Przetwarzanie języka naturalnego

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).

Przygotowanie środowiska

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.

W tym środowisku możemy wybrać akcelerator spośród GPU i TPU. Sprawdźmy, czy mamy dostęp do środowiska wyposażonego w akcelerator NVidii:

```
!nvidia-smi
```

```
Tue Jan  3 10:20:11 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   49C    P0    26W /  70W |      0MiB / 15109MiB |      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.

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 sacremoses datasets evaluate sentencepiece
```

```
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.8/dist-packages (from transformers) (1.21.6)
Requirement already satisfied: filelock in /usr/local/lib/python3.8/dist-packages (from transformers) (3.8.2)
Collecting tokenizers!=0.11.3,<0.14,>=0.11.1
  Downloading tokenizers-0.13.2-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.6 MB)
     |████████████████████████████████████████| 7.6 MB 53.1 MB/s
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.8/dist-packages (from transformers) (2022.6.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.8/dist-packages (from transformers) (6.0)
```

```
Collecting multiprocess
  Downloading multiprocess-0.70.14-py38-none-any.whl (132 kB)
     |████████████████████████████████| 132 kB 58.0 MB/s
Requirement already satisfied: dill<0.3.7 in /usr/local/lib/python3.8/dist-packages (from datasets) (0.3.6)
Requirement already satisfied: fsspec[http]>=2021.11.1 in /usr/local/lib/python3.8/dist-packages (from datasets) (2022.11.0)
Requirement already satisfied: pandas in /usr/local/lib/python3.8/dist-packages (from datasets) (1.3.5)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets) (1.8.2)
Requirement already satisfied: charset-normalizer<3.0,>=2.0 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets) (2.1.1)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets) (1.3.1)
Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets) (4.0.2)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets) (1.3.3)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets) (6.0.3)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets) (22.1.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/dist-packages (from requests->transformers) (2022.12.7)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.8/dist-packages (from requests->transformers) (3.0.4)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.8/dist-packages (from requests->transformers) (1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-packages (from requests->transformers) (2.10)
Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1
  Downloading urllib3-1.25.11-py2.py3-none-any.whl (127 kB)
     |████████████████████████████████| 127 kB 67.6 MB/s
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.8/dist-packages (from pandas->datasets) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (from pandas->datasets) (2022.6)
Building wheels for collected packages: sacremoses
  Building wheel for sacremoses (setup.py) ... done
  Created wheel for sacremoses: filename=sacremoses-0.0.53-py3-none-any.whl size=895260 sha256=74d98dded496ccd958264fda4aa716f34922c2e906
  Stored in directory: /root/.cache/pip/wheels/82/ab/9b/c15899bf659ba74f623ac776e861cf2eb8608c1825ddec66a4
Successfully built sacremoses
Installing collected packages: urllib3, xxhash, responses, multiprocess, huggingface-hub, tokenizers, datasets, transformers, sentencepiece, sacremose
  Attempting uninstall: urllib3
    Found existing installation: urllib3 1.24.3
    Uninstalling urllib3-1.24.3:
      Successfully uninstalled urllib3-1.24.3
```

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")
```

```
Downloading: 100%                570/570 [00:00<00:00, 8.90kB/s]

Downloading: 100%                436M/436M [00:11<00:00, 55.7MB/s]

Some weights of the model checkpoint at bert-base-cased were not used when initializing BertForMaskedLM: ['cls.seq_relationship.weight', 'cls.seq_relati
- This IS expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another architecture (e.g. initial
- This IS NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical (initializing a BertFo
```

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")
```

```
Downloading: 100%  29.0/29.0 [00:00<00:00, 1.43kB/s]
Downloading: 100%  213k/213k [00:00<00:00, 2.76MB/s]
Downloading: 100%  436k/436k [00:00<00:00, 4.71MB/s]
```

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 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 jęzka 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 (1 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

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

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)

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

```
Downloading: 100%  229/229 [00:00<00:00, 10.6kB/s]
```

WNIOSEK: wykorzystując inny tokenizer, otrzymujemy inne rezultaty (i dla angielskich i dla polskich zdań).

Aby edytować zawartość komórki, kliknij ją dwukrotnie (lub naciśnij klawisz Enter)

```
Downloading: 100%  129/129 [00:00<00:00, 8.94kB/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). 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.

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])
```

```
<s>|The</w>|qui|ck</w>|brow|n</w>|[</w>|MA|SK</w>|]</w>|ju|mp|s</w>|o|ver</w>|the</w>|la|zy</w>|do|g</w>|.</w>|</s>
tensor([-2.6457, -2.6808, -2.5248,  ..., -1.1240, -3.3259, -3.4256],
       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
```

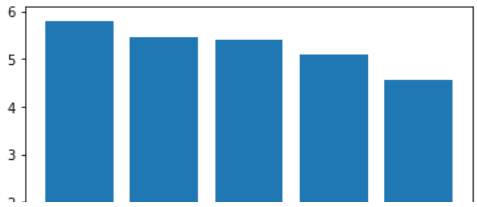
```
torch.return_types.topk(
values=tensor([5.8037, 5.4575, 5.4012, 5.1017, 4.5703], grad_fn=<TopkBackward0>),
indices=tensor([25015,  1116, 20741,  3318,   117]))
```

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>
```



Zgodnie z oczekiwaniami, najbardziej prawdopodobnym uzupełnieniem brakującego wyrazu jest `dog`. 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 nr 2 (2 punkty)

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

- uwzględniania polskich przypadków,
- uwzględniania długodystansowych związków w tekście,
- reprezentowania wiedzy o świecie.

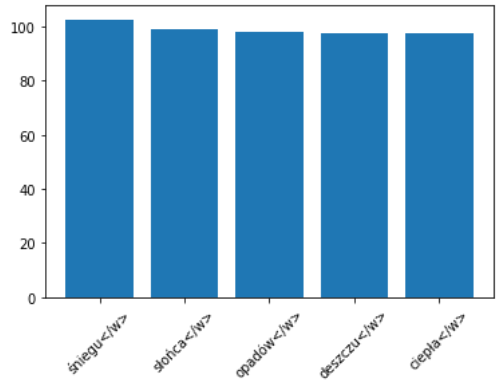
Dla każdego problemu wymyśl po 3 zdania sprawdzające i wyświetl predykcję dla 5 najbardziej prawdopodobnych wyrazów.

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`.

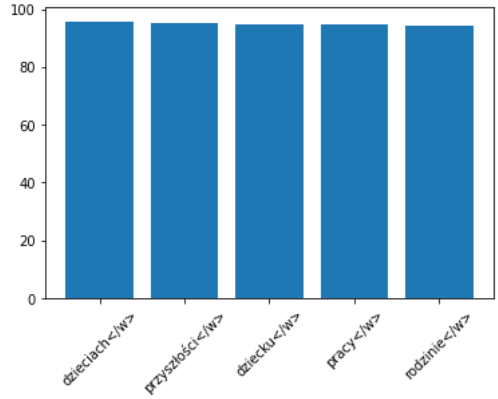
Oceń zdolności modelu w zakresie wskazanych zadań.

```
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")

# uwzględniania polskich przypadków
plot_words("Jest już zima, a nadal brakuje <mask>.", model, tokenizer)
plot_words("Cały czas myślę o <mask>.", model, tokenizer)
plot_words("Stoję na przystanku. Przyglądam się <mask> jazdy.", model, tokenizer)

# uwzględniania długodystansowych związków w tekście
plot_words("Ojciec bardzo kocha swojego <mask>.", model, tokenizer)
plot_words("Jestem głodny. Muszę coś <mask>.", model, tokenizer)
plot_words("Nie zdążyłem na przystanek. Przegapiłem <mask>", model, tokenizer)

# reprezentowania wiedzy o świecie
plot_words("Tom i <mask> to najlepsi przyjaciele.", model, tokenizer)
plot_words("Koty mają <mask> żyć.", model, tokenizer)
plot_words("Wieża Eiffla znajduje się w <mask>.", model, tokenizer)
```

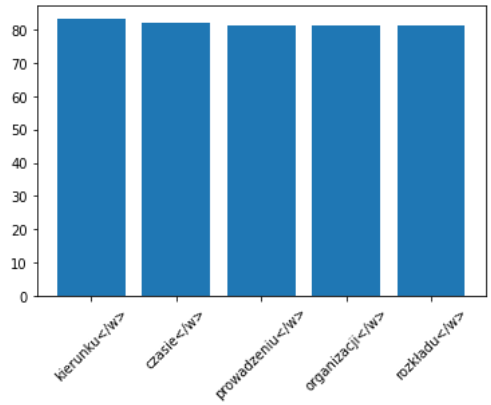
Some weights of the model checkpoint at allegro/herbert-base-cased were not used when initializing BertForMaskedLM: ['cls.sso.sso_relationship.bias', 'c
- This IS expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another architecture (e.g. initial
- This IS NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical (initializing a BertFo
<s>|Jest</w>|już</w>|zima</w>|,</w>|a</w>|nadal</w>|brakuje</w>|<mask>|.</w>|</s>



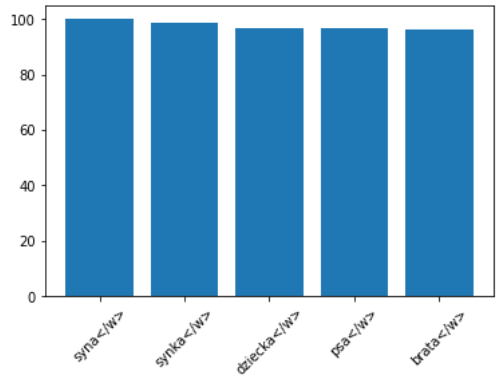
<s>|Cały</w>|czas</w>|myślę</w>|o</w>|<mask>|.</w>|</s>



<s>|Sto|ję</w>|na</w>|przystanku</w>|.</w>|Przy|glą|dam</w>|się</w>|<mask>|jazdy</w>|.</w>|</s>



<s>|Ojciec</w>|bardzo</w>|kocha</w>|swojego</w>|<mask>|.</w>|</s>



<s>|Jestem</w>|głod|ny</w>|.</w>|Muszę</w>|coś</w>|<mask>|.</w>|</s>

Aby edytować zawartość komórki, kliknij ją dwukrotnie (lub naciśnij klawisz Enter)

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.

```
!git clone https://github.com/ipipan/poquad.git
```

```
Cloning into 'poquad'...
remote: Enumerating objects: 33, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 33 (delta 13), reused 25 (delta 7), pack-reused 0
Unpacking objects: 100% (33/33), done.
```

Sprawdźmy jakie pliki znajdują się w katalogu.

```
!ls poquad
```

```
license.txt  poquad_dev.json  poquad_train.json  README.md
```

Możemy sprawadzić, co twórcy napisali na temat samego zbioru (niestety formatowanie tabel jest źle wyświetlane w Jupter notebooku):

```
from IPython.display import display, Markdown, Latex

with open("poquad/README.md") as file:
  display(Markdown(file.read()))
```

PoQuAD

PoQuAD is the Polish Question Answering Dataset. It is modeled on the SQuAD 2.0, including the impossible questions. Additionally it includes a generative answer layer, which allows to train models to return the most natural sounding responses to the queries. The textual data are original Polish texts from Wikipedia, and they were selected to reflect the topics most relevant to Polish speakers.

The dataset consists of

Split	Size
Train	37414
Dev	4667

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.

Zobaczmy jaka jest struktura plików z danymi:

```
!head -30 poquad/poquad_dev.json
```


```
{
  "version": "2022-12-07 19:03:41.585976",
  "data": [
    {
      "id": 9773,
      "title": "Miszna",
      "summary": "Miszna (hebr. משנה miszna „nauczać”, „ustnie przekazywać”, „studiować”, „badać”, od שנה szana „powtarzać”, „różnić się”, „być odmien
      "url": "https://pl.wikipedia.org/wiki/Miszna",
      "paragraphs": [
        {
          "context": "Pisma rabiniczne – w tym Miszna – stanowią kompilację poglądów różnych rabinów na określony temat. Zgodnie z wierzeniami juda
          "qas": [
            {
              "question": "Czym są pisma rabiniczne?",
              "answers": [
                {
                  "text": "kompilację poglądów różnych rabinów na określony temat",
                  "answer_start": 43,
                  "answer_end": 97,
                  "generative_answer": "kompilacją poglądów różnych rabinów na określony temat"
                }
              ],
              "is_impossible": false
            },
            {
              "question": "Z ilu komponentów składała się Tora przekazana Mojżeszowi?",
              "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 skład 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

with open("poquad/poquad_train.json") as input:
  train_data = json.loads(input.read())["data"]

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

with open("poquad/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: 7708
Dev data articles: 964
Train questions: 37417
Dev questions: 4667
```

Ponieważ w pierwszym problemie chcemy stwierdzić, czy na pytanie można udzielić odpowiedzi na podstawie kontekstu, połączymy wszystkie kontekstu 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} Czy kontekst zawiera pytanie?",
                        "label": 0,
                    }
                )
            else:
                tuples[idx].append(
                    {
                        "text": f"Pytanie: {question} Kontekst: {context} Czy kontekst zawiera pytanie?",
                        "label": 1,
                    }
                )
                while True:
                    negative_context = random.choice(all_contexts)
                    if negative_context != context:
                        tuples[idx].append(
                            {
                                "text": f"Pytanie: {question} Kontekst: {negative_context} Czy kontekst zawiera pytanie?",
                                "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: 68174/8520
Positive count in train/dev: 30757/3853
```

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 monachijskiego? Kontekst: Projekty konfederacji zaczęły się załamywać 5 sierpnia
[{'text': 'Pytanie: Czym są pisma rabiniczne? Kontekst: Pisma rabiniczne – w tym Miszna – stanowią kompilację poglądów różnych rabinów na określony te
```

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.

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("question-context-classification")
```

Dane tekstowe przed przekazaniem do modelu wmagają 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 najdłuszego 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.

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

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

```
100%                                    69/69 [00:46<00:00, 2.10ba/s]

100%                                    9/9 [00:06<00:00, 1.48ba/s]

Dataset({
    features: ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 68174
})
```

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? Kontekst: Projekty konfederacji zaczęły się załamywać 5 sierpnia 1942. P
[0, 14142, 1335, 3407, 2404, 14736, 6491, 4081, 6743, 2213, 19824, 25437, 3096, 13875, 1550, 2922, 8413, 1335, 46771, 2152, 17914, 10278, 2022, 11314, 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, 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, 1,
```

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]))
```

```
174
174
```

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
)
```

```
Some weights of the model checkpoint at allegro/herbert-base-cased were not used when initializing BertForSequenceClassification: ['cls.predictions.dec
- This IS expected if you are initializing BertForSequenceClassification from the checkpoint of a model trained on another task or with another architectur
- This IS NOT expected if you are initializing BertForSequenceClassification from the checkpoint of a model that you expect to be exactly identical (initializi
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at allegro/herbert-base-cased and are newly initialized: ['c
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

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.

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 - użycie arytmetyki o zmniejszonej dokładności, przyspieszającej proces uczenia.

```
from transformers import TrainingArguments
import numpy as np

arguments = TrainingArguments(
  output_dir="output",
  do_train=True,
  do_eval=True,
  evaluation_strategy="steps",
  eval_steps=400,
  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: 100%                    4.20k/4.20k [00:00<00:00, 263kB/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,
)
```

```
    Using cuda_amp half precision backend
```

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 output/runs
```

403. That's an error.

That's all we know.

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 400 kroków, będziemy mogli obserwować jak model radzie sobie z postawionym przed nim problemem na danych ewaluacyjnych.

```
trainer.train()
```

```
The following columns in the training set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: text. I
/usr/local/lib/python3.8/dist-packages/transformers/optimization.py:306: FutureWarning: This implementation of AdamW is deprecated and will be remo
  warnings.warn(
***** Running training *****
  Num examples = 68174
  Num Epochs = 1
  Instantaneous batch size per device = 16
  Total train batch size (w. parallel, distributed & accumulation) = 16
  Gradient Accumulation steps = 1
  Total optimization steps = 4261
  Number of trainable parameters = 124444418
```

[4261/4261 58:17, Epoch 1/1]

Step	Training Loss	Validation Loss	Accuracy
400	0.328000	0.274945	0.894131
800	0.345400	0.271966	0.894366
1200	0.292100	0.259841	0.899883
1600	0.259600	0.309275	0.900235
2000	0.246700	0.264882	0.899648
2400	0.237700	0.262183	0.902817
2800	0.245200	0.247284	0.902347
3200	0.284300	0.230100	0.908333
3600	0.231800	0.247846	0.906690
4000	0.236100	0.233614	0.909390

```
The following columns in the evaluation set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: tex
***** Running Evaluation *****
  Num examples = 8520
  Batch size = 16
The following columns in the evaluation set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: tex
***** Running Evaluation *****
  Num examples = 8520
  Batch size = 16
The following columns in the evaluation set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: tex
***** Running Evaluation *****
  Num examples = 8520
  Batch size = 16
The following columns in the evaluation set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: tex
***** Running Evaluation *****
  Num examples = 8520
  Batch size = 16
The following columns in the evaluation set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: tex
***** Running Evaluation *****
  Num examples = 8520
  Batch size = 16
The following columns in the evaluation set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: tex
***** Running Evaluation *****
  Num examples = 8520
  Batch size = 16
The following columns in the evaluation set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: tex
***** Running Evaluation *****
  Num examples = 8520
  Batch size = 16
The following columns in the evaluation set don't have a corresponding argument in `BertForSequenceClassification.forward` and have been ignored: tex
***** Running Evaluation *****
  Num examples = 8520
  Batch size = 16
```

Zadanie 3 (1 punkt)

Zinterpretuj wyniki uzyskane przy treningu modelu klasyfikacyjnego.

Strata treningowa jest odczuwalna. Za to precyzja jest w granicach tolerancji.

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


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"]:
            if not question_answers["is_impossible"]:
                question = question_answers["question"]
                answers = None
                for answer_type in ["answers", "plausible_answers"]:
                    if answer_type in question_answers:
                        answers = question_answers[answer_type][0]["generative_answer"]
                tuples[idx].append(
                    {
                        "text": f"Pytanie: {question} Kontekst: {context} Sformułuj odpowiedź na pytanie.",
                        "labels": answers,
                    }
                )

train_tuples, dev_tuples = tuples
print("Length of train-set:",len(train_tuples))
print("Length of dev-set:", len(dev_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("question-context-qa")
```

```
Length of train-set: 30757
Length of dev-set: 3853
```

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? Kontekst: Projekty konfederacji zaczęły się załamywać 5 sierpnia 1942. P
wymiana listów Ripka – Stroński
Pytanie: Czym są pisma rabiniczne? Kontekst: Pisma rabiniczne – w tym Miszna – stanowią kompilację poglądów różnych rabinów na określony temat. Zgc
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 ~~`google/mt5-small`~~ `allegro/plt5-base`.

```
from transformers import AutoTokenizer

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


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

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

```
loading configuration file config.json from cache at /root/.cache/huggingface/hub/models--allegro--plt5-base/snapshots/56379680948ce8b42d3d48df865
Model config T5Config {
  "_name_or_path": "allegro/plt5-base",
  "architectures": [
    "T5ForConditionalGeneration"
  ],
  "d_ff": 2048,
  "d_kv": 64,
  "d_model": 768,
  "decoder_start_token_id": 0,
  "dense_act_fn": "gelu_new",
  "dropout_rate": 0.1,
  "eos_token_id": 1,
  "feed_forward_proj": "gated-gelu",
  "initializer_factor": 1.0,
  "is_encoder_decoder": true,
  "is_gated_act": true,
  "layer_norm_epsilon": 1e-06,
  "model_type": "t5",
  "num_decoder_layers": 12,
  "num_heads": 12,
  "num_layers": 12,
  "output_past": true,
```

Sprawdźmy jak dane wyglądają po tokenizacji:

```
  "relative_attention_num_buckets": 32,
```

```
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, 39
[13862, 20622, 2178, 18204, 308, 8439, 2451, 1]
174
8
```

```
  "_name_or_path": "allegro/plt5-base",
```

Wykorzystywany przez nas model obsługuje teksty od długości do 512 sub-tokenów. Konieczne jest zatem sprawdzenie, czy w naszych danych nie ma tekstów od większej długości.

Zadanie 5 (1 punkt)

Stwórz histogramy prezentujące rozkład długości tekstów wejściowych (input_ids) oraz odpowiedzi (label) dla zbioru treningowego. Zinterpretuj otrzymane wyniki.

```
  "feed_forward_proj": "gated-gelu",
```

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

# your code goes here
def plot_hist(data, title):
  plt.hist(data, bins=50)
  plt.title(title)
  plt.show()

plot_hist([len(e) for e in tokenized_datasets["train"]["input_ids"]], "Input ids")
plot_hist([len(e) for e in tokenized_datasets["train"]["labels"]], "Labels")
```

Ponieważ nasz model obsługuje tylko teksty do długości 512 znaków, 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 ograniczamy do 256 tokenów, ze wzgędu na ograniczenia pamięciowe (zajętość pamięci dla modelu jest proporcjonalna do kwadratu długości tekstu).

```
def preprocess_function(examples):
  result = tokenizer(examples["text"], truncation=True, max_length=256)
  targets = tokenizer(
    examples["labels"], truncation=True, max_length=32, padding=True
  )
  input_ids = [
    [(l if l != 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)
```

```
100%  31/31 [00:23<00:00, 1.77ba/s]
100%  4/4 [00:02<00:00, 1.69ba/s]
```

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, 42404, 305, 373, 1525, 15643, 291, 2958, 273, 19605, 6869, 271, 298, 2256, 7465, 39
[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, -1
174
32
```

Dla problemu odpowiadania na pytania potrzebować będziemy innego pre-trenowanego modelu oraz innego przygotowania danych. Jako model bazowy wykrzystamy ~~wielojęzyczny~~ polski wariant modelu T5 - ~~mT5~~ 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 ~~mT5~~ plT5 nie przeprowadzono jednak takiego dodatkowego procesu.

```
from transformers import AutoModelForSeq2SeqLM

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

```
loading configuration file config.json from cache at /root/.cache/huggingface/hub/models--allegro--plt5-base/snapshots/56379680948ce8b42d3d48df865
Model config T5Config {
  "_name_or_path": "allegro/plt5-base",
  "architectures": [
    "T5ForConditionalGeneration"
  ],
  "d_ff": 2048,
  "d_kv": 64,
  "d_model": 768,
  "decoder_start_token_id": 0,
  "dense_act_fn": "gelu_new",
  "dropout_rate": 0.1,
  "eos_token_id": 1,
  "feed_forward_proj": "gated-gelu",
  "initializer_factor": 1.0,
  "is_encoder_decoder": true,
  "is_gated_act": true,
  "layer_norm_epsilon": 1e-06,
  "model_type": "t5",
  "num_decoder_layers": 12,
  "num_heads": 12,
  "num_layers": 12,
  "output_past": true,
```

Trening modelu QA

```
  "tokenizer_class": "T5Tokenizer"
```

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.

```
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
  decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)
  labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
  decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
  print(decoded_preds[0])
  print(decoded_labels[0])

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

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

  return result
```

```
Downloading builder script: 100%  5.67k/5.67k [00:00<00:00, 348kB/s]
Downloading builder script: 100%  5.94k/5.94k [00:00<00:00, 321kB/s]
Downloading extra modules:  4.07k/? [00:00<00:00, 239kB/s]
Downloading extra modules: 100%  3.34k/3.34k [00:00<00:00, 200kB/s]
```

Zadanie 7 (1 punkty)

Korzystając z klasy Seq2SeqTrainingArguments zdefiniuj następujące parametry trenignu:

- liczba epok: 3
- wielkość paczki: 16

- ewaluacja co 100 kroków,
- szybkość uczenia: ~~5e-05~~ 1e-4
- optymalizator: adafactor
- maksymalna długość generowanej odpowiedzi: 32,
- akumulacja wyników ewaluacji: 4
- generowanie wyników podczas ewaluacji

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

```
# your code
training_args = Seq2SeqTrainingArguments(
    output_dir="output_qa",
    do_train=True,
    do_eval=True,
    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",
    predict_with_generate=True,
    generation_max_length=32,
    eval_accumulation_steps=4,
    evaluation_strategy="steps",
    logging_first_step=True,
    logging_strategy="steps",
    logging_steps=50,
    save_strategy="epoch",
    fp16=True,
)
```

```
PyTorch: setting up devices
The default value for the training argument `--report_to` will change in v5 (from all installed integrations to none). In v5, you will need to use `--report_to a
```

Zadanie 8 (1 punkt)

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

Obiekt ten powinien:

- wykorzystywać model ~~`mt5-base`~~ `allegro/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 goes here
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["dev"],
    data_collator=DataCollatorWithPadding(tokenizer),
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)
```

```
Using cuda_amp half precision backend
```

```
%load_ext tensorboard
%tensorboard --logdir output_qa/runs
```

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

403. That’s an error.

That’s all we know.

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!

```
trainer.train()
```