Przetwarzanie języka naturalnego

Wstęp

Obecnie najpopularniejsze model służące do przetwarzania języka naturalnego wykorzystują architekturę transformacyjną. Istnieje kilka bibliotek, implementujących tę architekturę, ale w kontekście NLP najczęściej wykorzystuje się Huggingface transformers.

Biblioteka ta poza samym kodem źródłowym, zawiera szereg innych elementów. Do najważniejszych z nich należą:

- modele - olbrzymia i ciągle rosnąca liczba gotowych modeli, których możemy użyć do rozwiązywania wielu problemów z dziedziny NLP (ale również w zakresie rozpoznawania mowy, czy przetwarzania obrazu),
- zbiory danych - bardzo duży katalog przydatnych zbiorów danych, które możemy w prosty sposób wykorzystać do trenowania własnych modeli NLP (oraz innych modeli).

Weryfikacja dostępności GPU

Trening modeli NLP wymaga dostępu do akceleratorów sprzętowych, przyspieszających uczenie sieci neuronowych. Jeśli nasz komputer nie jest wyposażony w GPU, to możemy skorzystać ze środowiska Google Colab.

Open in Colab

W tym środowisku możemy wybrać akcelerator spośród GPU i TPU.

Sprawdźmy, czy mamy dostęp do środowiska wyposażonego w akcelerator NVidii:

```
!nvidia-smi
```

```
Mon Jan 22 13:42:44 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Versi
|-----------------------------------------+----------------------+----------
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
|                                         |                      |
|=========================================+======================+==========
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
| N/A   55C    P8              10W /  70W |      0MiB / 15360MiB |      0%
|                                         |                      |
+-----------------------------------------+----------------------+----------

+---------------------------------------------------------------------------------------+
| Processes:
|  GPU   GI   CI        PID   Type   Process name
|        ID   ID
|=======================================================================================
|  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==
```

```
Collecting evaluate==0.4.1
  Downloading evaluate-0.4.1-py3-none-any.whl (84 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.1/84.1 kB 13.6 MB/s eta 0:
Collecting sentencepiece==0.1.99
  Downloading sentencepiece-0.1.99-cp310-cp310-manylinux_2_17_x86_64.manyli
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 20.2 MB/s eta 0:00
Collecting accelerate==0.24.1
  Downloading accelerate-0.24.1-py3-none-any.whl (261 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 261.4/261.4 kB 18.4 MB/s eta
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: huggingface-hub<1.0,>=0.16.4 in /usr/local/l
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dis
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dis
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: tokenizers<0.19,>=0.14 in /usr/local/lib/pyt
Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/python3
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: joblib in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: pyarrow>=8.0.0 in /usr/local/lib/python3.10/
Requirement already satisfied: pyarrow-hotfix in /usr/local/lib/python3.10/
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 16.8 MB/s eta
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: xxhash in /usr/local/lib/python3.10/dist-pac
Collecting multiprocess (from datasets==2.15.0)
  Downloading multiprocess-0.70.15-py310-none-any.whl (134 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 18.7 MB/s eta
Requirement already satisfied: fsspec[http]<=2023.10.0,>=2023.1.0 in /usr/l
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-pa
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-pac
Requirement already satisfied: torch>=1.10.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.1
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/py
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/p
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/di
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/pyt
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/di
Installing collected packages: sentencepiece, sacremoses, dill, responses,
Successfully installed accelerate-0.24.1 datasets-2.15.0 dill-0.3.7 evaluat
```

Mając zainstalowane niezbedne bilioteki, 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 trenowywany, 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")
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
  warnings.warn(
config.json: 100%                                570/570 [00:00<00:00, 12.9kB/s]
model.safetensors:                               436M/436M [00:03<00:00,
100%                                             145MB/s]
Some weights of the model checkpoint at bert-base-cased were not used when
- This IS expected if you are initializing BertForMaskedLM from the checkpo
- This IS NOT expected if you are initializing BertForMaskedLM from the che
```

Załadowany model jest modułem PyTorcha. 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(',', ' ')
```

```
'108 340 804'
```

Widzimi 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
```

```
tokenizer_config.json: 100% 29.0/29.0 [00:00<00:00, 857B/s]
vocab.txt: 100% 213k/213k [00:00<00:00, 2.73MB/s]
tokenizer.json: 100% 436k/436k [00:00<00:00, 2.21MB/s]
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),
```

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,  36
           119,   102]])
torch.Size([1, 12])
tensor([[  101,   163,  1161, 28259,  7774, 20671,  7128,   176, 28221, 282
         1233, 28213,   179,  1161, 28257, 19339,   119,   102]])
torch.Size([1, 18])
```

Korzystając z tokenizera dla języka angielsiego do podziału polskiego zdania, widzimy, że otrzymujemy znacznie większą liczbę tokenów. Żeby zobaczyć, w jaki sposób tokenizer dokonał podziału tekstu, możemy wykorzystać wywołanie `covert_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|##l|##ą|j|##a|##ź|##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 porzedzającym 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
tokenizer2 = AutoTokenizer.from_pretrained("allegro/herbert-base-cased")
```

```
tokenizer_config.json: 100% 229/229 [00:00<00:00, 10.4kB/s]
config.json: 100% 472/472 [00:00<00:00, 25.0kB/s]
vocab.json: 100% 907k/907k [00:00<00:00, 12.5MB/s]
merges.txt: 100% 556k/556k [00:00<00:00, 2.13MB/s]
special_tokens_map.json: 100% 120/120 [00:00<00:00
```

```
sentence1 = tokenizer2.encode(
    "The quick brown fox jumps over the lazy dog.", return_tensors="pt"
)
print(sentence1)
print(sentence1.shape)

sentence2 = tokenizer2.encode("Zażółć gęślą jaźń.", return_tensors="pt")
print(sentence2)
print(sentence2.shape)
print("|".join(tokenizer2.convert_ids_to_tokens(list(sentence1[0]))))
print("|".join(tokenizer2.convert_ids_to_tokens(list(sentence2[0]))))
```

```
tensor([[    0,  7117, 22991,  4879, 25015,  1016,  3435,  1055,  2202,  49
          1010,    83, 10259,  6854,  2050,  3852,  2065,  1031,  1899,
torch.Size([1, 20])
tensor([[    0,  2237,  7227,  1048,  7029, 46389,  2059,   272,  1059,  18
            2]])
torch.Size([1, 11])
<s>|The</w>|qui|ck</w>|brow|n</w>|fo|x</w>|ju|mp|s</w>|o|ver</w>|the</w>|la
<s>|Za|żół|ć</w>|gę|ślą</w>|ja|ź|ń</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 dystrubowanym 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
```

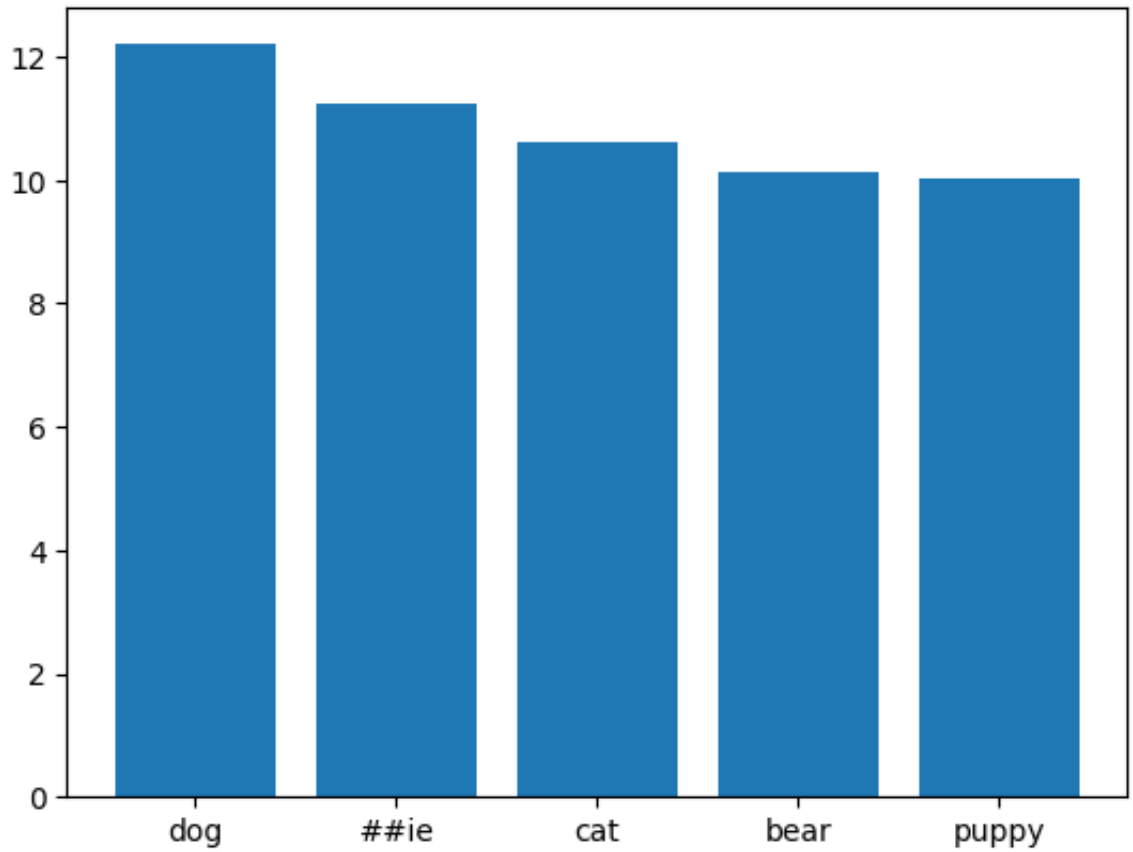
```
torch.return_types.topk(
values=tensor([12.1982, 11.2289, 10.6009, 10.1278, 10.0120], grad_fn=
<TopkBackward0>),
indices=tensor([ 3676,  1663,  5855,  4965, 21566]))
```

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, weryfikujące 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.

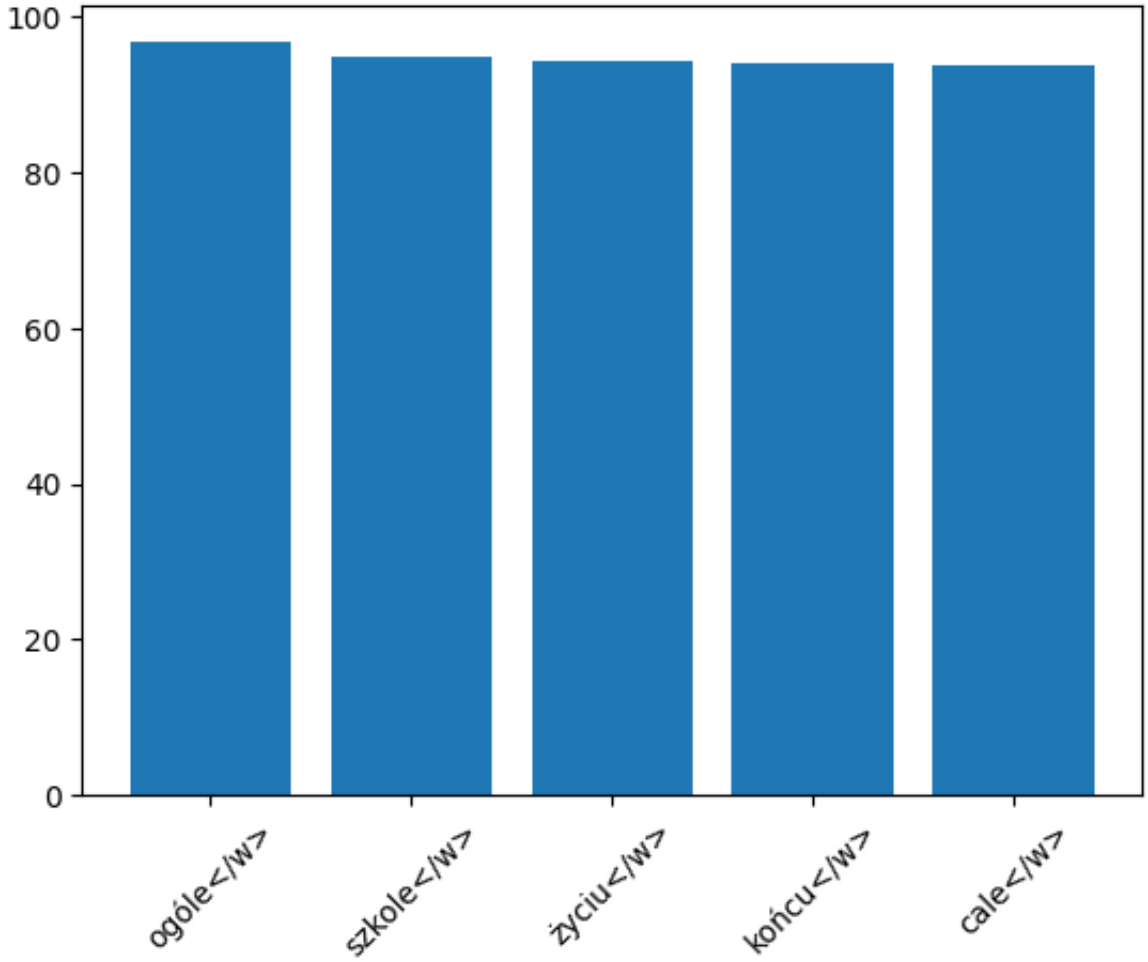
```
def plot_words(sentence, word_model, word_tokenizer, mask="<mask>"):
    sentence = word_tokenizer.encode(sentence, return_tensors="pt")
    tokens = word_tokenizer.convert_ids_to_tokens(list(sentence[0]))
    print("|".join(tokens))
    target = word_model(sentence)
    top = torch.topk(target.logits[0][tokens.index(mask)],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

```
# your_code
tokenizer = AutoTokenizer.from_pretrained("allegro/herbert-base-cased")
model = AutoModelForMaskedLM.from_pretrained("allegro/herbert-base-cased")
```

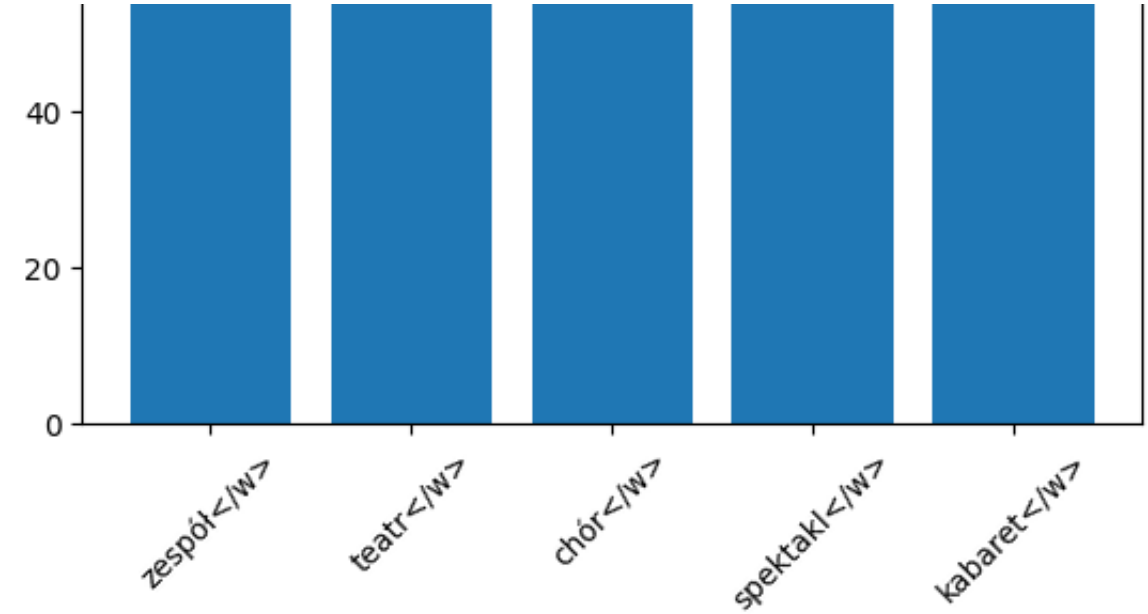
pytorch_model.bin: 100% 654M/654M [00:06<00:00, 148MB/s]

```
plot_words("Kto studiuje na AGH ten w <mask> się nie śmieje", model, tokenizer)
plot_words("Objazdowy <mask> odwiedził nasze miasto. Na występie pojawiły się s
plot_words("Ostatnie show udało się całemu <mask>. Ludzie świetnie się bawili r

plot_words("Nauczyciel przyrody zagroził Antkowi zagrożeniem, dlatego Antek w c
plot_words("Ulubionym deserem Jasia jest kremówka, dlatego na wycieczcę szkolne
plot_words("Filip od dziecka lubił komputery, dlatego poszedł na studia <mask>.

plot_words("Lech Wałęsa był prezydentem <mask>.", model, tokenizer)
plot_words("Togo jest krajem położonym w <mask>.", model, tokenizer)
plot_words("Kangury występują jedynie w <mask>.", model, tokenizer)
```

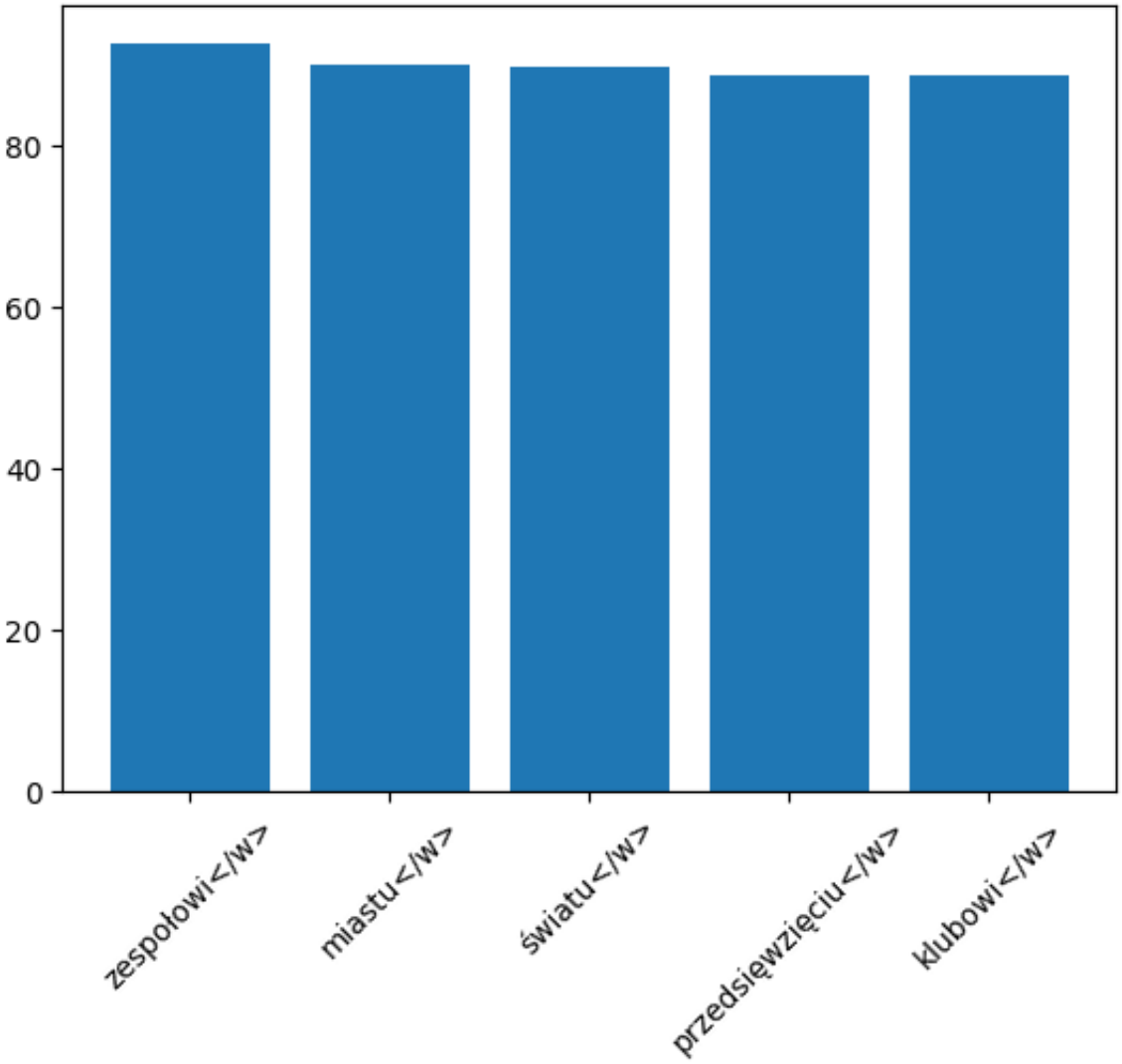
<s>|Kto</w>|studiuje</w>|na</w>|AGH</w>|ten</w>|w</w>|<mask>|się</w>|nie</w



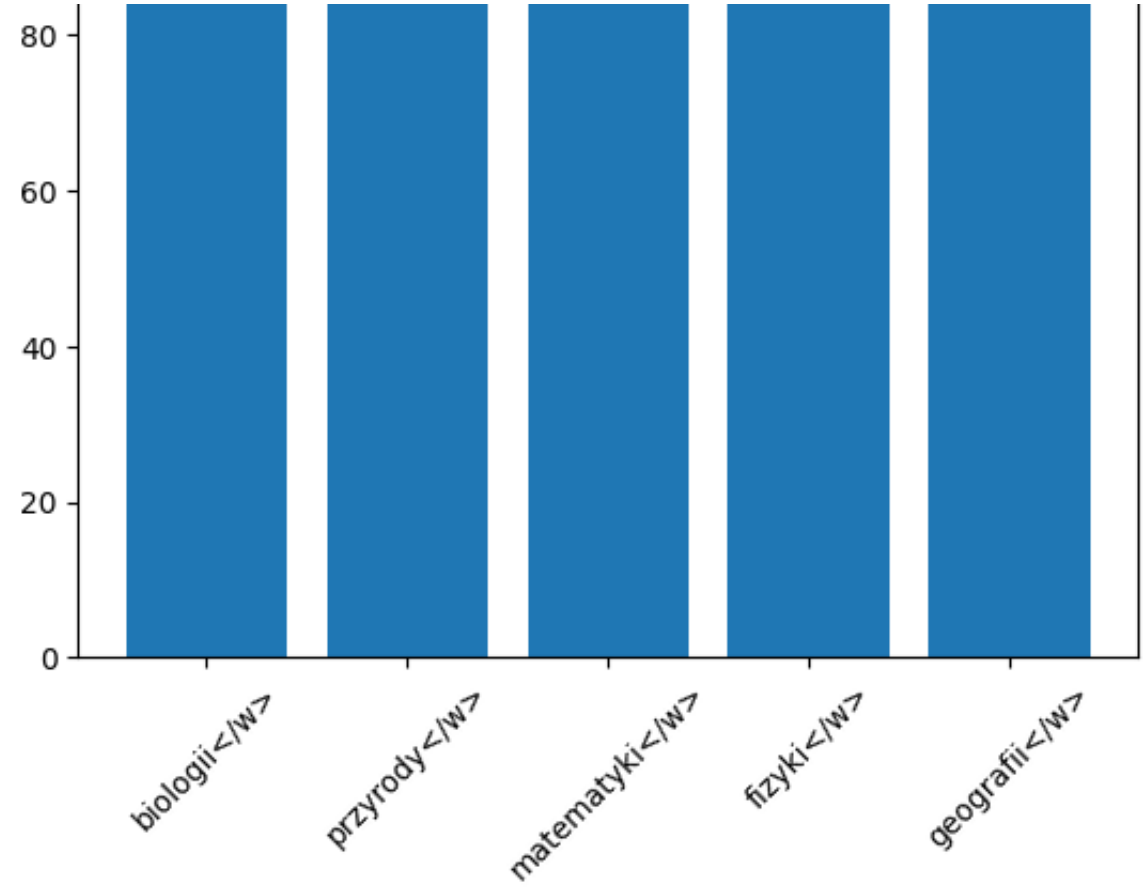
<s>|Ob|jaz|dowy</w>|<mask>|odwiedził</w>|nasze</w>|miasto</w>|.</w>|Na</w>|



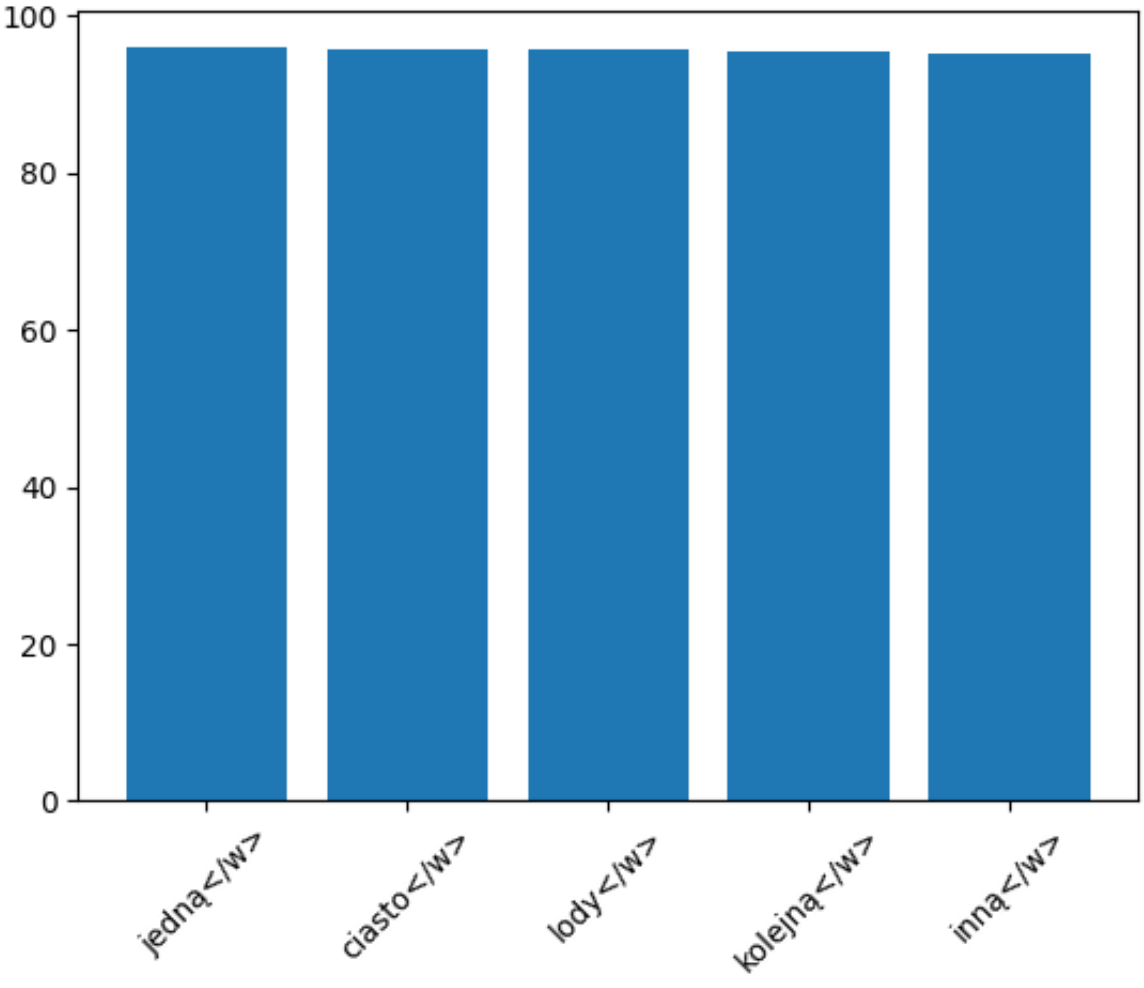
<s>|Ostatnie</w>|show</w>|udało</w>|się</w>|całemu</w>|<mask>|.</w>|Ludzie<



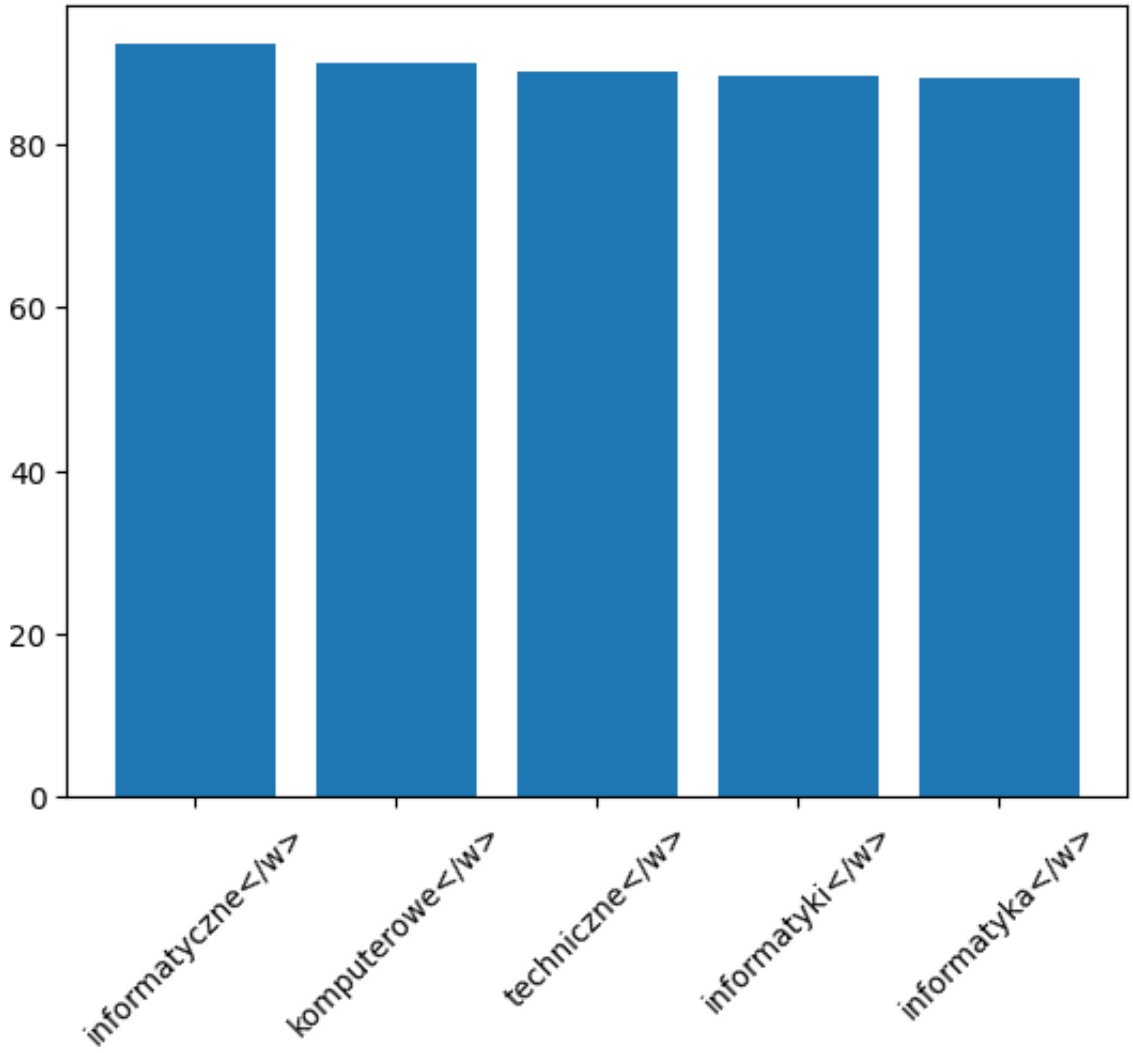
<s>|Nauczyciel</w>|przyrody</w>|zagroz|ił</w>|An|tkowi</w>|zagrożeniem</w>|



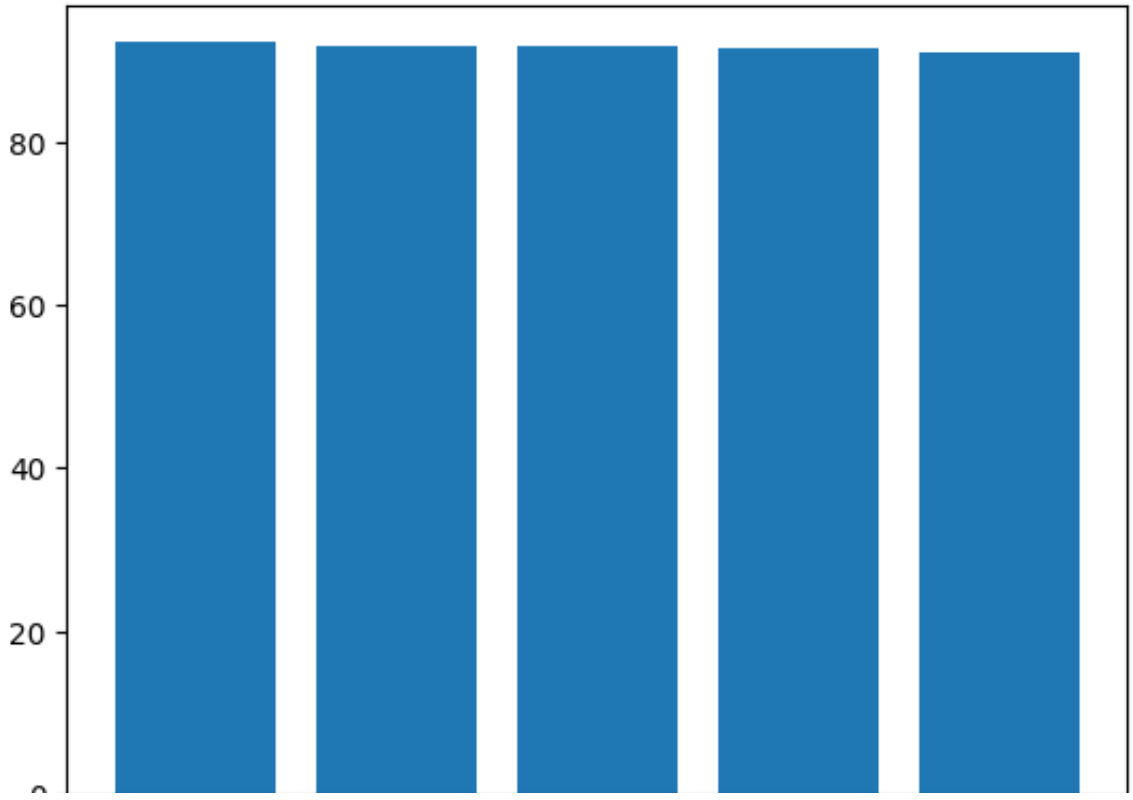
<s>|Ulubi|onym</w>|de|serem</w>|Ja|sia</w>|jest</w>|kre|mów|ka</w>|,</w>|dl

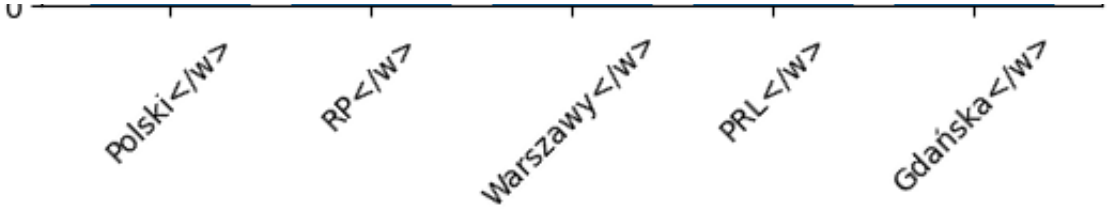


<s>|Filip</w>|od</w>|dziecka</w>|lubił</w>|komputery</w>|,</w>|dlatego</w>|

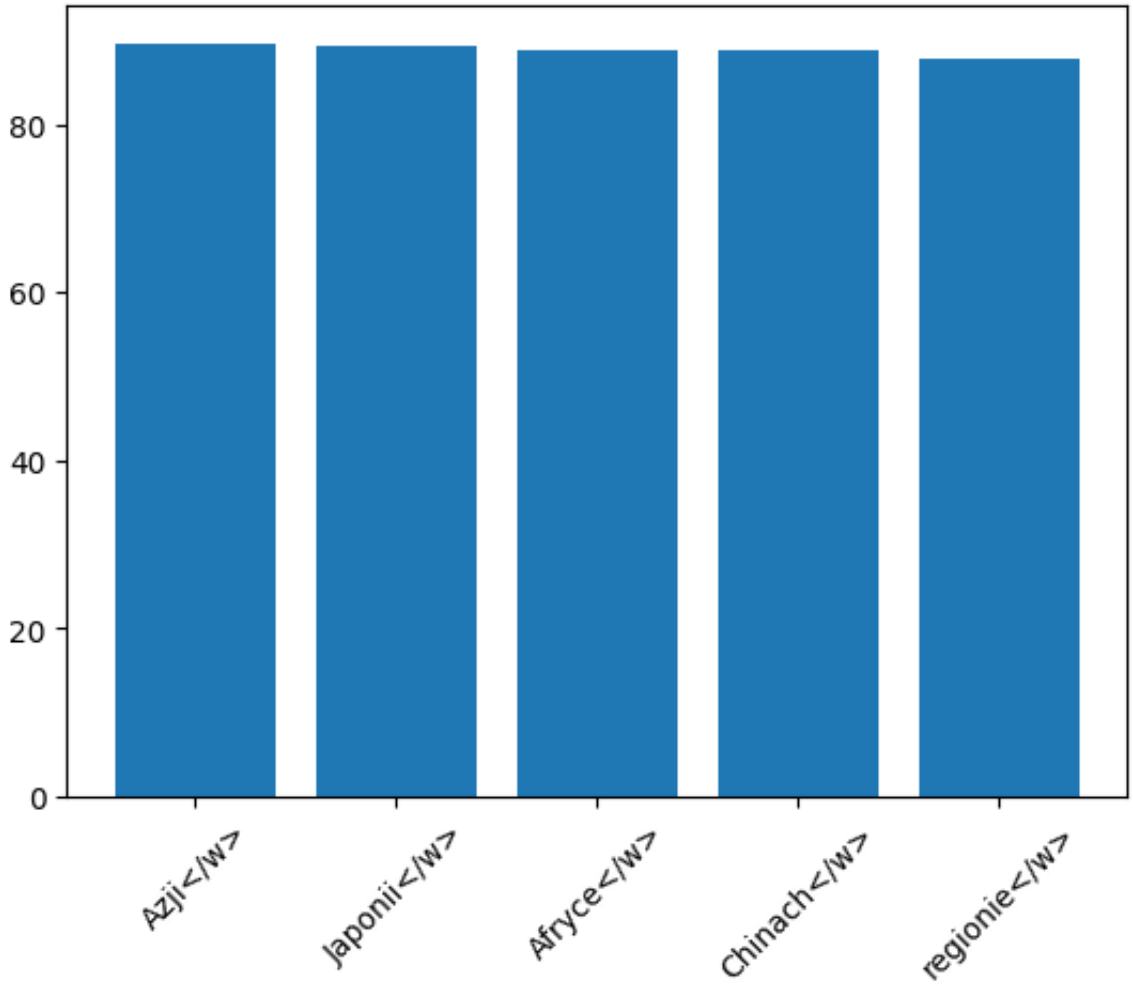


<s>|Lech</w>|Wałęsa</w>|był</w>|prezydentem</w>|<mask>|.</w>|</s>

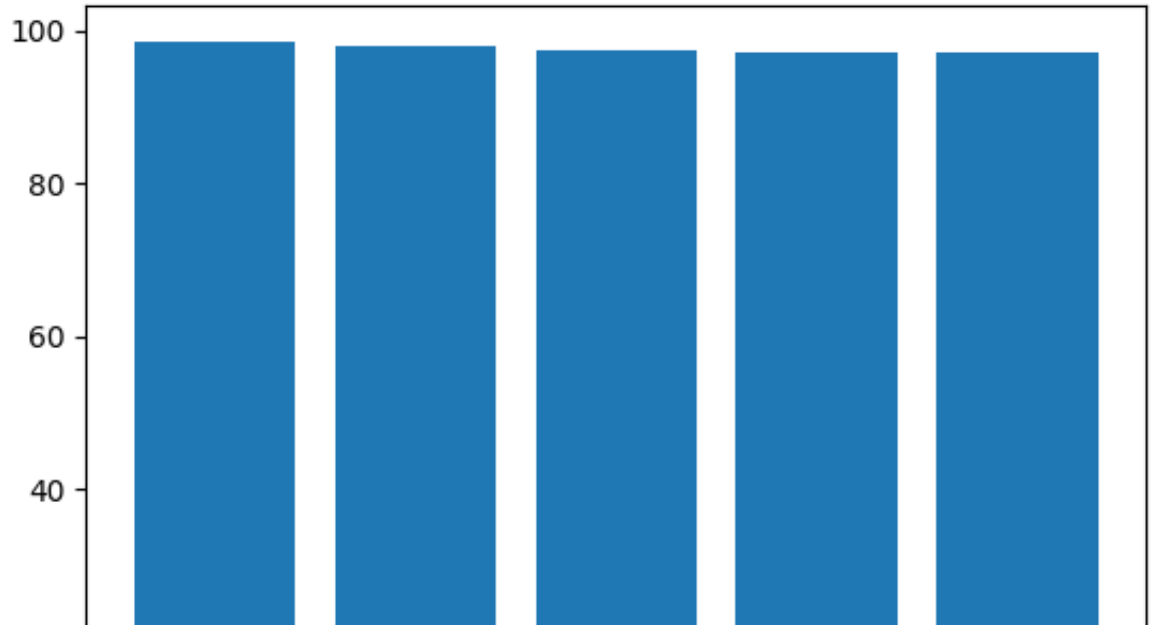


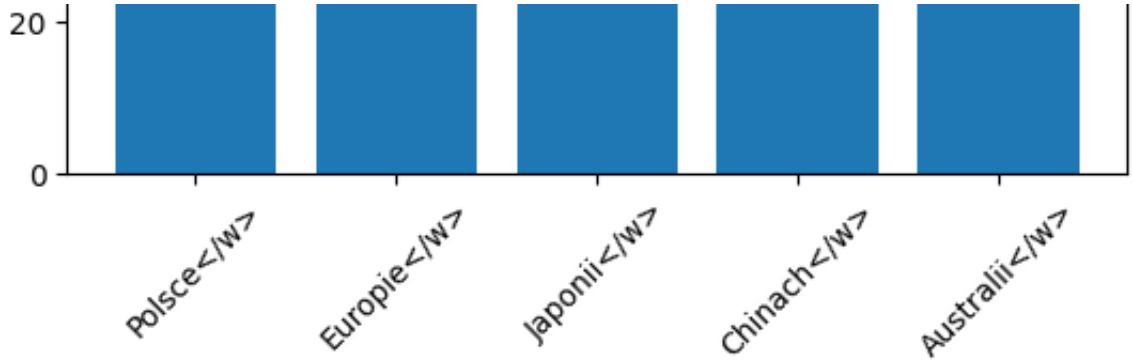


<s>|To|go</w>|jest</w>|krajem</w>|położonym</w>|w</w>|<mask>|.</w>|</s>



<s>|Kan|gury</w>|występują</w>|jedynie</w>|w</w>|<mask>|.</w>|</s>





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 nauczenia 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")
```

```
Downloading builder script: 100% 5.35k/5.35k [00:00<00:00, 135kB/s]
Downloading readme: 100% 317/317 [00:00<00:00, 11.7kB/s]
Downloading data files: 100% 2/2 [00:04<00:00, 2.12s/it]
Downloading data: 100% 47.2M/47.2M [00:01<00:00, 34.9MB/s]
Downloading data: 6.29M/6.29M [00:00<00:00,
```

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 udstoępniaja 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 początkotwych 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.jsor
!wget https://huggingface.co/datasets/clarin-pl/poquad/resolve/main/poquad-trai
```

```
--2024-01-22 13:44:20--  https://huggingface.co/datasets/clarin-pl/poquad/r
Resolving huggingface.co (huggingface.co)... 18.164.174.23, 18.164.174.55,
Connecting to huggingface.co (huggingface.co)|18.164.174.23|:443... connect
HTTP request sent, awaiting response... 200 OK
Length: 6286317 (6.0M) [text/plain]
Saving to: ‘poquad-dev.json’

poquad-dev.json     100%[===================>]   5.99M  13.1MB/s    in 0.5s

2024-01-22 13:44:20 (13.1 MB/s) - ‘poquad-dev.json’ saved [6286317/6286317]

--2024-01-22 13:44:20--  https://huggingface.co/datasets/clarin-pl/poquad/r
Resolving huggingface.co (huggingface.co)... 18.164.174.23, 18.164.174.55,
Connecting to huggingface.co (huggingface.co)|18.164.174.23|:443... connect
HTTP request sent, awaiting response... 302 Found
Location: https://cdn-lfs.huggingface.co/repos/18/de/18ded45e8046dd5f58b736
--2024-01-22 13:44:21--  https://cdn-lfs.huggingface.co/repos/18/de/18ded45
Resolving cdn-lfs.huggingface.co (cdn-lfs.huggingface.co)... 18.65.25.83, 1
Connecting to cdn-lfs.huggingface.co (cdn-lfs.huggingface.co)|18.65.25.83|:
HTTP request sent, awaiting response... 200 OK
Length: 47183344 (45M) [application/json]
Saving to: ‘poquad-train.json’

poquad-train.json   100%[===================>]  45.00M   161MB/s    in 0.3s

2024-01-22 13:44:21 (161 MB/s) - ‘poquad-train.json’ saved [47183344/471833
```

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
```


```
{
  "version": "02-20",
  "data": [
    {
      "id": 9773,
      "title": "Miszna",
      "summary": "Miszna (hebr. משנה miszna „nauczać”, „ustnie przekazywać”
      "url": "https://pl.wikipedia.org/wiki/Miszna",
      "paragraphs": [
        {
          "context": "Pisma rabiniczne – w tym Miszna – stanowią kompilację
          "qas": [
            {
              "question": "Czym są pisma rabiniczne?",
              "answers": [
                {
                  "text": "kompilację poglądów różnych rabinów na określony
                  "answer_start": 43,
                  "answer_end": 97,
                  "generative_answer": "kompilacją poglądów różnych rabinów
                }
              ],
              "is_impossible": false
            },
            {
              "question": "Z ilu komponentów składała się Tora przekazana M
              "answers": [
                {
                  "text": "dwóch",
                  "answer_start": 172,
```

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` , wskazujace 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` .

Zaczniemy 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_da
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 oczekiwanie biblioteki Transformer.

Pytania, które mają ustawiną 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 wylkuczyć 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: {negati
                                "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])}/{s
)
```

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

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

Dobrą 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 monach
[{'text': 'Pytanie: Czym są pisma rabiniczne? Kontekst: Pisma rabiniczne –
```

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")
```

```
Saving the dataset (1/1 shards): 100% 75605/75605 [00:00<00:00, 304772.65 examples/s]
Saving the dataset (1/1 shards): 100% 12979/12979 [00:00<00:00, 186491.55 examples/s]
```

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"]
```

```
Map:                                                  75605/75605 [01:11<00:00, 1183.31
100%                                                  examples/s]
Map:                                                  12372/12372 [00:11<00:00, 888.22
100%                                                  examples/s]
Dataset({
```

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

```
Pytanie: Co było powodem powrócenia konceptu porozumieniu monachijskiego? K
[0, 14142, 1335, 3407, 2404, 14736, 6491, 4081, 6743, 2213, 19824, 25437, 3
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

Możem też sprawdzić, jak został stokenizowany pierwszy przykład:

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

```
<s>|Pytanie</w>|:</w>|Co</w>|było</w>|powodem</w>|powró|cenia</w>|koncep|tu
```

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 atencji, 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 Transformes pozwala na załadowanie tego samego modelu dostosowanego do różnych zadań. Wcześniej używaliśmy modelu HerBERT do predykcji brakującego wyrazu. Teraz załadujemy 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
You should probably TRAIN this model on a down-stream task to be able to us
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 dostosowywane 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 - jeślie 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 zawieraja 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 prawodopodobnej 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)
```

```
Downloading builder script:  4.20k/4.20k [00:00<00:00,
100%                         241kB/s]
```

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 jezyka 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 (tagowaina). 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.

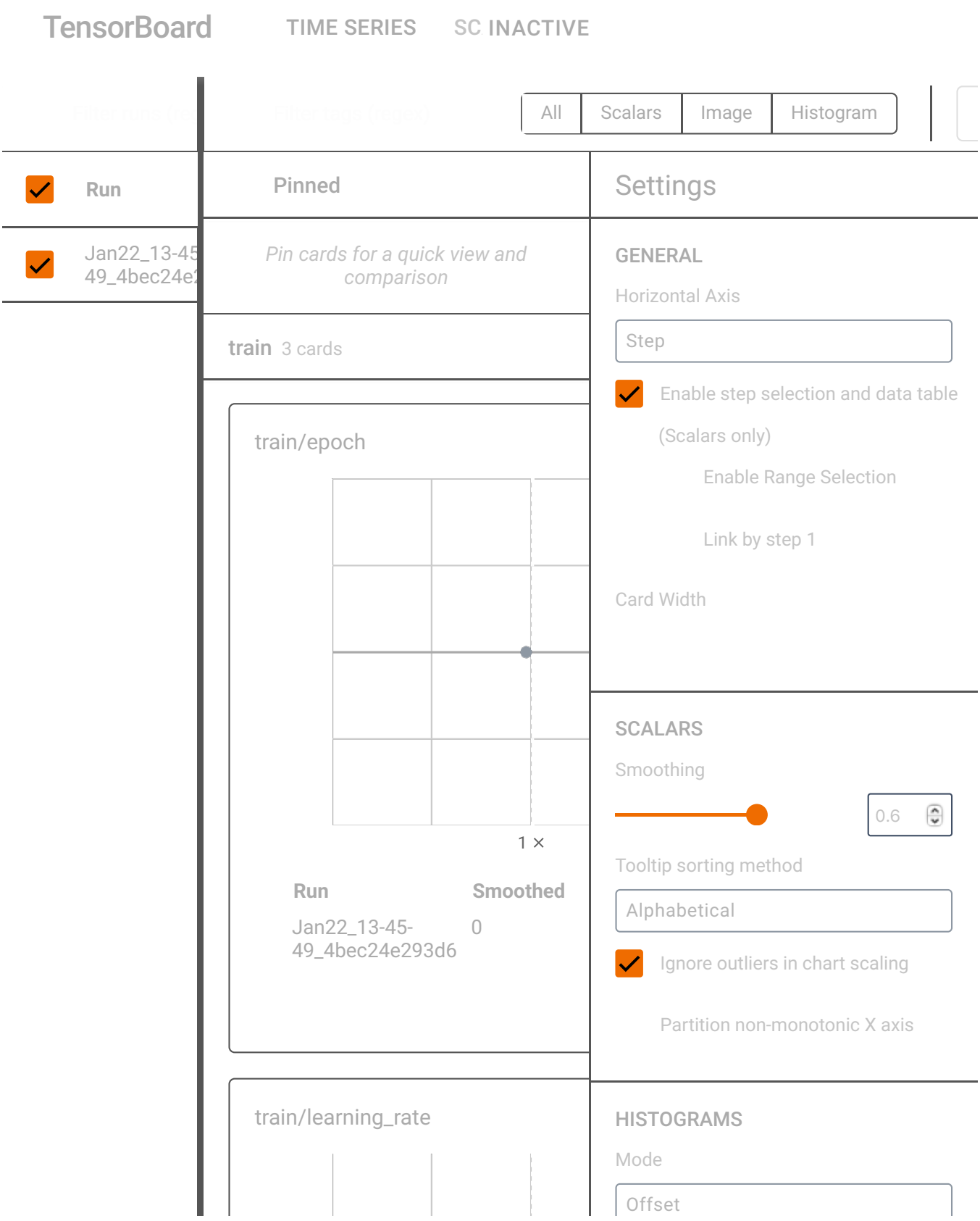
Wykorzystamy wcześniej utworzone zbiory danych i dodatkowo zmienimy kolejność danych, tak aby uniknąć potencjalnego problemu z korelacją danych w ramach batcha. Wykorzystujemy do tego wywołanie `shuffle`.

```
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 niezbędy. 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 osiągane wartości interesujących nas parametrów.

Warto zauważyć, że istenieje 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
```



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 radzie sobie z postawionym przed nim problemem na danych ewaluacyjnych.

```
trainer.train()
```

[2283/4726 40:15 < 43:06, 0.94 it/s, Epoch 0.48/1]

Step	Training Loss	Validation Loss	Accuracy
300	0.346900	0.319772	0.884901
600	0.308800	0.274794	0.890398
900	0.275100	0.281818	0.892742
1200	0.267500	0.330841	0.885790
1500	0.289900	0.270413	0.898076
1800	0.242200	0.322670	0.891933
2100	0.305500	0.246748	0.899127

[4726/4726 1:24:55, Epoch 1/1]

Step	Training Loss	Validation Loss	Accuracy
300	0.346900	0.319772	0.884901
600	0.308800	0.274794	0.890398
900	0.275100	0.281818	0.892742
1200	0.267500	0.330841	0.885790
1500	0.289900	0.270413	0.898076
1800	0.242200	0.322670	0.891933
2100	0.305500	0.246748	0.899127
2400	0.314900	0.257198	0.898723
2700	0.272500	0.265869	0.893550
3000	0.258500	0.252463	0.899774
3300	0.242600	0.243881	0.903896
3600	0.258500	0.246244	0.903977
3900	0.270700	0.248252	0.904785
4200	0.229700	0.242992	0.905997
4500	0.242900	0.250312	0.905351

```
TrainOutput(global_step=4726, training_loss=0.2750239751624496, metrics=
{'train_runtime': 5096.9551, 'train_samples_per_second': 14.833,
'train_steps_per_second': 0.927, 'total_flos': 1.9892513404928e+16
```

Zadanie 3 (1 punkt)

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

```
context = """Kontekst: Wojna wietnamska (zwana też drugą wojną indochińską, woj
 – działania militarne na Półwyspie Indochińskim w latach 1955–1975.
W konflikt zaangażowane były z jednej strony komunistyczna Demokratyczna Republ
gł. Związek Radziecki oraz Chiny, ale też w mniejszym stopniu przez niektóre pa
i wspierane przez to państwo organizacje komunistyczne w Wietnamie Południowym,
a z drugiej strony Republika Wietnamu wraz z międzynarodową koalicją obejmująca
Tajlandię, Australię, Nową Zelandię i Filipiny. Wbrew swojej woli stronami konf
Walki toczyły się na terytoriach Wietnamu Południowego, Laosu i Kambodży. Amery

questions = [
    "Czy w wojnę wietnamską zamieszane była Mongolia?",
    "Czy walki toczyły się w Laosie?",
    "Jakie zdanie na temat wojny miała ówczesna polska władza?",
    "Czy Stany Zjednoczone były po tej samej stronie co Chiny?",
    "Kto dokonywał nalotów bombowych?"
    ]

for q in questions:
  input = f"Pytanie: {q} {context}"
  input = pl_tokenizer.encode(input, return_tensors="pt", padding="max_length",
  output = model(input, attention_mask=(input != 1).int())
  print(f"{q} | {'Można' if output.logits.argmax() else 'Nie można'}")
```

```
Czy w wojnę wietnamską zamieszane była Mongolia? | Można
Czy walki toczyły się w Laosie? | Można
Jakie zdanie na temat wojny miała ówczesna polska władza? | Nie można
Czy Stany Zjednoczone były po tej samej stronie co Chiny? | Można
Kto dokonywał nalotów bombowych? | Można
```

Model poprawnie ocenia możliwość odpowiedzenia na zadane pytanie

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. Wybierzem 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

# your_code
sets = [[], []]

for id, 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"]:
                for answer in question_answers["answers"]:
                  sets[id].append(
                      {
                          "text": f"Pytanie: {question} Kontekst: {context}",
                          "labels": answer["generative_answer"],
                      }
                  )
            if "plausible_answers" in question_answers:
                for answer in question_answers["plausible_answers"]:
                  sets[id].append(
                      {
                          "text": f"Pytanie: {question} Kontekst: {context}",
                          "labels": answer["generative_answer"],
                      }
                  )

train_set, dev_set = sets
print(f"Total count in train set: {len(train_set)}")
print(f"Total count in dev set: {len(dev_set)}")

train_dataset = Dataset.from_list(train_tuples)
dev_dataset = Dataset.from_list(dev_tuples)
datasets = DatasetDict({"train": train_dataset, "dev": dev_dataset})
```

```
Total count in train set: 41577
Total count in dev set: 6809
```

Zanim przejdziemy do dalszej części, sprawdźmy, czy dane zostały poprawnie utworzone. Zweryfikujmy przede wszystkim, czy klucze `text` oraz `label` zawieraja 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 porozumieniu monachijskiego? K
wymiana listów Ripka — Stroński
Pytanie: Czym są pisma rabiniczne? Kontekst: Pisma rabiniczne — w tym Miszn
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`.

```
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)
```

```
tokenizer_config.json: 100%  141/141 [00:00<00:00, 9.25kB/s]
config.json: 100%  658/658 [00:00<00:00, 54.0kB/s]
spiece.model: 100%  1.12M/1.12M [00:00<00:00, 3.50MB/s]
special_tokens_map.json: 100%  65.0/65.0 [00:00<00:00, 5.03kB/s]
You are using the default legacy behaviour of the <class 'transformers.mode
```

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

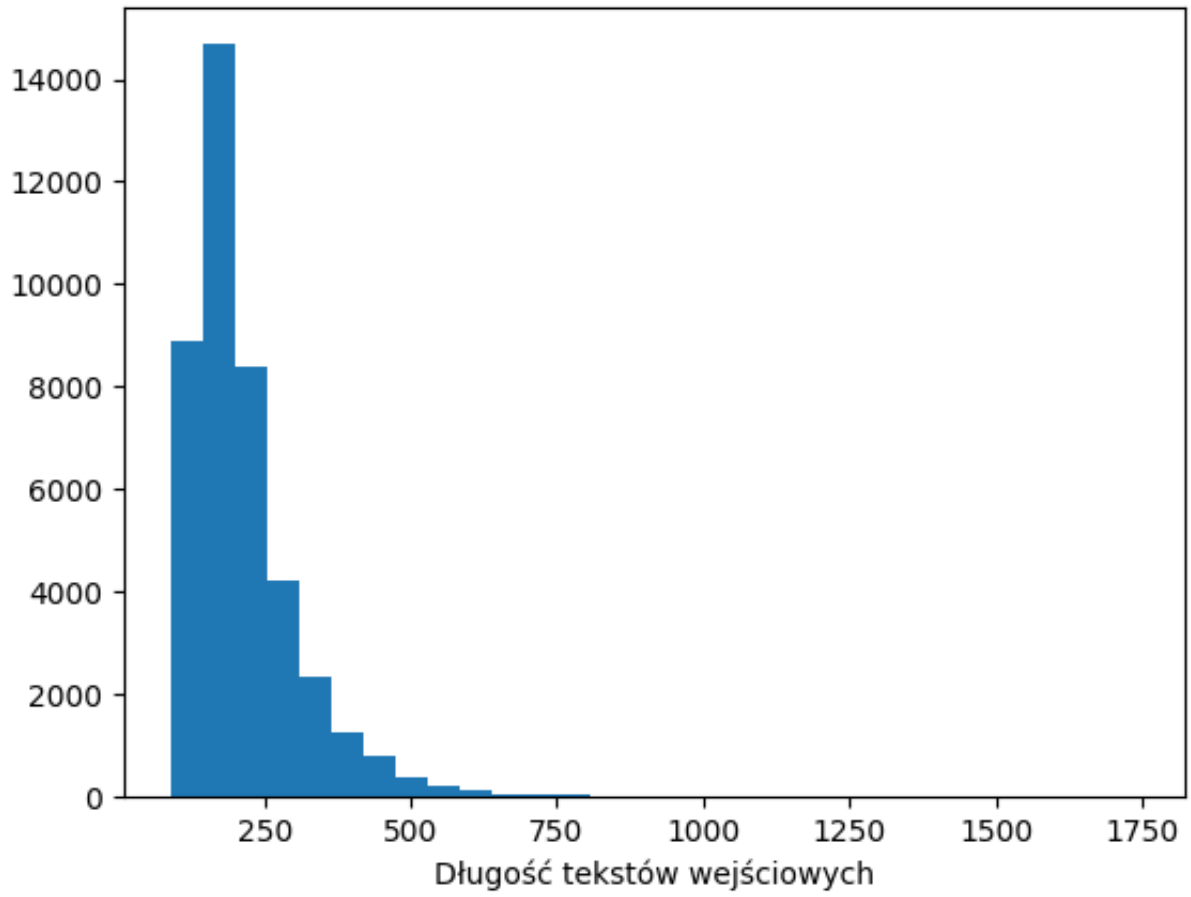
```
dict_keys(['text', 'labels', 'input_ids', 'attention_mask'])
[21584, 291, 639, 402, 11586, 292, 23822, 267, 1269, 8741, 280, 24310, 4240
[13862, 20622, 2178, 18204, 308, 8439, 2451, 1]
165
8
_Pytanie|:|_Co|_było|_powodem|_po|wróceni|a|_kon|cept|u|_porozumieniu|_mona
_wymiana|_listów|_Ri|pka|_–|_Stro|ński|</s>
```

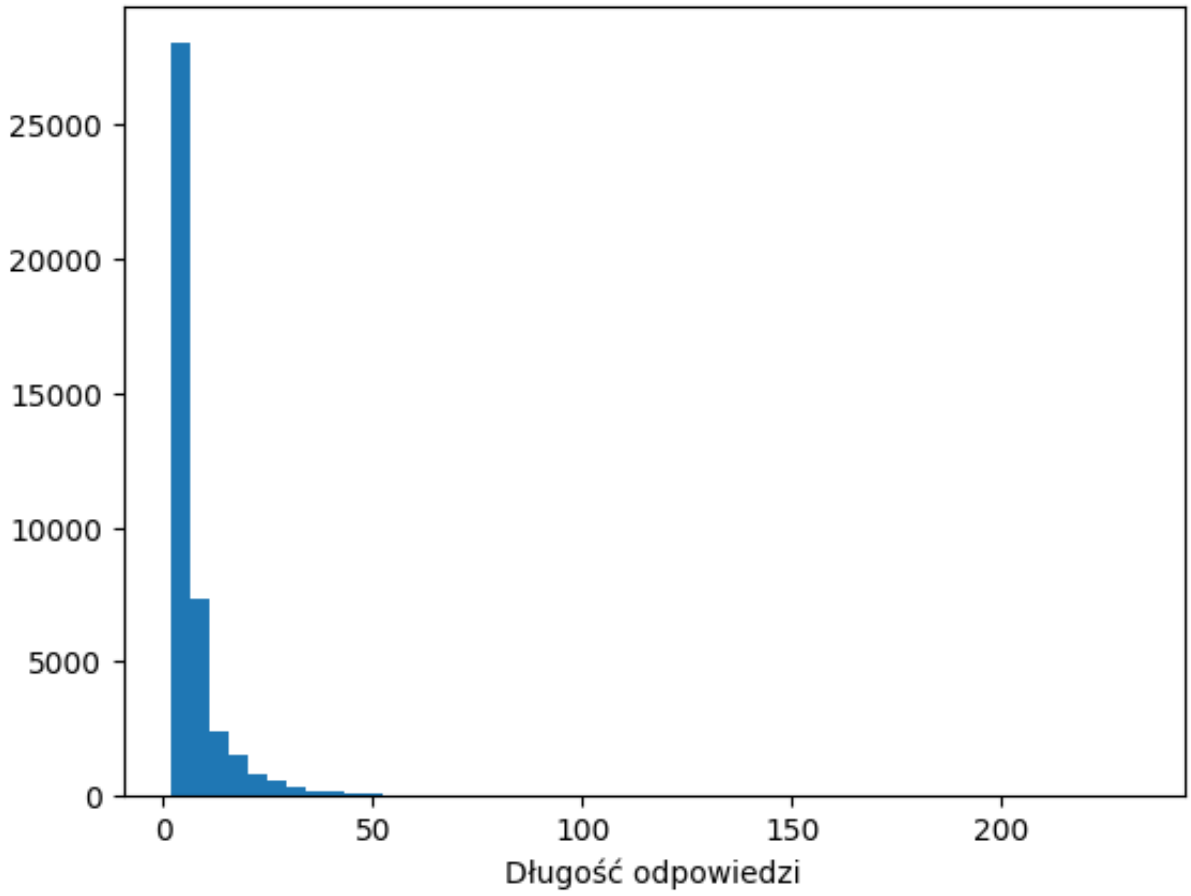
Wykorzystywany przez nas model obsługuje teksty od długości do 512 sub-tokenów (w zasadzie ograniczenie to, w przeciwieństwie do modelu BERT nie wynika z samego modelu, więc teoretycznie moglibyśmy wykorzystywać dłuższe sekwencje, co jednak prowadzi do nadmiernej konsumpcji pamięci). Konieczne jest zatem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

# your_code
input_lengths = np.array([len(i) for i in tokenized_datasets['train']['input_ic
label_lengths = np.array([len(i) for i in tokenized_datasets['train']['labels']

plt.hist(input_lengths, bins=30)
plt.xlabel("Długość tekstów wejściowych")
plt.show()
plt.hist(label_lengths, bins=50)
plt.xlabel("Długość odpowiedzi")
plt.show()
```


14000
12000
10000
8000
6000
4000
2000
0
250
500
750
1000
1250
1500
1750
Długość tekstów wejściowych


25000
20000
15000
10000
5000
0
0
50
100
150
200
Długość odpowiedzi

Double-click (or enter) to edit

Przyjmiemy 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 sekewncje 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)
```

```
Map:                                              41577/41577 [00:39<00:00, 1052.49
100%                                              examples/s]
Map:                                              6800/6800 [00:05<00:00, 1263.00
```

Następnie weryfkiujemy, 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, 4240
[13862, 20622, 2178, 18204, 308, 8439, 2451, 1, -100, -100, -100, -100, -10
165
32
```

Dla problemu odpowiadania na pytania potrzebować będziemy innego pre-trenowanego modelu oraz innego przygotowania danych. Jako model bazowy wykrzystamy 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 pretrenowany był na kilku konkretnych zadaniach z zakresu NLP (w tym odpowiadaniu na pytania). W wariancie 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")
```

```
pytorch_model.bin:                                  1.10G/1.10G [00:19<00:00,
100%                                                57.4MB/s]
```

Trening modelu QA

Ostatnim krokiem przed uruchomieniem treningu jest zdefiniowanie metryk, wskazujacych 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 bibilotekę `evaluate`, która zawiera definicje obu metryk.

Przy konwersji identyfikatorów tokenów na tekstu zamieniamy również z powroten 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.pac
    decoded_preds = plt5_tokenizer.batch_decode(predictions, skip_special_toker
    labels = np.where(labels != -100, labels, plt5_tokenizer.pad_token_id)
    decoded_labels = plt5_tokenizer.batch_decode(labels, skip_special_tokens=Tr
    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=de
    del result["precisions"]

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

    return result
```

```
Downloading builder script: 100% 5.67k/5.67k [00:00<00:00, 163kB/s]
Downloading builder script: 100% 5.94k/5.94k [00:00<00:00, 120kB/s]
Downloading extra 4.07k/? [00:00<00:00
```

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 przeprowadzoany jest proces uczenia i ewaluacji.

```
arguments = Seq2SeqTrainingArguments(
    output_dir = path + "/output_qa",
    num_train_epochs = 3,
    per_device_train_batch_size = 16,
    per_device_eval_batch_size = 16,
    eval_steps = 100,
    learning_rate = 1e-4,
    optim = 'adafactor',
    eval_accumulation_steps = 4,
    do_train = True,
    do_eval = True,
    evaluation_strategy ="steps",
    generation_max_length=32,
    predict_with_generate=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,
- wykorzystawać zbiór `dev` do evaluacji,
- wykorzystać klasę batchującą (`data_collator`) o nazwie `DataCollatorWithPadding`.

```
from transformers import DataCollatorWithPadding

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


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

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

No dashboards are active for the current data set.

Probable causes:

- You haven't written any data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the README and perhaps the TensorBoard tutorial.

If you think TensorBoard is configured properly, please see the section of the README devoted to missing data problems and consider filing an issue on GitHub.

Last reload: Jan 22, 2024, 5:07:42 PM

Log directory: gdrive/MyDrive/poquad/output_qa/runs

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 Hugginface.

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

```
model_apohllo = AutoModelForSeq2SeqLM.from_pretrained("apohllo/plt5-base-poquac
tokenizer_apohllo = AutoTokenizer.from_pretrained("apohllo/plt5-base-poquad")
```

```
config.json: 100%                                  826/826 [00:00<00:00, 32.0kB/s]
model.safetensors:                                 1.10G/1.10G [00:28<00:00,
100%                                               29.5MB/s]
generation_config.json:                            112/112 [00:00<00:00,
100%                                               7.09kB/s]
tokenizer_config.json:                             833/833 [00:00<00:00,
100%                                               54.4kB/s]
spiece.model:                                      1.12M/1.12M [00:00<00:00,
```

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 tygo 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.

```
# your_code
context = """
Po wykryciu przez wywiad USA na początku sierpnia 1962 dziwnych budowli na Kubie
 a następnie po wykryciu urządzeń i rakiet, w odpowiedzi na zagrożenie prezydent
  (blokadę transportu środków bojowych) i zażądał wycofania rakiet. Potencjalny
  gdy do Kuby zaczęły się zbliżać radzieckie statki wiozące kolejne materiały mi
```

```
Konflikt został zażegnany, kiedy po żądaniu prezydenta USA przywódca radziecki N
zawrócić statki oraz wyraził zgodę na demontaż wyrzutni rakietowych, w zamian za
a także wycofania rakiet amerykańskich z Turcji. Takie rozwiązanie wywołało nega
do przejściowego ochłodzenia stosunków sowiecko-kubańskich.

Stany Zjednoczone w owym czasie nie podały do publicznej wiadomości faktu wycofa
co pozwoliło Chruszczowowi zachować honor na arenie międzynarodowej, jednak w mi

Konsekwencją zagrożenia katastrofą atomową było uruchomienie gorącej linii telef
aby w przyszłości móc się w łatwiejszy sposób uporać z potencjalnymi konfliktami

Propaganda radziecka przedstawiała pomyślne zakończenie kryzysu jako wielki triu
że nie można na nim polegać, jako na przywódcy państwa i partii. Uważali, że Chr
Porażka z Kennedym miała znaczenie dla przedwczesnego zakończenia kariery Chrusz
w wyniku spisku z udziałem Leonida Breżniewa. Breżniew został nowym przywódcą ZS
polityki – bezpośredniego starcia z USA i otwartego, bezceremonialnego ingerowan
"""

questions = [
    "Kto wykrył dziwne budowlne na Kubie?",
    "Kto był prezydentem USA w w czasie kryzysu kubańskiego?",
    "Kto był przywódcą ZSRR w trakcie konfliktu?",
    "W którym roku miał miejsce kryzys?",
    "Kto rządził Kubą?",
    "Kiedy kryzys został zażegnany?",
    "Kiedy został obalony Nikita Chruszczow?",
    "Kto został następcą Chruszczowa?",
    "Skąd USA wycofało swoje rakiety?",
    "Jakie zagrożenie było bardzo możliwe podczas kryzysu kubańskiego?"
]

inputs = [f"Pytanie: {q} Kontekst: {context}" for q in questions]
inputs = tokenizer_apohllo(inputs, return_tensors="pt", padding=True)
outputs = model_apohllo.generate(
    input_ids=inputs["input_ids"],
    attention_mask=inputs["attention_mask"]
)
outputs = tokenizer_apohllo.batch_decode(outputs, skip_special_tokens=True)

for i in range(len(questions)):
  print(f"Pytanie: {questions[i]}")
  print(f"Odpowiedź: {outputs[i]}")
```

```
/usr/local/lib/python3.10/dist-packages/transformers/generation/utils.py:12
  warnings.warn(
Pytanie: Kto wykrył dziwne budowlne na Kubie?
Odpowiedź: wywiad USA
Pytanie: Kto był prezydentem USA w w czasie kryzysu kubańskiego?
Odpowiedź: John F. Kennedy
Pytanie: Kto był przywódcą ZSRR w trakcie konfliktu?
Odpowiedź: Breżniew
Pytanie: W którym roku miał miejsce kryzys?
Odpowiedź: 1962
Pytanie: Kto rządził Kubą?
Odpowiedź: John F. Kennedy
Pytanie: Kiedy kryzys został zażegnany?
Odpowiedź: kiedy po żądaniu prezydenta USA przywódca radziecki Nikita Chrus
Pytanie: Kiedy został obalony Nikita Chruszczow?
Odpowiedź: 28 października
Pytanie: Kto został następcą Chruszczowa?
Odpowiedź: Breżniew
Pytanie: Skąd USA wycofało swoje rakiety?
Odpowiedź: z Turcji
Pytanie: Jakie zagrożenie było bardzo możliwe podczas kryzysu kubańskiego?
Odpowiedź: katastrofa atomowa
```

Model potrafi poprawnie odpowiedzieć na dość oczywiste pytania, jendak nie jest w stanie wychwycić informacji która nie jest dana wprost. Ma również problem gdy te same nazwiska występują z różnymi datami koło siebie (Kiedy został obalony Nikita Chruszczow) oraz potrafi dawać w odpowiedzi sprzeczne informacje (Pytanie o przywódcę USA i Kuby)

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 przyspieszenie 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.