Metody przetwarzania języka naturalnego

Laboratorium 6 – Klasyfikacja i NER z LLM

mgr inż. Aleksandra Kwiatkowska

1. Wprowadzenie

1.1. Cel instrukcji

Celem niniejszej instrukcji jest zapoznanie się z nowoczesnymi metodami przetwarzania języka naturalnego opartymi na dużych modelach językowych (ang. *Large Language Models*, LLM) oraz ich zastosowaniem w zadaniach klasyfikacji tekstu i rozpoznawania nazwanych encji. Na poprzednich zajęciach omówiono klasyczne metody uczenia maszynowego stosowane w klasyfikacji tekstu, takie jak Naive Bayes, SVM, sieci neuronowe oraz lasy losowe. Metody te, choć skuteczne w wielu zastosowaniach, wymagają zazwyczaj ręcznej inżynierii cech, dużych zbiorów danych treningowych oraz oddzielnego trenowania modeli dla każdego nowego zadania. Duże modele językowe stanowią przełomowe podejście, które pozwala na wykonywanie zadań przetwarzania języka naturalnego w sposób bardziej uniwersalny, elastyczny i często nie wymagający lub wymagający znacznie mniejszej ilości danych treningowych.

1.2. Czym są duże modele językowe?

Duże modele językowe to zaawansowane modele uczenia maszynowego, które zostały wytrenowane na ogromnych ilościach danych tekstowych w celu zrozumienia i generowania języka naturalnego. Termin „duże" odnosi się zarówno do wielkości zbioru treningowego, jak i do liczby parametrów modelu, która może sięgać od setek milionów do setek miliardów. LLM bazują na architekturze transformera i wykorzystują mechanizm uwagi (ang. *attention mechanism*), który pozwala im modelować skomplikowane zależności między słowami w tekście, niezależnie od ich wzajemnej odległości.

Kluczową cechą LLM jest ich zdolność do uczenia się reprezentacji języka w sposób samonadzorowany. Oznacza to, że modele te są najpierw trenowane na ogólnym zadaniu przewidywania kolejnych słów w tekście lub odtwarzania zamaskowanych fragmentów, bez potrzeby ręcznego etykietowania danych. W trakcie tego procesu, zwanego pre-trainingiem, model nabywa szerokiej wiedzy o strukturze języka, gramatyce, semantyce zawartych w danych treningowych. Następnie, wytrenowany model może być dostosowany do specyficznych zadań poprzez proces fine-tuningu na mniejszych, zadaniowo-specyficznych zbiorach danych, lub może być wykorzystywany bezpośrednio poprzez odpowiednie formułowanie zapytań (ang. *prompting*).

LLM różnią się fundamentalnie od wcześniej omawianych metod klasyfikacji. Podczas gdy tradycyjne modele uczą się mapowania cech wejściowych na kategorie wyjściowe dla konkretnego zadania, LLM uczą się ogólnej reprezentacji języka, która może być wykorzystana do wielu różnych zadań. Ta uniwersalność wynika z faktu, że modele te „rozumieją" kontekst, znaczenie słów i ich relacje w sposób znacznie bardziej zaawansowany niż klasyczne metody oparte na bag-of-words czy prostych embedingach słów.

1.3. Zastosowania dużych modeli językowych

Duże modele językowe znalazły szerokie zastosowanie w niemal wszystkich obszarach przetwarzania języka naturalnego. Ich uniwersalność i elastyczność sprawiają, że można je wykorzystywać zarówno w zadaniach, które były tradycyjnie rozwiązywane za pomocą specjalistycznych modeli, jak i w zupełnie nowych aplikacjach.

W kontekście klasyfikacji tekstu, LLM mogą być używane do kategoryzacji dokumentów, analizy sentymentu, detekcji spamu, moderacji treści czy klasyfikacji intencji użytkownika w systemach dialogowych. W przeciwieństwie do klasycznych metod, LLM często pozwalają na klasyfikację zero-shot lub few-shot, czyli wykonywanie zadania bez żadnych lub z bardzo niewielką ilością przykładów treningowych. Wystarczy odpowiednio sformułować zadanie w języku naturalnym, a model jest w stanie je wykonać na podstawie swojej ogólnej wiedzy językowej. W przypadku rozpoznawania nazwanych encji (NER), LLM mogą identyfikować i klasyfikować encje takie jak osoby, miejsca, organizacje, daty, liczby czy specjalistyczne terminy techniczne i medyczne. Tradycyjne systemy NER wymagały ręcznego tworzenia słowników encji i reguł ekstrakcji, podczas gdy LLM mogą rozpoznawać encje na podstawie kontekstu i swojej wiedzy uzyskanej podczas treningu.

Poza klasyfikacją i NER, LLM znajdują zastosowanie w tłumaczeniu maszynowym, streszczaniu tekstu, odpowiadaniu na pytania, generowaniu tekstu, analizie semantycznej, ekstrakcji informacji, tworzeniu chatbotów i asystentów konwersacyjnych, analizie kodu źródłowego, wspomaganiu pisania czy wsparciu

w zadaniach kreatywnych. Warto również zauważyć, że LLM coraz częściej są wykorzystywane jako komponenty większych systemów, gdzie współpracują z innymi modelami i bazami danych. Przykładem może być Retrieval-Augmented Generation (RAG), gdzie LLM łączy swoją wiedzę z informacjami pobranymi z zewnętrznych źródeł, co pozwala na tworzenie bardziej dokładnych i aktualnych odpowiedzi.

1.4. Najpopularniejsze LLM do zadań klasyfikacji i NER

Wybór odpowiedniego modelu językowego zależy od wielu czynników, takich jak specyfika zadania, dostępne zasoby obliczeniowe, wymagania dotyczące prywatności danych czy potrzeba fine-tuningu. Współcześnie istnieje wiele dostępnych modeli, które różnią się rozmiarem, architekturą i sposobem udostępnienia.

Do najpopularniejszych modeli typu encoder-only, szczególnie przydatnych w zadaniach klasyfikacji i NER, należy rodzina modeli **BERT** (*Bidirectional Encoder Representations from Transformers*) i jej warianty. Model BERT, opublikowany przez Google w 2018 roku, stał się fundamentem dla wielu rozwiązań w NLP. Działa on poprzez analizę kontekstu zarówno od lewej, jak i prawej strony każdego słowa, co pozwala na głębokie zrozumienie znaczenia. Do popularnych wariantów BERT należą RoBERTa, DistilBERT, ALBERT czy modele językowe specyficzne dla poszczególnych języków, takie jak HerBERT dla języka polskiego.

W przypadku zadań wymagających zarówno rozumienia, jak i generowania tekstu, stosuje się modele typu encoder-decoder lub decoder-only. Do tej kategorii należą modele **GPT** (*Generative Pre-trained Transformer*) rozwijane przez OpenAI, włączając GPT-3, GPT-4 i nowsze wersje. Modele te, choć pierwotnie zaprojektowane do generowania tekstu, mogą być bardzo skuteczne w zadaniach klasyfikacji i NER poprzez odpowiednie formułowanie promptów.

Claude (model firmy Anthropic), LLaMA (rodzina modeli Meta), Mistral i Mixtral (modele firmy Mistral AI) to przykłady nowoczesnych modeli decoder-only, które osiągają doskonałe wyniki w różnorodnych zadaniach NLP. Modele te często są dostępne w różnych rozmiarach, od kilku miliardów do kilkuset miliardów parametrów, co pozwala na wybór optymalnego kompromisu między wydajnością a jakością. Dla zadań specjalistycznych powstały również modele domenowe, takie jak BioBERT czy SciBERT dla tekstów naukowych i medycznych, czy FinBERT dla tekstów finansowych. Modele te są najpierw trenowane na ogólnych danych, a następnie dodatkowo na danych domenowych, co pozwala im osiągać lepsze wyniki w specjalistycznych zastosowaniach.

Wybór między modelami open-source (takimi jak LLaMA, Mistral, czy BERT) a komercyjnymi API (jak GPT-4 czy Claude) zależy od wymagań projektu. Modele open-source oferują pełną kontrolę i możliwość fine-tuningu, ale wymagają własnej infrastruktury i zasobów obliczeniowych. Komercyjne API są łatwiejsze w użyciu i nie wymagają własnego sprzętu, ale wiążą się z kosztami wykorzystania i potencjalnymi ograniczeniami dotyczącymi prywatności danych.

2. Podstawy funkcjonowania LLM

2.1. Zasada działania LLM

Duże modele językowe opierają się na paradygmacie uczenia głębokiego i wykorzystują sieci neuronowe o bardzo dużej liczbie warstw i parametrów. Fundamentalną zasadą działania LLM jest uczenie się statystycznych wzorców występujących w języku poprzez przetwarzanie ogromnych ilości danych tekstowych. Model nie jest wprost programowany do wykonywania konkretnych zadań, lecz uczy się ich samodzielnie na przykładach.

Proces uczenia LLM można podzielić na kilka etapów:

1) Pre-training - model jest trenowany na bardzo dużym, nieoznaczonym korpusie tekstów przy użyciu zadań samonadzorowanych. Najpopularniejszym zadaniem dla modeli typu encoder (jak BERT) jest Masked Language Modeling (MLM), gdzie losowe słowa w tekście są maskowane, a model uczy się ich przewidywać na podstawie kontekstu. W przypadku modeli typu decoder (jak GPT) stosuje się Causal Language Modeling (CLM), gdzie model uczy się przewidywać kolejne słowo w sekwencji na podstawie poprzedzających słów.
2) Reprezentacja tekstu - zamiast traktować słowa jako dyskretne symbole, LLM przekształcają je w ciągłe wektory liczb zwane embedingami. Te wektory są następnie przetwarzane przez kolejne warstwy sieci, z których każda dodaje coraz bardziej abstrakcyjne i kontekstowe informacje. W przeciwieństwie do prostych

embeddingów słów (takich jak Word2Vec), gdzie każde słowo ma stałą reprezentację, LLM tworzą kontekstowe embedddingi, gdzie reprezentacja słowa zmienia się w zależności od otaczającego kontekstu.

3) Mechanizm uwagi – kluczowy element architektury transformerów. Pozwala on modelowi skupić się na różnych częściach tekstu wejściowego podczas przetwarzania każdego elementu. Gdy model analizuje dane słowo, mechanizm uwagi oblicza, jak bardzo to słowo powinno „zwracać uwagę" na każde inne słowo w sekwencji. To umożliwia modelowanie długodystansowych zależności w tekście, co było trudne dla wcześniejszych architektur, takich jak sieci rekurencyjne (RNN) czy LSTM.
4) Fine-tuning - po pre-trainingu model posiada ogólną wiedzę językową, ale może być dodatkowo dostosowany do specyficznych zadań. W tym procesie model jest dalej trenowany na mniejszym zbiorze danych specyficznym dla danego zadania, na przykład na oznaczonych przykładach klasyfikacji sentymentu. Alternatywnie, można wykorzystać nowoczesne techniki takie jak *prompt engineering*, gdzie zadanie jest formułowane jako pytanie lub instrukcja w języku naturalnym, a model odpowiada bez dodatkowego treningu. Inne podejścia to *few-shot learning*, gdzie modelowi podaje się kilka przykładów zadania w prompcie, oraz *zero-shot learning*, gdzie model wykonuje zadanie bez żadnych przykładów, jedynie na podstawie opisu zadania.

2.2. Tokenizacja i reprezentacja tekstu

Zanim tekst może być przetworzony przez LLM, musi zostać przekształcony w format zrozumiały dla modelu. Proces ten rozpoczyna się od tokenizacji, czyli dzielenia tekstu na mniejsze jednostki zwane tokenami. Token nie musi być pojedynczym słowem, może to być fragment słowa, całe słowo, znak interpunkcyjny, a nawet pojedynczy znak, w zależności od zastosowanej metody tokenizacji.

Najczęściej stosowane metody tokenizacji w LLM to Byte Pair Encoding (BPE) oraz WordPiece. Metody te operują na poziomie subword, co oznacza, że częste słowa mogą być reprezentowane jako pojedyncze tokeny, podczas gdy rzadsze słowa są dzielone na mniejsze, częściej występujące fragmenty. To podejście rozwiązuje problem out-of-vocabulary words - słów nieznanych modelowi - ponieważ prawie każde słowo można zbudować z mniejszych, znanych fragmentów.

Po tokenizacji każdy token jest przekształcany w wektor liczbowy poprzez embedding layer. Jest to wyuczona macierz, która przypisuje każdemu tokenowi z słownika wielowymiarowy wektor liczb rzeczywistych. Te wektory są początkowo inicjalizowane losowo, a następnie uczone podczas treningu tak, aby tokeny o podobnym znaczeniu lub występujące w podobnych kontekstach miały podobne reprezentacje wektorowe. Oprócz embeddingu tokena, modele dodają również informacje o pozycji tokena w sekwencji. Ponieważ architektura transformera przetwarza wszystkie tokeny równolegle, nie ma wbudowanego pojęcia kolejności słów. Positional encoding rozwiązuje ten problem przez dodanie do embeddingu tokena dodatkowego wektora, który koduje informację o pozycji. Mogą to być wyuczone wektory pozycyjne lub pozycyjne enkodowania oparte na funkcjach trygonometrycznych, które pozwalają modelowi zrozumieć, który token jest pierwszy, drugi i tak dalej.

Połączenie embeddingu tokena i kodowania pozycji tworzy finalną reprezentację wejściową, która jest następnie przekazywana do kolejnych warstw transformera. Ta reprezentacja zawiera zarówno informację o tym, co dany token oznacza, jak i gdzie znajduje się w tekście, co jest kluczowe dla zrozumienia struktury i znaczenia całej sekwencji.

3. Architektura transformera

3.1. Wprowadzenie do architektury

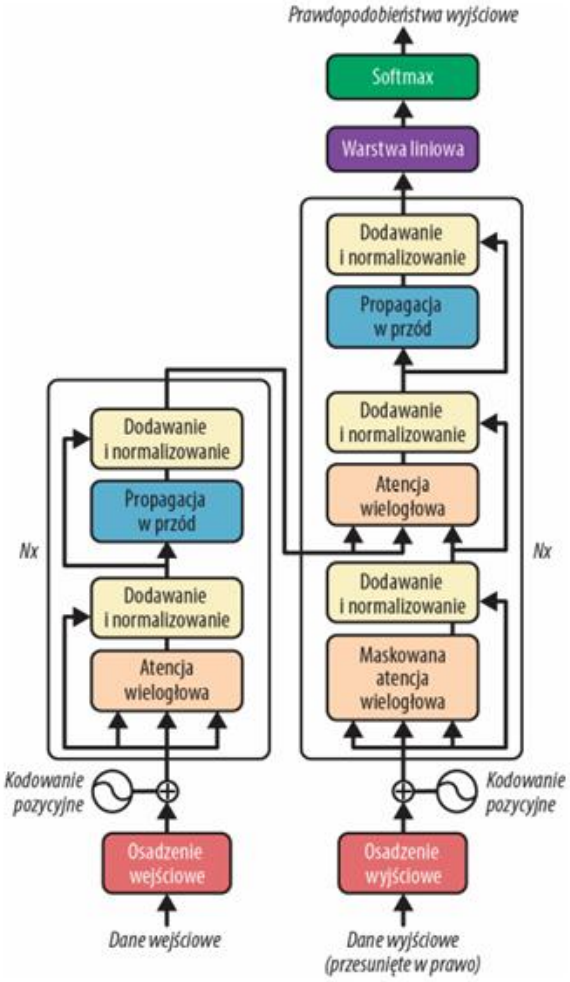
Architektura transformera, zaproponowana w przełomowej pracy *Attention is All You Need* przez Vaswaniego i współpracowników w 2017 roku, stanowi fundament wszystkich współczesnych dużych modeli językowych. Transformer całkowicie odszedł od wcześniejszych sekwencyjnych architektur, takich jak sieci rekurencyjne (RNN) i LSTM, wprowadzając przetwarzanie równoległe i mechanizm uwagi jako główny sposób modelowania zależności w tekście.

Oryginalny transformer składa się z dwóch głównych komponentów:

- Enkoder przetwarza sekwencję wejściową i tworzy jej głęboką, kontekstową reprezentację,
- Dekoder wykorzystuje tę reprezentację do generowania sekwencji wyjściowej.

Taka architektura encoder-decoder jest stosowana w zadaniach typu *sequence-to-sequence*, takich jak tłumaczenie maszynowe. Jednak wiele nowoczesnych LLM wykorzystuje tylko jedną z tych części. Modele typu encoder-only (jak BERT) używają tylko enkodera i są szczególnie dobre w zadaniach wymagających rozumienia tekstu, takich jak klasyfikacja czy NER. Modele decoder-only (jak GPT) stosowane są głównie do generowania tekstu.

Podstawową jednostką budulcową transformera jest warstwa (ang. *layer*), a modele mogą mieć od kilku do kilkudziesięciu. Każda warstwa transformuje reprezentację wejściową, dodając coraz bardziej abstrakcyjne i semantyczne informacje. Głębokość modelu, czyli liczba warstw, wraz z innymi parametrami, takimi jak wymiar embeddingów czy liczba głowic uwagi, determinuje całkowitą liczbę parametrów modelu, która w przypadku największych LLM może przekraczać sto miliardów.



Architektura transformera

3.2. Mechanizm uwagi (ang. *Attention Mechanism*)

Mechanizm uwagi jest centralnym elementem architektury transformera i kluczem do jej sukcesu. Podstawową ideą uwagi jest umożliwienie modelowi dynamicznego skupiania się na różnych częściach sekwencji wejściowej podczas przetwarzania każdego elementu. W kontekście przetwarzania języka oznacza to, że dla każdego słowa model może określić, które inne słowa w zdaniu są najbardziej istotne dla jego interpretacji.

Mechanizm self-attention, stosowany w transformerze, działa poprzez porównanie każdego tokena z każdym innym tokenem w sekwencji. Dla każdego tokena obliczane są trzy wektory:

- *query* – zapytanie, który dany token zadaje innym tokenom,
- *key* - klucz, to „opis" każdego tokenu,
- *value* – wartość, oznaczająca rzeczywistą informację do przekazania.

Wskaźnik atencji dla każdego słowa oblicza się ze wzoru:

$$Attention(Q, K, V) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- Q (Query) – zapytanie
- K (Key) – klucz
- V (Value) – wartość
- d_k – liczba wymiarów klucza (dla normalizacji wyniku)
- softmax – zamienia wyniki dopasowań na rozkład prawdopodobieństwa (czyli macierz wag uwagi)

Proces obliczania uwagi dla danego tokena przebiega następująco: wektor query tego tokena jest porównywany (poprzez iloczyn skalarny) z wektorami key wszystkich tokenów w sekwencji. Te porównania określają, jak bardzo dany token powinien „zwrócić uwagę" na każdy inny token. Wyniki są następnie normalizowane poprzez funkcję softmax, aby otrzymać wagi uwagi sumujące się do jedności. Te wagi określają, jak bardzo każdy token przyczynia się do finalnej reprezentacji analizowanego tokena. Ostatecznie, wagi te są używane do obliczenia ważonej sumy wektorów value wszystkich tokenów, tworząc nową reprezentację kontekstową dla rozpatrywanego tokena.

Kluczową zaletą tego mechanizmu jest to, że działa on w sposób równoległy dla wszystkich tokenów jednocześnie. Wszystkie porównania query-key mogą być obliczone jako operacje macierzowe, co pozwala na efektywne wykorzystanie nowoczesnych akceleratorów obliczeniowych, takich jak GPU czy TPU. Ta równoległość sprawia, że transformery są znacznie szybsze w treningu i inferencji niż sekwencyjne architektury RNN.

3.3. Multi-Head Attention

W praktyce transformer nie stosuje pojedynczego mechanizmu uwagi, lecz wykorzystuje *multi-head attention*, czyli wiele równoległych mechanizmów uwagi działających jednocześnie. Każda *attention head* uczy się skupiać na innych aspektach relacji między tokenami. Niektóre głowice mogą nauczyć się relacji syntaktycznych, takich jak związek między podmiotem a orzeczeniem, inne mogą wychwytywać relacje semantyczne, jeszcze inne mogą skupiać się na zależnościach długodystansowych. Technicznie, *multi-head attention* działa poprzez podział wymiaru embeddingu na mniejsze części, z których każda jest przetwarzana przez osobną głowicę uwagi. Na przykład, jeśli wymiar embeddingu wynosi 768 i używamy 12 głowic, każda głowica operuje na wektorach o wymiarze 64. Każda głowica ma swoje własne macierze wag do obliczania *query*, *key* i *value*, co pozwala jej uczyć się różnych wzorców uwagi. Wyniki ze wszystkich głowic są następnie konkatenowane i przekształcane przez dodatkową macierz wag, tworząc ostateczną reprezentację wyjściową. To połączenie wielu perspektyw pozwala modelowi na bardziej bogate i wieloaspektowe rozumienie tekstu. *Multi-head attention* daje również modelowi redundancję. Jeśli jedna głowica nie wykryje istotnej relacji, inne mogą to zrobić. Ta redundancja zwiększa niezawodność i generalizację modelu, pozwalając mu skutecznie działać na różnorodnych danych i w różnych kontekstach.

3.4. Feed-Forward Networks

Po bloku *multi-head attention* każda warstwa transformera zawiera jeszcze jeden kluczowy komponent - **sieć feed-forward** (FFN). Jest to prosta, w pełni połączona sieć neuronowa stosowana niezależnie do każdej pozycji w sekwencji. Sieć ta składa się typowo z dwóch warstw liniowych z nieliniową funkcją aktywacji (najczęściej GELU lub ReLU) pomiędzy nimi.
Pierwsza warstwa liniowa zwykle zwiększa wymiarowość, często kilkukrotnie względem wymiaru embeddingu. Ta ekspansja pozwala modelowi na bardziej ekspresyjną transformację każdej reprezentacji tokena. Druga warstwa liniowa redukuje wymiar z powrotem do oryginalnego rozmiaru embeddingu. Ta architektura, pozwala modelowi na wykonywanie skomplikowanych nieliniowych transformacji przy zachowaniu stałego wymiaru reprezentacji. Pełna warstwa transformera łączy *multi-head attention* i feed-forward network w sposób sekwencyjny, z dodatkowymi elementami stabilizującymi trening. Po każdym z tych dwóch podmodułów stosowane jest:

- Połączenie rezydualne polega na dodaniu wejścia podmodułu do jego wyjścia. Połączenia rezydualne służą kilku celom. Po pierwsze, ułatwiają przepływ gradientów podczas treningu, co jest kluczowe w bardzo głębokich sieciach - umożliwiają one bezpośrednią ścieżkę propagacji sygnału i gradientu przez wiele warstw, zapobiegając problemowi zanikających gradientów. Po drugie, pozwalają każdej warstwie skupić się na uczeniu się różnicy względem wejścia, a nie całkowicie nowej reprezentacji, co często prowadzi do szybszego i bardziej stabilnego treningu.
- Normalizacja warstw, która normalizuje aktywacje w ramach pojedynczego przykładu i pojedynczej warstwy, standaryzując je do średniej zero i wariancji jeden. To stabilizuje trening i pozwala na używanie większych *learning rate*. W niektórych nowszych implementacjach transformera normalizacja jest stosowana przed podmodułami (Pre-Layer Normalization) zamiast po nich, co okazało się jeszcze bardziej stabilne w przypadku bardzo głębokich modeli.

4. Biblioteka *transformers*

Biblioteka *transformers* od Hugging Face stanowi obecnie standard w dostępie do dużych modeli językowych. Jej architektura opiera się na trzech głównych warstwach abstrakcji, które umożliwiają pracę z modelami

na różnych poziomach zaawansowania. Najwyższym poziomem jest API pipeline, które ukrywa szczegóły implementacyjne i pozwala na wykonywanie zadań NLP za pomocą kilku linii kodu. Średni poziom to klasy Auto, które automatycznie dobierają odpowiednią architekturę modelu na podstawie jego nazwy. Najniższy poziom stanowią konkretne klasy modeli i tokenizerów, dające pełną kontrolę nad wszystkimi aspektami przetwarzania.

Biblioteka została zaprojektowana z myślą o spójności interfejsów - niezależnie od tego, czy pracuje się z modelem BERT, GPT czy T5, API pozostaje podobne. To znacznie ułatwia eksperymentowanie z różnymi modelami i przechodzenie między nimi. Wszystkie modele w bibliotece są zorganizowane wokół wspólnego wzorca: tokenizacja tekstu, przekazanie tokenów do modelu, otrzymanie wyników i ich postprocessing.

4.1. Pipeline - API wysokiego poziomu

Pipeline jest najprostszym i najbardziej zalecanym sposobem rozpoczęcia pracy z modelami językowymi. Enkapsuluje on cały proces przetwarzania: od załadowania modelu i tokenizera, przez tokenizację i inference, aż po przekształcenie wyników do czytelnej formy.
Funkcja *pipeline()* jest głównym punktem wejścia do korzystania z gotowych modeli. Przyjmuje ona następujące argumenty:

- *task (str, wymagany)* - określa rodzaj zadania NLP do wykonania. Dostępne wartości to między innymi:
  - *"sentiment-analysis"* lub *"text-classification"* dla klasyfikacji tekstu,
  - *"ner"* lub *"token-classification"* dla rozpoznawania nazwanych encji,
  - *"question-answering"* dla odpowiadania na pytania,
  - *"summarization"* dla streszczania tekstu,
  - *"translation"* dla tłumaczenia maszynowego,
  - *"text-generation"* dla generowania tekstu,
  - *"fill-mask"* dla uzupełniania zamaskowanych słów,
  - *"zero-shot-classification"* dla klasyfikacji bez przykładów treningowych,
- *model (str lub model object, opcjonalny)* - nazwa modelu z Hugging Face Hub lub lokalnie zapisany model. Jeśli nie zostanie podany, pipeline użyje domyślnego modelu dla danego zadania.
- *tokenizer (str lub tokenizer object, opcjonalny)* - tokenizer do użycia. Jeśli nie podano, zostanie automatycznie dobrany na podstawie modelu.
- *device (int, opcjonalny)* - określa urządzenie do obliczeń. Wartość 0 oznacza pierwsze GPU, -1 oznacza CPU (domyślnie), liczby dodatnie wskazują konkretne GPU w systemach wieloprocesorowych.
- *framework (str, opcjonalny)* - framework do użycia: "pt" dla PyTorch lub "tf" dla TensorFlow. Domyślnie używany jest zainstalowany framework.

```
from transformers import pipeline
from transformers import AutoTokenizer, AutoModelForSequenceClassification

classifier = pipeline(
    "sentiment-analysis",
    model="distilbert-base-uncased-finetuned-sst-2-english",
    device=0,
    tokenizer=AutoTokenizer.from_pretrained("bert-base-uncased")
)
```

Po utworzeniu pipeline można go wywołać jak funkcję, przekazując dane do przetworzenia. Pipeline akceptuje różne formy danych wejściowych i zwraca wyniki w standardowym formacie.
Argumenty wywołania pipeline:

- *inputs (str, List[str] lub Dict)* - dane wejściowe do przetworzenia. Może to być pojedynczy tekst, lista tekstów lub słownik z dodatkowymi parametrami dla niektórych zadań.
- *batch_size (int, opcjonalny)* - rozmiar batcha przy przetwarzaniu wielu przykładów. Większe wartości przyspieszają obliczenia, ale wymagają więcej pamięci.
- *top_k (int, opcjonalny)* - dla zadań klasyfikacji określa liczbę najlepszych klas do zwrócenia. Wartość None zwraca wszystkie klasy.
- *truncation (bool, opcjonalny)* - czy obciąć tekst przekraczający maksymalną długość modelu. Domyślnie True.
- *padding (bool lub str, opcjonalny)* - strategia paddingu dla tekstów krótszych niż maksymalna długość.

4.2. AutoClasses – automatyczny dobór architektury

Klasy Auto stanowią warstwę pośrednią, która automatycznie dobiera odpowiednią architekturę modelu na podstawie jego nazwy lub konfiguracji. Są to klasy fabrykujące (ang. *factory classes*), które ukrywają szczegóły konkretnych implementacji.

AutoTokenizer automatycznie ładuje odpowiedni tokenizer dla danego modelu. Główną metodą jest *from_pretrained()*.
AutoTokenizer.from_pretrained() przyjmuje następujące argumenty:

- *pretrained_model_name_or_path (str, wymagany)* - nazwa modelu z Hugging Face lub ścieżka do lokalnego modelu.
- *cache_dir (str, opcjonalny)* - katalog do cache'owania pobranych modeli.
- *force_download (bool, opcjonalny)* - czy wymusić ponowne pobranie, ignorując cache.
- *use_fast (bool, opcjonalny)* - czy użyć szybkiej implementacji tokenizera w Rust (jeśli dostępna).
- *padding_side (str, opcjonalny)* - strona do paddingu: "right" lub "left".

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(
    "gpt2",
    use_fast=True,
    padding_side="left"
)
```

4.3. Pełny pipeline ręczny

Dla lepszego zrozumienia działania biblioteki warto zobaczyć, co pipeline robi pod spodem Poniżej przedstawiono ręczną implementację procesu klasyfikacji:

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import torch.nn.functional as F

# 1. Załadowanie tokenizera i modelu
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased-finetuned-sst-2-english")
model    =    AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased-finetuned-sst-2-
english")

# 2. Przygotowanie tekstu
text = "I love this product!"

# 3. Tokenizacja
inputs = tokenizer(
    text,
    return_tensors="pt",
    padding=True,
    truncation=True,
    max_length=512
)

# 4. Inference (bez obliczania gradientów)
model.eval()
with torch.no_grad():
    outputs = model(**inputs)

# 5. Ekstrakcja logits (surowych predykcji)
logits = outputs.logits
print("Logits:", logits)

# 6. Konwersja logits na prawdopodobieństwa (softmax)
probabilities = F.softmax(logits, dim=-1)
print("Probabilities:", probabilities)

# 7. Wybranie klasy z największym prawdopodobieństwem
```

```
predicted_class_id = logits.argmax(dim=-1).item()
print("Predicted class ID:", predicted_class_id)  # 1

# 8. Mapowanie ID na nazwę klasy
id2label = model.config.id2label
predicted_label = id2label[predicted_class_id]
confidence = probabilities[0][predicted_class_id].item()

print(f"Prediction: {predicted_label}, Confidence: {confidence:.4f}")
```

5. Zadania do wykonania

Zadanie 1: Analiza sentymentu recenzji filmowych

Plik movie_reviews.csv zawiera recenzje filmowe. Ma następującą strukturę:
- Kolumna *review* - treść recenzji filmowej (tekst)
- Kolumna *sentyment* - prawdziwa etykieta sentymentu (positive/negative)

Kroki:
1. Wczytać dane z pliku CSV.
   - Załadować plik movie_reviews.csv używając pandas.read_csv()
   - Sprawdzić strukturę danych (liczba recenzji, rozkład klas)
   - Zweryfikować czy nie ma brakujących wartości
2. Załadować model.
   - Utworzyć pipeline z modelem DistilBERT: *distilbert-base-uncased-finetuned-sst-2-english*
   - Ustawić device (GPU/CPU) i truncation=True
3. Sklasyfikować wszystkie recenzje używając modelu.
   - Przekonwertować kolumnę *review* na listę tekstów
   - Wywołać classifier z batch_size=8 dla wydajności
   - Każdy wynik zawiera label (POSITIVE/NEGATIVE) i score (pewność 0-1)
4. Przygotować wyniki do ewaluacji.
   - Wyekstraktować predykcje
   - Wyekstraktować pewność
   - Dodać kolumny “predicted_sentiment” i “confidence” do DataFrame
   - Upewnić się, że prawdziwe etykiety są w tym samym formacie (*lowercase*)
5. Obliczyć metryki klasyfikacji.
   - Użyć sklearn.metrics do obliczenia metryk
   - Wyświetlić szczegółowy classification_report

Zadanie 2: Ekstrakcja informacji z tekstów

Kroki:
1. Wczytać dane z pliku CSV.
   - Załadować plik ag_news_articles.csv używając pandas.read_csv()
   - Sprawdzić strukturę danych (liczba artykułów, rozkład kategorii)
     1 - reprezentuje *World*, 2 reprezentuje *Sports*, 3 reprezentuje *Business* and 4 reprezentuje *Sci/Tech*.
2. Załadownie modelu.
   - Utworzyć pipeline z modelem *dslim/bert-base-NER*
   - Parametry pipeline**:** task="ner”, model="dslim/bert-base-NER", device=0 aggregation_strategy="simple".
3. Wykonać NER na wszystkich artykułach
4. Wyekstraktować wszystkie encje (PER, ORG, LOC, MISC)
5. Pogrupować encje według kategorii artykułów
6. Przeanalizować rozkład encji (ile każdego typu w każdej kategorii)
7. Analiza pewności rozpoznania encji