Natural Language Processing (NLP)

NLP (ang. Natural Language Processing), czyli Przetwarzanie Języka Naturalnego. Jej głównym celem jest umożliwienie komputerom rozumienia, interpretowania i generowania ludzkiego języka w sposób, który jest wartościowy i sensowny. NLP pozwala na automatyczne przeszukiwanie tysięcy raportów.

Cele projektu:

- Poznanie podstaw przetwarzania języka naturalnego (NLP) – jak analizować raporty geologiczne lub dane z misji kosmicznych.
- Praktyczna nauka tokenizacji, pad_sequences i embeddingów w Keras/TensorFlow.
- Tworzenie prostego modelu klasyfikacji tekstu, np. czy raport dotyczy złóż rud, minerałów czy eksploracji kosmosu.
- Zastosowanie Pipeline do automatyzacji przetwarzania danych.
- Analiza wyników i refleksja nad interpretacją modelu w kontekście Nauk o Ziemii.

Symulacja raportu geologicznego lub wyników misji kosmicznej - lista zdań

```
texts = [
    "Odkryto złoża żelaza w warstwach osadowych",
    "Badania geofizyczne wskazują anomalię magnetyczną",
    "Misja kosmiczna potwierdziła obecność minerałów na Marsie",
    "Wiercenia w rejonie rud miedzi wykazały wysoką koncentrację metalu",
    "Sonda wylądowała na powierzchni krateru uderzeniowego na Księżycu",
    "Analiza próbek skał wskazuje na wysoką zawartość krzemianów i bazaltu"
]
```

Najważniejsze słowa kluczowe do klasyfikacji:

- kosmos: ["misja", "Mars", "sonda", "asteroidzie"],
- geologia: ["złoża", "wiercenia", "rudy", "geofizyczne"].

Tokenizacja

Tokenizacja to proces, w którym każde słowo w zdaniu jest przypisywane do unikalnej liczby całkowitej, aby komputer mógł je przetwarzać. Zdania zamieniane są na listy liczb.

```
from tensorflow.keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(num_words=100)
tokenizer.fit_on_texts(texts)
print(tokenizer.word_index)
```

```
{'na': 1, 'w': 2, 'wysoką': 3, 'odkryto': 4, 'złoża': 5, 'żelaza': 6, 'warstwach': 7, 'osadowych': 8, 'badania': 9, 'geofizy
```

Jakie słowo otrzymało najmniejszy indeks? - Słowa występujące najczęściej (np. spójniki "na", "w") otrzymują zazwyczaj najniższe indeksy. Unikalne terminy, nie występujące w innych zdaniach (np. "bazalt"), otrzymują najwyższe indeksy w słowniku.

Zamiana tekstu na sekwencje liczbowych tokenów

```
sequences = tokenizer.texts_to_sequences(texts)
print(sequences)
```

```
[[4, 5, 6, 2, 7, 8], [9, 10, 11, 12, 13], [14, 15, 16, 17, 18, 1, 19], [20, 2, 21, 22, 23, 24, 3, 25, 26], [27, 28, 1, 29, 3
```

Każde zdanie jest teraz listą liczb całkowitych.
Każda liczba odpowiada słowu w słowniku (tokenizer.word_index).
Sekwencje mają różną długość – dokładnie taką, ile było słów w oryginalnym zdaniu.
Odkryto złoża żelaza w warstwach osadowych -> [4, 5, 6, 2, 7, 8]

Padding (Wyrównywanie długości)

Modele deep learningowe (=neuronowe) wymagają, aby wszystkie sekwencje wejściowe miały taką samą długość. Dlatego stosujemy padding, czyli dopasowujemy krótsze sekwencje do długości najdłuższej (lub ustalonej).

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

padded = pad_sequences(sequences, padding='post')
print(padded)
```

```
[[ 4  5  6  2  7  8  0  0  0  0]
 [ 9 10 11 12 13  0  0  0  0  0]
 [14 15 16 17 18  1 19  0  0  0]
 [20  2 21 22 23 24  3 25 26  0]
 [27 28  1 29 30 31  1 32  0  0]
 [33 34 35 36  1  3 37 38 39 40]]
```

- **Jak zmieniła się długość?** Wszystkie sekwencje mają teraz długość najdłuższego zdania w zbiorze (10 elementów).
- **Jakie liczby reprezentują brakujące miejsca (padding)?** Jest to liczba 0.
- **Dlaczego takie uzupełnienie jest potrzebne w modelach sieci neuronowych?** Warstwy wejściowe sieci neuronowych (np. Dense) wymagają macierzy o stałych wymiarach (tensors). Sieć nie potrafi "sama z siebie" przyjąć raz wektora 5-elementowego, a raz 10-elementowego w tej samej warstwie.

Tworzenie i trenowanie modelu

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense

# Rozszerzone dane
texts = [
    "Odkryto złoża żelaza w warstwach osadowych",
    "Wiercenia w rejonie rud miedzi wykazały wysoką koncentrację metalu",
    "Badania geofizyczne wykazały anomalię magnetyczną",
    "Misja kosmiczna potwierdziła obecność minerałów na Marsie",
    "Sonda wylądowała na powierzchni krateru",
    "Orbiter krąży wokół nowej egzoplanety"
]
labels = np.array([0, 0, 0, 1, 1, 1]) # 0-geologia, 1-kosmos

tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
max_length = max([len(x) for x in sequences])
padded = pad_sequences(sequences, maxlen=max_length, padding='post')

model = Sequential([
    Embedding(input_dim=100, output_dim=8, input_length=max_length),
    Flatten(),
    Dense(1, activation='sigmoid') # dla klasyfikacji binarnej
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(padded, labels, epochs=20, verbose=0)

predictions = model.predict(padded)
for i in range(len(texts)):
    print(f"Tekst: {texts[i]}")
    print(f"Prawdopodobieństwo: {predictions[i][0]:.4f} (Klasa: {labels[i]})\n")
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is depre
  warnings.warn(
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step
Tekst: Odkryto złoża żelaza w warstwach osadowych
Prawdopodobieństwo: 0.4743 (Klasa: 0)

Tekst: Wiercenia w rejonie rud miedzi wykazały wysoką koncentrację metalu
Prawdopodobieństwo: 0.4599 (Klasa: 0)

Tekst: Badania geofizyczne wykazały anomalię magnetyczną
Prawdopodobieństwo: 0.4674 (Klasa: 0)

Tekst: Misja kosmiczna potwierdziła obecność minerałów na Marsie
Prawdopodobieństwo: 0.5354 (Klasa: 1)

Tekst: Sonda wylądowała na powierzchni krateru
Prawdopodobieństwo: 0.5273 (Klasa: 1)

Tekst: Orbiter krąży wokół nowej egzoplanety
Prawdopodobieństwo: 0.5381 (Klasa: 1)
```

Jak embeddingi reprezentują słowa w przestrzeni liczbowej?

Embeddingi zamieniają słowa na współrzędne w wielowymiarowej przestrzeni, gdzie odległość między punktami odpowiada podobieństwu znaczeniowemu słów. Zamiast reprezentować słowo jako zwykły numer (jak w tokenizacji, gdzie "Mars" = 5, a "Ziemia" = 6), embedding zamienia je w wektor, czyli listę liczb zmiennoprzecinkowych (np. $[0.12, -0.54, 0.89, \ldots]$).

Najważniejszą cechą embeddingów jest to, że słowa, które występują w podobnych kontekstach, lądują blisko siebie w tej przestrzeni.

- Słowa "żelazo" i "miedź" będą miały podobne współrzędne, bo oba są metalami i pojawiają się w raportach geologicznych.
- Słowo "orbita" będzie znajdować się w zupełnie innej części tej przestrzeni, blisko słów "planeta" czy "satelita".

Czy model poprawnie odróżnia raporty geologiczne od kosmicznych?

Model z warstwą Dense(1, activation='sigmoid') zwraca wartość od 0 do 1:

- Wynik blisko 0.0 (np. 0.02) oznacza dużą pewność, że to geologia.
- Wynik blisko 1.0 (np. 0.98) oznacza dużą pewność, że to kosmos.
- Wynik w okolicy 0.5 oznacza, że model jest zdezorientowany i nie widzi wystarczającej liczby słów kluczowych, które zna z etapu nauki.

Wszystkie wyniki oscylują wokół 0.50 (różnice są na poziomie zaledwie 3-7%). Oznacza to, że model jest bardzo niepewny.

Testowanie nowego zdania

```
test_text = ["Nowe badania wskazują złoża niklu na asteroidzie"]
seq = tokenizer.texts_to_sequences(test_text)
padded_test = pad_sequences(seq, maxlen=max_length, padding='post')

prediction = model.predict(padded_test)
print(f"Wynik (blisko 0 = geologia, blisko 1 = kosmos): {prediction[0][0]:.4f}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step
Wynik (blisko 0 = geologia, blisko 1 = kosmos): 0.4876
```

Czy model prawidłowo sklasyfikował zdanie?

Nie, model nie sklasyfikował tego zdania prawidłowo.

Wynik ten znajduje się niemal dokładnie pośrodku skali (0.5). Oznacza to, że model jest „zdezorientowany" i w praktyce zgaduje. Dla zdania o asteroidzie (klasa 1 - kosmos), oczekiwalibyśmy wyniku powyżej 0.80.

Jak można poprawić działanie modelu przy nowych słowach?

Aby model lepiej radził sobie z nowymi słowami (np. "nikiel", jeśli go wcześniej nie było), należy trenować na znacznie większym korpusie tekstów.

- Przy tworzeniu Tokenizera warto dodać specjalny znacznik dla nieznanych słów. Dzięki temu model uczy się, że istnieją słowa spoza słownika i może przypisać im pewne znaczenie kontekstowe.
- Użycie Pre-trained Embeddings (np. Word2Vec lub GloVe): Zamiast uczyć się znaczenia słów od zera, można użyć gotowych "map" słów wygenerowanych na milionach dokumentów
- Lematyzacja i Stemming: Sprowadzenie słów do formy podstawowej. Dla komputera "złoża", "złożu" i "złoża" to trzy różne liczby. Po lematyzacji wszystkie stają się słowem "złoże". To drastycznie zmniejsza rozmiar słownika i ułatwia naukę.
- Augmentacja danych: Dodanie większej liczby przykładów. W NLP można to robić np. poprzez zamianę słów na synonimy (np. zamiast "złoża żelaza" – "pokłady ferrytu").

Pipeline i automatyzacja

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer

def tokenize_pad(X):
    # Ważne: używamy tokenizer.texts_to_sequences zdefiniowanego wcześniej
    seq = tokenizer.texts_to_sequences(X)
    return pad_sequences(seq, maxlen=max_length, padding='post')

pipeline = Pipeline([
    ('tokenize_pad', FunctionTransformer(tokenize_pad, validate=False)),
    ('model', model)
])
```

Jak pipeline chroni przed błędami przy nowych słowach?

1. Spójność transformacji: Pipeline chroni przed błędami, bo zapewnia, że nowe dane testowe przejdą identyczną ścieżkę transformacji (taka sama długość, ten sam słownik), co dane treningowe.
2. Izolacja słownika (Data Leakage): Pipeline wymusza, aby transformacja danych testowych opierała się wyłącznie na słowniku (word_index), który został zbudowany podczas treningu. Dzięki temu mamy realistyczny obraz tego, jak model radzi sobie z nieznanymi słowami.
3. Obsługa słów spoza słownika (OOV): Jeśli w nowym zdaniu pojawi się słowo, którego nie było w treningu to dzięki FunctionTransformer wewnątrz Pipeline'u, proces texts_to_sequences po prostu pominie nieznane słowo (lub zamieni je na token , jeśli go skonfigurowano).
4. Automatyzacja "czyszczenia": Jeśli w przyszłości zdecydujemy się dodać krok usuwania znaków interpunkcyjnych lub zamiany liter na małe, robimy to w jednym miejscu w Pipeline. Mamy wtedy pewność, że nowe dane testowe nie zostaną odrzucone tylko dlatego, że ktoś wpisał zdanie wielkimi literami.

Test pipeline'u

```
new_texts = ["Eksploracja Marsa wykazała obecność żelaza i niklu"]
print(f"Wynik (blisko 0 = geologia, blisko 1 = kosmos): {pipeline.predict(new_texts)[0][0]:.4f}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
Wynik (blisko 0 = geologia, blisko 1 = kosmos): 0.4940
/usr/local/lib/python3.12/dist-packages/sklearn/pipeline.py:62: FutureWarning: This Pipeline instance is not fitted yet. Cal
  warnings.warn(
```

Czy pipeline poprawnie przetworzył dane i dokonał predykcji?

Wynik 0.4940 ponownie wskazuje na to, że model jest całkowicie niepewny (wynik niemal idealnie pośrodku). Choć technicznie Pipeline zadziałał poprawnie (przyjął tekst, zamienił go na liczby i przeszedł przez model bez błędu), to sama predykcja jest bezużyteczna.

Jak można rozbudować pipeline o preprocessing, np. usuwanie stop-words lub stemming?

Aby model działał lepiej, musimy "wyczyścić" tekst, zanim trafi do modelu. Możemy to zrobić, dodając nową funkcję do naszego Pipeline'u.

1. Usuwanie Stop-words (słów nieznaczących) Słowa takie jak "i", "w", "na", "z" występują w obu kategoriach raportów. Ich usunięcie pozwala modelowi skupić się na "mięsie" (słowach kluczowych).
2. Stemming / Lematyzacja Dla modelu "Marsa" i "Marsie" to dwa różne słowa. Stemming ucina końcówki, sprowadzając je do wspólnego rdzenia "Mars".
3. Usuwanie znaków interpunkcyjnych
4. Usuwanie różnicy między wielkością liter

Rozbudowany Pipeline

```
import re

# Przykładowa lista polskich stop-words
STOP_WORDS = {"i", "w", "na", "z", "o", "u", "pod", "ponad", "wykazała", "obecność"}

def clean_text(texts):
    cleaned_texts = []
    for text in texts:
        # 1. Małe litery
        text = text.lower()
        # 2. Usuwanie znaków interpunkcyjnych
        text = re.sub(r'[^\w\s]', '', text)
        # 3. Usuwanie stop-words
        text = " ".join([word for word in text.split() if word not in STOP_WORDS])
        cleaned_texts.append(text)
    return cleaned_texts

# Nowy, rozbudowany Pipeline
from sklearn.preprocessing import FunctionTransformer

pipeline = Pipeline([
    ('text_cleaning', FunctionTransformer(clean_text, validate=False)),
    ('tokenize_pad', FunctionTransformer(tokenize_pad, validate=False)),
    ('model', model)
])
```