 6869, 271, 298,
2256, 7465, 394, 540, 2142, 259, 17542, 13760, 10331, 9511, 322,
31220, 261, 358, 348, 267, 7243, 430, 470, 271, 39908, 20622, 2178,
18204, 308, 8439, 2451, 259, 1974, 455, 540, 2142, 1283, 272, 994,
525, 259, 15697, 1978, 267, 264, 644, 259, 14988, 19434, 265, 1109,
287, 274, 357, 259, 21308, 264, 525, 259, 35197, 305, 265, 793, 823,
259, 25318, 2750, 4724, 31015, 21207, 4162, 40335, 18058, 259, 274,
4862, 7030, 261, 5269, 259, 658, 497, 261, 6971, 1890, 35042, 267,
266, 3260, 644, 259, 14988, 19434, 1187, 20919, 284, 27584, 19605,
1230, 2555, 259, 12531, 7278, 3845, 8726, 10486, 1187, 10676, 261,
996, 347, 260, 2548, 2142, 525, 259, 15697, 1978, 309, 27648, 31887,
19605, 259, 274, 4931, 36525, 37011, 4162, 10036, 7141, 265, 6340,
266, 465, 346, 269, 3648, 4383, 6704, 294, 465, 567, 2142, 454, 1]
[13862, 20622, 2178, 18204, 308, 8439, 2451, 1]
165
8
_Pytanie|:_Co|było|powodem|po|wrócenia|kon|ceptu|porozumieniu|
_monachijskiego|?|_Kon|tekst|:_Projekt|y|konfederacji|zaczęły|
_się|za|łamywać|_5|sierpnia|_1942|.|_Ponownie|wróciła|kwestia|
_mon|ach|ijska|,|_co|u|a|ktyw|ni|ło|się|wymianą|listów|_Ri|pka|_|
_Stro|ński|.|_Natomiast|_17|sierpnia|_1942|doszło|do|spotkania|
_E|.|_Bene|ś|a|_i|_J|.|_Masa|ryka|_z|jednej|a|_W|ł|.|_Sikorskiego|
_i|_E|.|_Raczyński|ego|_z|drugiej|strony|.|_Polscy|dyplom|acji|
_zaproponowali|podpisanie|układu|konfederac|yjnego|.|_W|_następnym|
_miesiącu|,|_tj|.|_24|września|,|_strona|_polska|_przesłał|a|_na|
_ręce|_J|.|_Masa|ryka|_projekt|_deklaracji|_o|_przyszłej|
_konfederacji|_obu|_państw|.|_Strona|_cze|cho|słow|acka|_projekt|
_przyjęła|,|_lecz|_już|_w|_listopadzie|_1942|_E|.|_Bene|ś|_pod|ważył|
_ideę|_konfederacji|.|_W|_zamian|_zaproponowano|_zawarcie|_układu|
_sojusz|niczego|_z|_Polską|_na|_20|_lat|_(|form|alnie|_nastąpiło|_to|
_20|_listopada|_1942|)|.|</s>
_wymiana|_listów|_Ri|pka|_|_|_Stro|ński|</s>

```

Wykorzystywany przez nas model obsługuje teksty od długości do 512 sub-tokenów (w zasadzie ograniczenie to, w przeciwieństwie do modelu BERT nie wynika z samego modelu, więc teoretycznie moglibyśmy wykorzystywać dłuższe sekwencje, co jednak prowadzi do nadmiernej konsumpcji pamięci). Konieczne jest zatem sprawdzenie, czy w naszych danych nie ma tekstów o większej długości.

Zadanie 5 (0.5 punkt)

Stwórz histogramy prezentujące rozkład długości (jako liczby tokenów) tekstów wejściowych (`input_ids`) oraz odpowiedzi (`labels`) dla zbioru treningowego. Zinterpretuj otrzymane wyniki.

```
import matplotlib.pyplot as plt
import numpy as np

input_lengths = [len(example["input_ids"]) for example in
tokenized_datasets["train"]]

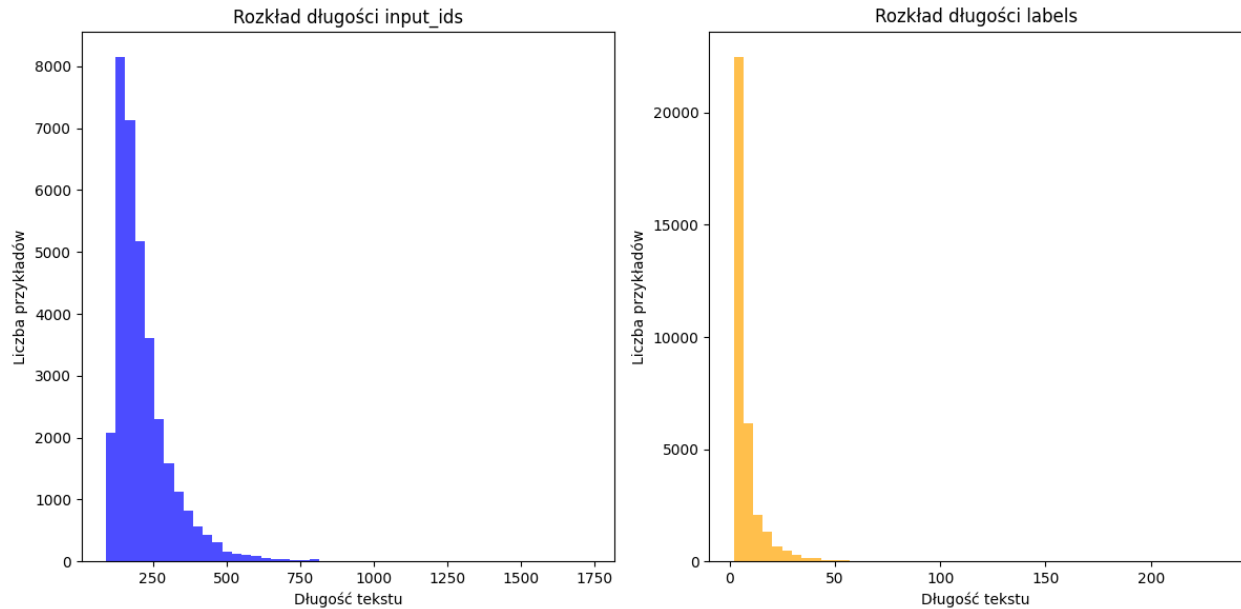
# Długości odpowiedzi
label_lengths = [len(example["labels"]) for example in
tokenized_datasets["train"]]

# Tworzenie histogramów
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.hist(input_lengths, bins=50, color='blue', alpha=0.7)
plt.title('Rozkład długości input_ids')
plt.xlabel('Długość tekstu')
plt.ylabel('Liczba przykładów')

plt.subplot(1, 2, 2)
plt.hist(label_lengths, bins=50, color='orange', alpha=0.7)
plt.title('Rozkład długości labels')
plt.xlabel('Długość tekstu')
plt.ylabel('Liczba przykładów')

plt.tight_layout()
plt.show()
```



Przyjmijmy założenie, że teksty wejściowe będą miały maksymalnie 256 tokenów, a większość odpowiedzi jest znacznie krótsza niż maksymalna długość, ograniczmy je do długości 32.

W poniższym kodzie uwzględniamy również fakt, że przy obliczaniu funkcji straty nie interesuje nas wliczanie tokenów wypełnienia (PAD), gdyż ich udział byłby bardzo duży, a nie wpływają one w żaden pozytywny sposób na ocenę poprawności działania modelu.

Konteksty (pytanie + kontekst odpowiedzi) ograniczamy do 256 tokenów, ze względu na ograniczenia pamięciowe (zajętość pamięci dla modelu jest proporcjonalna do kwadratu długości tekstu). Dla kontekstów nie używamy parametru `padding`, ponieważ w trakcie treningu użyjemy modułu, który automatycznie doda padding, tak żeby wszystkie sekwencje miały długość najdłuższego tekstu w ramach paczki (moduł ten to `DataCollatorWithPadding`).

```
def preprocess_function(examples):
    result = plt5_tokenizer(examples["text"], truncation=True,
max_length=256)
    targets = plt5_tokenizer(
        examples["labels"], truncation=True, max_length=32,
padding=True
    )
    input_ids = [
        [(l if l != plt5_tokenizer.pad_token_id else -100) for l in e]
        for e in targets["input_ids"]
    ]
    result["labels"] = input_ids
    return result
```

```
tokenized_datasets = datasets.map(preprocess_function, batched=True)
```

```
{
  "model_id": "1c5d798a1a264cf48cf262576c2972fe",
  "version_major": 2,
  "version_minor": 0
}

{
  "model_id": "697dc237a07e46078462a362866cb43f",
  "version_major": 2,
  "version_minor": 0
}
```

Następnie weryfikujemy, czy przetworzone teksty mają poprawną postać.

```
print(tokenized_datasets["train"][0].keys())
print(tokenized_datasets["train"][0]["input_ids"])
print(tokenized_datasets["train"][0]["labels"])
print(len(tokenized_datasets["train"][0]["input_ids"]))
print(len(tokenized_datasets["train"][0]["labels"]))

dict_keys(['text', 'labels', 'input_ids', 'attention_mask'])
[21584, 291, 639, 402, 11586, 292, 23822, 267, 1269, 8741, 280, 24310,
42404, 305, 373, 1525, 15643, 291, 2958, 273, 19605, 6869, 271, 298,
2256, 7465, 394, 540, 2142, 259, 17542, 13760, 10331, 9511, 322,
31220, 261, 358, 348, 267, 7243, 430, 470, 271, 39908, 20622, 2178,
18204, 308, 8439, 2451, 259, 1974, 455, 540, 2142, 1283, 272, 994,
525, 259, 15697, 1978, 267, 264, 644, 259, 14988, 19434, 265, 1109,
287, 274, 357, 259, 21308, 264, 525, 259, 35197, 305, 265, 793, 823,
259, 25318, 2750, 4724, 31015, 21207, 4162, 40335, 18058, 259, 274,
4862, 7030, 261, 5269, 259, 658, 497, 261, 6971, 1890, 35042, 267,
266, 3260, 644, 259, 14988, 19434, 1187, 20919, 284, 27584, 19605,
1230, 2555, 259, 12531, 7278, 3845, 8726, 10486, 1187, 10676, 261,
996, 347, 260, 2548, 2142, 525, 259, 15697, 1978, 309, 27648, 31887,
19605, 259, 274, 4931, 36525, 37011, 4162, 10036, 7141, 265, 6340,
266, 465, 346, 269, 3648, 4383, 6704, 294, 465, 567, 2142, 454, 1]
[13862, 20622, 2178, 18204, 308, 8439, 2451, 1, -100, -100, -100, -
100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
-100, -100, -100, -100, -100, -100, -100, -100, -100]
165
32
```

Dla problemu odpowiadania na pytania potrzebować będziemy innego pre-trenowanego modelu oraz innego przygotowania danych. Jako model bazowy wykorzystamy polski wariant modelu T5 - [pLT5](#). Model ten trenowany był w zadaniu *span corruption*, czyli zadani polegającym na usunięciu fragmentu tekstu. Model na wejściu otrzymywał tekst z pominiętymi pewnymi fragmentami, a na wyjściu miał odtwarzać te fragmenty. Oryginalny model T5 dodatkowo pre-trenowany był na kilku konkretnych zadaniach z zakresu NLP (w tym odpowiadaniu na pytania). W wariantcie pLT5 nie przeprowadzono jednak takiego dodatkowego procesu.

Poniżej ładujemy model dla zadania, w którym model generuje tekst na podstawie innego tekstu (tzn. jest to zadanie zamiany tekstu na tekst, po angielsku zwanego też *Sequence-to-Sequence*).

```
from transformers import AutoModelForSeq2SeqLM

model = AutoModelForSeq2SeqLM.from_pretrained("allegro/plt5-base")
```

```
{"model_id": "2bc79ee8159744b1922771695dc31a09", "version_major": 2, "version_minor": 0}
```

Trening modelu QA

Ostatnim krokiem przed uruchomieniem treningu jest zdefiniowanie metryk, wskazujących jak model radzi sobie z problemem. Wykorzystamy dwie metryki:

- *exact match* - która sprawdza dokładne dopasowanie odpowiedzi do wartości referencyjnej, metryka ta jest bardzo restrykcyjna, ponieważ pojedynczy znak będzie powodował, że wartość będzie niepoprawna,
- *blue score* - metryka uwzględniająca częściowe dopasowanie pomiędzy odpowiedzią a wartością referencyjną, najczęściej używana jest do oceny maszynowego tłumaczenia tekstu, ale może być również przydatna w ocenie wszelkich zadań, w których generowany jest tekst.

Wykorzystujemy bibliotekę `evaluate`, która zawiera definicje obu metryk.

Przy konwersji identyfikatorów tokenów na tekst zamieniamy również z powrotem tokeny o wartości -100 na identyfikatory paddingu. W przeciwnym razie dostaniemy błąd o nieistniejącym identyfikatorze tokenu.

W procesie treningu pokazujemy również różnicę między jedną wygenerowaną oraz prawdziwą odpowiedzią dla zbioru ewaluacyjnego. W ten sposób możemy śledzić co rzeczywiście dzieje się w modelu.

```
from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments
import numpy as np
import evaluate

exact = evaluate.load("exact_match")
bleu = evaluate.load("bleu")

def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = np.where(predictions != -100, predictions,
plt5_tokenizer.pad_token_id)
    decoded_preds = plt5_tokenizer.batch_decode(predictions,
skip_special_tokens=True)
    labels = np.where(labels != -100, labels,
plt5_tokenizer.pad_token_id)
    decoded_labels = plt5_tokenizer.batch_decode(labels,
skip_special_tokens=True)
    print("prediction: " + decoded_preds[0])
    print("reference : " + decoded_labels[0])

    result = exact.compute(predictions=decoded_preds,
references=decoded_labels)
    result = {**result, **bleu.compute(predictions=decoded_preds,
```

```

references=decoded_labels)}}
    del result["precisions"]

    prediction_lens = [np.count_nonzero(pred !=
plt5_tokenizer.pad_token_id) for pred in predictions]
    result["gen_len"] = np.mean(prediction_lens)

    return result

{"model_id":"3330d133beb04bce88ecf3ac27db7584","version_major":2,"version_minor":0}

{"model_id":"75b0c795e8894f529c93b362f6add377","version_major":2,"version_minor":0}

{"model_id":"14f5d1971eba4edeabca85dca7a6ed40","version_major":2,"version_minor":0}

{"model_id":"e43e9a69c9a544bda37e395e0c06cff7","version_major":2,"version_minor":0}

```

Zadanie 6 (0.5 punkty)

Korzystając z klasy Seq2SeqTrainingArguments zdefiniuj następujące parametry trenignu:

- inny katalog z wynikami
- liczba epok: 3
- wielkość paczki: 16
- ewaluacja co 100 kroków,
- szybkość uczenia: 1e-4
- optymalizator: adafactor
- maksymalna długość generowanej odpowiedzi: 32,
- akumulacja wyników ewaluacji: 4
- generowanie wyników podczas ewaluacji

W treningu nie używamy optymalizacji FP16! Jej użycie spowoduje, że model nie będzie się trenował. Jeśli chcesz użyć optymalizacji, to możesz skorzystać z **BF16**.

Argumenty powinny również wskazywać, że przeprowadzono jest proces uczenia i ewaluacji.

```

# your_code
from transformers import Seq2SeqTrainingArguments
import numpy as np

arguments = Seq2SeqTrainingArguments(
    output_dir=path + "/output_for_answers_model",
    do_train=True,
    do_eval=True,
    num_train_epochs=3,

```

```

per_device_train_batch_size=16,
per_device_eval_batch_size=16,
evaluation_strategy="steps",
eval_steps=100,
optim="adafactor",
generation_max_length=32,
learning_rate=1e-4,
gradient_accumulation_steps=4,
logging_first_step=True,
logging_strategy="steps",
logging_steps=50,
save_strategy="epoch",
# bf16=True,
)

```

Zadanie 7 (0.5 punktu)

Utwórz obiekt trenujący `Seq2SeqTrainer`, za pomocą którego będzie trenowany model odpowiadający na pytania.

Obiekt ten powinien:

- wykorzystywać model `plt5-base`,
- wykorzystywać zbiór `train` do treningu,
- wykorzystywać zbiór `dev` do ewaluacji,
- wykorzystać klasę batchującą (`data_collator`) o nazwie `DataCollatorWithPadding`.

```

from transformers import DataCollatorWithPadding

# your_code
trainer = Seq2SeqTrainer(
    model=model,
    args=arguments,
    train_dataset=tokenized_datasets["train"].shuffle(seed=42),
    eval_dataset=tokenized_datasets["dev"].shuffle(seed=42),
    compute_metrics=compute_metrics,
    data_collator = DataCollatorWithPadding(tokenizer=plt5_tokenizer)
)

%load_ext tensorboard
%tensorboard --logdir gdrive/MyDrive/poquad/output_qa/runs

The tensorboard extension is already loaded. To reload it, use:
%reload_ext tensorboard

<IPython.core.display.Javascript object>

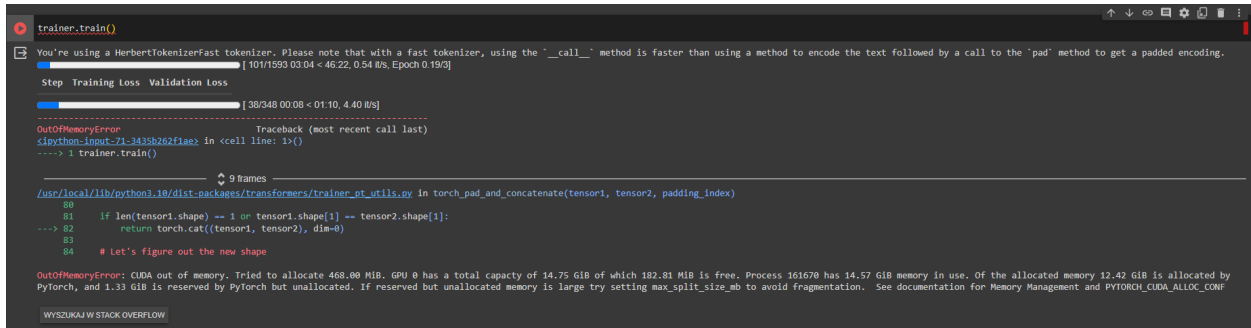
```

Mając przygotowane wszystkie dane wejściowe możemy rozpocząć proces treningu.

Uwaga: proces treningu na Google Colab z wykorzystaniem akceleratora zajmuje ok. 3 godziny. Uruchomienie treningu na CPU może trwać ponad 1 dzień!

Możesz pominąć ten proces i w kolejnych krokach wykorzystać gotowy model `apohllo/plt5-base-poquad`, który znajduje się w repozytorium Huggingface.

```
# trainer.train()
```



Zadanie 8 (1.5 punkt)

Korzystając z wywołania `generate` w modelu, wygeneruj odpowiedzi dla 1 kontekstu i 10 pytań dotyczących tego kontekstu. Pamiętaj aby zamienić identyfikatory tokenów na ich treść. Możesz do tego wykorzystać wywołanie `decode` z tokenizera.

Jeśli w poprzednim punkcie nie udało Ci się wytrenować modelu, możesz skorzystać z modelu `apohllo/plt5-base-poquad`.

Oceń wyniki (odpowiedzi) generowane przez model.

```
model = AutoModelForSeq2SeqLM.from_pretrained("apohllo/plt5-base-poquad")

{"model_id": "f46552963fc14783b6bbc0ac1b6de950", "version_major": 2, "version_minor": 0}

{"model_id": "84941d465db54b4682fd866873bb088d", "version_major": 2, "version_minor": 0}

{"model_id": "9db352ea6aad425c80da082ca56f5e7a", "version_major": 2, "version_minor": 0}

# your_code
```

context="8 czerwca do Grand Prix Kanady Kubica wystartował z 2. miejsca. Zdołał obronić tę pozycję na starcie. Po wyjeździe samochodu bezpieczeństwa na 17. okrążeniu zniwelowana została jego różnica do pierwszego Hamiltona, a także trzeciego Räikkönena. Dwa okrążenia później zjechał do pit-stopu, gdzie założono mu kolejny komplet

miękkich opon. Na wyjeździe z alei serwisowej zderzyli się Räikkönen i Hamilton, co spowodowało ich odpadnięcie z wyścigu. Polak jechał na ósmym miejscu, będąc pierwszym kierowcą po pit-stopie. Na 23. okrążeniu próbował wyprzedzić Sebastiana Vettela. Siedem okrążeń później wyprzedził Nicka Heidfelda, a kolejni kierowcy zaczęli swoje zjazdy, więc Kubica został liderem na 42. okrążeniu. Był najszybszy na torze, dzięki czemu wypracował sobie bezpieczną przewagę. Na 49. okrążeniu po raz drugi zjechał do alei serwisowej, gdzie założono mu supermiękką mieszankę. Do mety dojechał pierwszy. Było to pierwsze zwycięstwo w jego karierze, a także pierwszy triumf BMW Sauber. Było to także pierwsze zwycięstwo polskiego kierowcy w wyścigu Grand Prix i jednocześnie także pierwsze zwycięstwo kierowcy pochodzącego z Europy Środkowo-Wschodniej w historii Formuły 1. Kubica został również liderem klasyfikacji generalnej kierowców, awansując z czwartego miejsca"

```
questions = (  
    "Kiedy i gdzie Kubica odniósł swoje pierwsze zwycięstwo?",  
    "Po co kierowcy zjeżdżają do alei serwisowej?",  
    "Jak Kubica poradził sobie na starcie wyścigu?",  
    "Co spowodowało znielowanie różnicy Kubicy do pierwszego  
Hamiltona podczas wyścigu?",  
    "Dlaczego Räikkönen i Hamilton odpadli z wyścigu?",  
    "Z jakiej części europy pochodził Kubica?",  
    "Kiedy Kubica został liderem wyścigu?",  
    "Jak Kubica wypracował sobie przewagę na torze?",  
    "Czy Kubica wygrał Grand Prix w swojej karierze?",  
    "Jakie konsekwencje zwycięstwo Kubicy miało dla klasyfikacji  
generalnej kierowców?"  
)
```

```
def generate_question_context(questions, context):  
    tuples=[]  
    for question in questions:  
        tuples.append({"text": f"Pytanie: {question} Kontekst:  
{context}"})  
    return tuples
```

```
def tokenize_function(examples):  
    return plt5_tokenizer(examples["text"],  
padding="max_length",return_tensors="pt", max_length=256,  
truncation=True,)
```

```
tuples=generate_question_context(questions,context)  
test_dataset = Dataset.from_list(tuples)
```

```
tokenized_test_dataset=test_dataset.map(tokenize_function,batched=True  
)  
input_ids = tokenized_test_dataset["input_ids"]
```

```

attention_mask = tokenized_test_dataset["attention_mask"]

formatted_dataset = {"input_ids": torch.tensor(input_ids),
"attention_mask": torch.tensor(attention_mask)}

output=model.generate(**formatted_dataset,max_length=50)
decoded_tokens = [plt5_tokenizer.decode(tokens,
skip_special_tokens=True) for tokens in output]

for i, tokens in enumerate(decoded_tokens):
    print(f"Pytanie: {questions[i]} Odpowiedź: {tokens}")
#zmienic pytania

{"model_id":"c3283657a9f741bd8ea9aca6a3cfd41f","version_major":2,"version_minor":0}

```

Pytanie: Kiedy i gdzie Kubica odniósł swoje pierwsze zwycięstwo?
Odpowiedź: w wyścigu Grand Prix
Pytanie: Po co kierowcy zjeżdżają do alei serwisowej? Odpowiedź:
założono mu supermiękką mieszankę
Pytanie: Jak Kubica poradził sobie na starcie wyścigu? Odpowiedź:
Zdołał obronić tę pozycję
Pytanie: Co spowodowało zniwelowanie różnicy Kubicy do pierwszego
Hamiltona podczas wyścigu? Odpowiedź: wyjazd samochodu bezpieczeństwa
Pytanie: Dlaczego Räikkönen i Hamilton odpadli z wyścigu? Odpowiedź:
Na wyjeździe z alei serwisowej zderzyli się Räikkönen i Hamilton
Pytanie: Z jakiej części europy pochodził Kubica? Odpowiedź: środkowo-
wschodniej
Pytanie: Kiedy Kubica został liderem wyścigu? Odpowiedź: na 42.
okrążeniu
Pytanie: Jak Kubica wypracował sobie przewagę na torze? Odpowiedź: Był
najszybszy
Pytanie: Czy Kubica wygrał Grand Prix w swojej karierze? Odpowiedź:
tak
Pytanie: Jakie konsekwencje zwycięstwo Kubicy miało dla klasyfikacji
generalnej kierowców? Odpowiedź: był najszybszy na torze

Niestety nie udało się wytrenować własnego modelu ponieważ Colab zablokował dostęp do GPU. Dla gotowego modelu wyniki odpowiedzi są dobre. Na drugie pytanie odpowiada w dobrym kontekście ale w złej formie , a na ostatnie odpowiada całkiem źle, a pierwsze jest zbyt ogólne. Pozostałe odpowiedzi są bardzo dobre.

Zadanie dodatkowe (2 punkty)

Stworzenie pełnego rozwiązania w zakresie odpowiadania na pytania wymaga również znajdowania kontekstów, w których może pojawić się pytanie.

Obenie istnieje coraz więcej modeli neuronalnych, które bardzo dobrze radzą sobie ze znajdowaniem odpowiednich tekstów. Również dla języka polskiego następuje tutaj istotny postęp. Powstała m.in. [strona śledząca postępy w tym zakresie](#).

Korzystając z informacji na tej stronie wybierz jeden z modeli do wyszukiwania kontekstów (najlepiej o rozmiarze `base` lub `small`). Zamień konteksty występujące w zbiorze PoQuAD na reprezentacje wektorowe. To samo zrób z pytaniami występującymi w tym zbiorze. Dla każdego pytania znajdź kontekst, który według modelu najlepiej odpowiada na zadane pytanie. Do znalezienia kontekstu oblicz iloczyn skalarny pomiędzy reprezentacją pytania oraz wszystkimi kontekstami ze zbioru. Następnie uruchom model generujący odpowiedź na znalezionym kontekście. Porównaj wyniki uzyskiwane w ten sposób, z wynikami, gdy poprawny kontekst jest znany.

W celu przyspieszenia obliczeń możesz zmniejszyć liczbę pytań i odpowiadających im kontekstów. Pamiętaj jednak, żeby liczba kontekstów była odpowiednio duża (sugerowana wartość min. to 1000 kontekstów), tak żeby znalezienie kontekstu nie było trywialne.

