WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Marcin Przybylski

Bigram oraz word2vec
(Natural Language Processing)

PRACA LABORATORYJNA

Rzeszów, 2025

Spis treści

1. Wstęp

Celem niniejszego laboratorium było praktyczne zapoznanie się z technikami przetwarzania języka naturalnego, ze szczególnym uwzględnieniem modeli reprezentacji słów Word2Vec. Ćwiczenia obejmowały implementację i analizę modeli przy użyciu bibliotek Gensim oraz PyTorch (architektura CBOW), przetwarzanie różnych korpusów tekstowych (dialogi, 4-gramy), analizę implementacji "from scratch"oraz zastosowanie technik pomocniczych, takich jak klasteryzacja K-Means i wizualizacja za pomocą PCA/t-SNE.

Wszystkie zadania programistyczne zostały wykonane w środowisku Google Colaboratory. Poniżej znajdują się odnośniki do notatnika roboczego oraz do dokumentu zawierającego instrukcje i odnośniki do materiałów źródłowych wykorzystanych podczas laboratorium:

- **Notatnik Roboczy Colab:** `https://colab.research.google.com/drive/1rTfWR7hWEWjju0BuYifdH3jyyPwRUsrx?usp=sharing`
- **Instrukcja do Zajęć (Google Docs):** `https://docs.google.com/document/d/1lW9e3X6aJ5noTBodFizZG4t7JrMaU3oqDuQw-mNFksY/edit?tab=t.0`

Niniejsze sprawozdanie szczegółowo opisuje przebieg wykonanych zadań, prezentuje uzyskane wyniki oraz zawiera ich interpretację.

2. Zadanie 1: Model Bigram - Generacja Imion na Podstawie Częstości

Pierwsze zadanie polegało na zaimplementowaniu i przetestowaniu prostego modelu bigramowego do generowania imion. Model ten opiera się na zliczaniu częstości występowania par sąsiadujących znaków (bigramów) w dostarczonym zbiorze danych i wykorzystaniu tych częstości do obliczenia prawdopodobieństw przejść między znakami.

2.1. Wczytanie Danych i Budowa Modelu

Jako dane wejściowe wykorzystano plik tekstowy zawierający polskie imiona (‘IMIĘ_PIERWSZE.txt‘). Kod najpierw wczytał zawartość pliku, a następnie zidentyfikował wszystkie unikalne znaki występujące w imionach, dodając specjalny token ‘. reprezentujący początek i koniec imienia. Na tej podstawie stworzono mapowania znak-indeks (‘stoi‘) oraz indeks-znak (‘itos‘). Następnie zbudowano macierz częstości bigramów ‘N‘, zliczając wystąpienia każdej możliwej pary sąsiadujących znaków. Finalnie, macierz ‘N‘ przekształcono w macierz prawdopodobieństw przejść ‘P‘, stosując wygładzanie (+1) i normalizując wiersze.

Listing 2.1. Output: Wczytanie danych i budowa modelu bigramowego

```
Wczytywanie imion z pliku: /content/sample_data/IMI_PIERWSZE.txt
Wczytano 27457 imion.
Pierwsze 10 imion: ['ZUZANNA', 'MAJA', 'IGA', 'AMELIA', 'JAGODA', 'OLIWIA', 'MATYLDA',
     'LENA', 'MARIA', 'POLA']

Rozmiar slownika (liczba unikalnych znakow + '.'): 41
Mapowanie znak->indeks (stoi): {' ': 1, '-': 2, 'A': 3, 'B': 4, 'C': 5, 'D': 6, 'E':
    7, 'F': 8, 'G': 9, 'H': 10, 'I': 11, 'J': 12, 'K': 13, 'L': 14, 'M': 15, 'N': 16,
    'O': 17, 'P': 18, 'R': 19, 'S': 20, 'T': 21, 'U': 22, 'V': 23, 'W': 24, 'X': 25, '
    Y': 26, 'Z': 27, '\u00c0': 28, '\u00c1': 29, '\u00c9': 30, '\u00cb': 31, '\u00d0':
     32, '\u00d3': 33, '\u0118': 34, '\u0141': 35, '\u015a': 36, '\u017b': 37, '\u01a0
    ': 38, '\u01af': 39, '\u1ea0': 40, '.': 0}
Mapowanie indeks->znak (itos): {1: ' ', 2: '-', 3: 'A', 4: 'B', 5: 'C', 6: 'D', 7: 'E
    ', 8: 'F', 9: 'G', 10: 'H', 11: 'I', 12: 'J', 13: 'K', 14: 'L', 15: 'M', 16: 'N',
    17: 'O', 18: 'P', 19: 'R', 20: 'S', 21: 'T', 22: 'U', 23: 'V', 24: 'W', 25: 'X',
    26: 'Y', 27: 'Z', 28: '\u00c0', 29: '\u00c1', 30: '\u00c9', 31: '\u00cb', 32: '\
    u00d0', 33: '\u00d3', 34: '\u0118', 35: '\u0141', 36: '\u015a', 37: '\u017b', 38:
    '\u01a0', 39: '\u01af', 40: '\u1ea0', 0: '.'}

Utworzono macierz prawdopodobienstw P o ksztalcie: torch.Size([41, 41])
```

Uzyskano słownik składający się z 41 unikalnych znaków, w tym polskich liter diakrytyzowanych obecnych w pliku. Macierz ‘P‘ o wymiarach 41x41 zawierała wyuczone prawdopodobieństwa przejść między tymi znakami.

2.2. Funkcja Kosztu i Generacja Imion

Zdefiniowano dwie kluczowe funkcje: ‘calculate_loss‘ do obliczania średniego negatywnego logarytmu prawdopodobieństwa (average NLL) dla danego imienia na podstawie macierzy ‘P‘, oraz ‘generate_name‘ do generowania nowych imion poprzez próbkowanie kolejnych znaków zgodnie z rozkładami prawdopodobieństw w ‘P‘. Jako przykład obliczono koszt dla pierwszego imienia z wczytanej listy:

Listing 2.2. Output: Przykładowy koszt dla imienia ’ZUZANNA’

```
Przykladowy koszt dla imienia ’ZUZANNA’: 2.0925
```

Wartość ta (ok. 2.09) wskazuje na wysokie prawdopodobieństwo sekwencji znaków w imieniu "ZUZANNA"według modelu nauczonego na dostarczonym zbiorze.

Następnie przystąpiono do głównej części zadania: wygenerowania 50 imion, obliczenia ich kosztu i wyświetlenia 5 imion o najniższym (najbardziej prawdopodobnych wg modelu) i najwyższym (najmniej prawdopodobnych) koszcie.

Listing 2.3. Output: Generacja 50 imion i ocena wg kosztu

```
Uzyto seed: 573

Generowanie 50 imion...

Sortowanie wynikow...

--- 5 imion z NAJNIZSZYM kosztem (najbardziej ’podobne’ do danych) ---
’MIA’ (Koszt: 1.2312)
’MA’ (Koszt: 1.2975)
’JA’ (Koszt: 1.4019)
’JA’ (Koszt: 1.4019)
’JA’ (Koszt: 1.4019)

--- 5 imion z NAJWYZSZYM kosztem (najmniej ’podobne’ do danych) ---
’P’ (Koszt: 2.6737)
’H’ (Koszt: 2.7222)
’FLI’ (Koszt: 2.7494)
’ A ’ (Koszt: 2.7858)
’  ’ (Koszt: 3.1386)
```

Jak widać, imiona o najniższym koszcie są bardzo krótkie i składają się z popularnych sylab/sekwencji (’MA’, ’JA’, ’MIA’). Imiona o najwyższym koszcie zawierają rzadkie sekwencje lub znaki rzadko występujące na początku/końcu imion (np. ’Ł’). Wyniki te są zgodne z oczekiwaniami dla prostego modelu bigramowego opartego na częstościach.

3. Zadanie 2: Model Bigram - Generacja Imion Siecią Neuronową

Drugie zadanie laboratorium miało na celu zapoznanie się ze sposobem implementacji modelu bigramowego przy użyciu prostej sieci neuronowej w bibliotece PyTorch. Celem było stworzenie odpowiednika modelu opartego na zliczeniach (z Zadania 1), gdzie sieć uczy się rozkładu prawdopodobieństwa wystąpienia kolejnego znaku na podstawie znaku poprzedzającego. Wykorzystano ten sam zbiór polskich imion (‘IMIĘ_PIERWSZE.txt‘) oraz ten sam słownik znaków co w Zadaniu 1.

3.1. Przygotowanie Danych Treningowych

Pierwszym krokiem było przygotowanie danych treningowych w formacie odpowiednim dla sieci neuronowej. Na podstawie listy imion stworzono dwie listy indeksów: ‘xs‘ (indeksy znaków wejściowych) oraz ‘ys‘ (indeksy znaków docelowych, czyli następujących po znakach z ‘xs‘). Każdy bigram ‘(znak1, znak2)‘ ze zbioru danych został przekształcony w parę ‘(indeks_znak1, indeks_znak2)‘.

Następnie wejściowa lista indeksów ‘xs‘ została zakodowana w formacie one-hot przy użyciu funkcji ‘torch.nn.functional.one_hot‘. W tej reprezentacji każdy znak wejściowy jest wektorem o długości równej rozmiarowi słownika (41), zawierającym zera na wszystkich pozycjach z wyjątkiem jednej – odpowiadającej indeksowi danego znaku. Taki format jest standardowym wejściem dla prostych sieci przetwarzających dane kategorialne.

Listing 3.1. Output: Przygotowanie danych dla sieci neuronowej

```
Liczba przykladow treningowych (bigramw): 195216
Pierwsze 10 par indeksow (xs, ys):
  Input: 0 (.) -> Target: 27 (Z)
  Input: 27 (Z) -> Target: 22 (U)
  Input: 22 (U) -> Target: 27 (Z)
  Input: 27 (Z) -> Target: 3 (A)
  Input: 3 (A) -> Target: 16 (N)
  Input: 16 (N) -> Target: 16 (N)
  Input: 16 (N) -> Target: 3 (A)
  Input: 3 (A) -> Target: 0 (.)
  Input: 0 (.) -> Target: 15 (M)
  Input: 15 (M) -> Target: 3 (A)

Ksztalt zakodowanego wejscia (one-hot): torch.Size([195216, 41])
```

Przygotowano 195,216 par treningowych (bigramów). Macierz wejściowa ‘x_onehot‘ ma wymiary odpowiadające liczbie przykładów i rozmiarowi słownika.

3.2. Architektura Modelu i Forward Pass

W tym zadaniu model bigramowy został zaimplementowany jako pojedyncza warstwa liniowa, reprezentowana przez macierz wag ‘W‘ o wymiarach ‘[rozmiar_słownika, rozmiar_słownika]‘ (41x41). Macierz ‘W‘ została zainicjalizowana losowymi wartościami z rozkładu normalnego i ustawiono dla niej flagę ‘requires_grad=True‘, aby umożliwić obliczanie gradientów podczas treningu.

Listing 3.2. Output: Inicjalizacja macierzy wag

```
Zainicjalizowano macierz wag W o ksztalcie: torch.Size([41, 41])
```

Przejście w przód (forward pass) dla tej sieci polega na przemnożeniu macierzy wejściowej w formacie one-hot (‘x_onehot‘) przez macierz wag ‘W‘. Wynikiem tej operacji (‘logits = x_onehot @ W‘) jest macierz logitów, gdzie każdy wiersz odpowiada jednemu znakowi wejściowemu, a kolumny reprezentują "surowe wyniki"(przed normalizacją) dla każdego możliwego znaku następnego.

Aby przekształcić logity na rozkład prawdopodobieństwa, stosuje się funkcję Softmax. W kodzie zrealizowano to dwuetapowo: najpierw obliczono ‘counts = logits.exp()‘, a następnie znormalizowano ‘counts‘, dzieląc każdy wiersz przez sumę jego elementów (‘probs = counts / counts.sum(1, keepdims=True)‘). Wynikowa macierz ‘probs‘ zawiera prawdopodobieństwa wyboru każdego znaku jako następnego, przy czym suma prawdopodobieństw w każdym wierszu wynosi 1.

Listing 3.3. Output: Przykładowe prawdopodobieństwa (przed treningiem)

```
Przykladowe prawdopodobienstwa dla pierwszych 5 wejsc:
tensor([[0.0089, 0.0433, 0.0124, 0.0315, 0.0033, 0.0059, 0.0725, 0.0330, 0.0105,
         0.0250, 0.0082, 0.0325, 0.0077, 0.0645, 0.0209, 0.0411, 0.0123, 0.0018,
         0.0024, 0.0171, 0.0053, 0.2563, 0.0011, 0.0036, 0.0164, 0.0051, 0.0067,
         0.0013, 0.0184, 0.0047, 0.0026, 0.0124, 0.0446, 0.0143, 0.0035, 0.0532,
         0.0194, 0.0263, 0.0059, 0.0394, 0.0045],
        [0.0150, 0.0194, 0.0843, 0.0097, 0.0045, 0.0028, 0.0053, 0.0452, 0.0096,
         0.0127, 0.0127, 0.0487, 0.0131, 0.0046, 0.0628, 0.0030, 0.0111, 0.0373,
         0.0827, 0.0348, 0.0152, 0.0129, 0.0214, 0.0129, 0.0019, 0.0309, 0.0110,
         0.0295, 0.0045, 0.0253, 0.0071, 0.0100, 0.0250, 0.0200, 0.0087, 0.0125,
         0.0126, 0.0004, 0.0458, 0.1267, 0.0463],
        [0.0067, 0.0126, 0.0208, 0.0052, 0.0372, 0.0058, 0.0079, 0.0321, 0.0414,
         0.0120, 0.0191, 0.0483, 0.0217, 0.0208, 0.0133, 0.0142, 0.0618, 0.0489,
         0.0050, 0.0026, 0.0027, 0.0565, 0.0032, 0.0148, 0.0825, 0.0085, 0.0056,
         0.0039, 0.0668, 0.0513, 0.0055, 0.0178, 0.0312, 0.0119, 0.0583, 0.0035,
         0.0165, 0.0050, 0.0520, 0.0612, 0.0038],
        [0.0150, 0.0194, 0.0843, 0.0097, 0.0045, 0.0028, 0.0053, 0.0452, 0.0096,
         0.0127, 0.0127, 0.0487, 0.0131, 0.0046, 0.0628, 0.0030, 0.0111, 0.0373,
         0.0827, 0.0348, 0.0152, 0.0129, 0.0214, 0.0129, 0.0019, 0.0309, 0.0110,
         0.0295, 0.0045, 0.0253, 0.0071, 0.0100, 0.0250, 0.0200, 0.0087, 0.0125,
         0.0126, 0.0004, 0.0458, 0.1267, 0.0463],
        [0.0210, 0.0193, 0.0258, 0.0046, 0.0119, 0.0084, 0.0142, 0.0022, 0.0206,
```

```
         0.0981, 0.0387, 0.0044, 0.0116, 0.1275, 0.0063, 0.0026, 0.0314, 0.0095,
         0.0190, 0.0054, 0.0110, 0.0455, 0.0117, 0.0101, 0.0053, 0.0286, 0.0157,
         0.0099, 0.0123, 0.0398, 0.0153, 0.0163, 0.0091, 0.0156, 0.0051, 0.0406,
         0.0101, 0.0020, 0.1588, 0.0359, 0.0191]], grad_fn=<DivBackward0>)
Ksztalt macierzy prawdopodobienstw: torch.Size([5, 41])
Suma prawdopodobienstw w pierwszym wierszu: tensor(1.0000, grad_fn=<SumBackward0>)
```

Początkowe, losowe wagi ‘W‘ prowadzą do rozkładów prawdopodobieństw, które nie odzwierciedlają jeszcze rzeczywistych zależności w danych.

3.3. Trening Modelu Neuronowego

Proces treningu polegał na iteracyjnym dostosowywaniu macierzy wag ‘W‘ w celu minimalizacji funkcji kosztu. Jako funkcję kosztu wybrano średni negatywny logarytm prawdopodobieństwa (NLL) dla poprawnych następnych znaków, z dodanym członem regularyzacji L2, który zapobiega nadmiernemu wzrostowi wartości wag. Trening przeprowadzono przez 100 epok, wykorzystując metodę spadku gradientu z relatywnie dużym krokiem uczenia (‘learning_rate = 50‘).

Listing 3.4. Obliczenie funkcji kosztu i aktualizacja wag (fragment pętli treningowej)

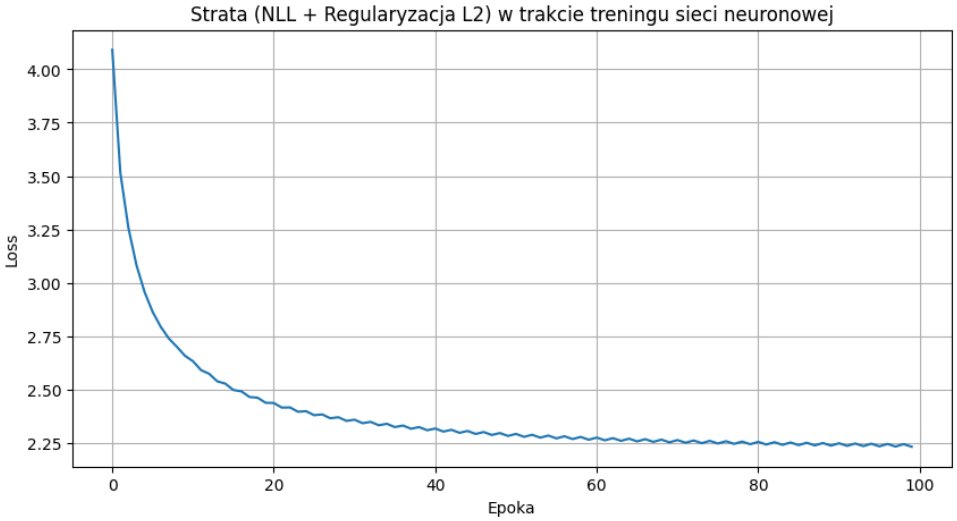
```
  loss = -probs[torch.arange(num_examples), ys].log().mean() + regularization_strength
     * (W**2).mean()
  losses.append(loss.item())
  W.grad = None
  loss.backward()
  W.data += -learning_rate * W.grad
```

Postęp treningu monitorowano, zapisując wartość funkcji kosztu po każdej epoce i wyświetlając ją co 10 epok:

Listing 3.5. Output: Postęp treningu sieci neuronowej

```
Rozpoczynanie treningu...
Epoka 10/100, Strata: 2.6586
Epoka 20/100, Strata: 2.4385
Epoka 30/100, Strata: 2.3538
Epoka 40/100, Strata: 2.3100
Epoka 50/100, Strata: 2.2833
Epoka 60/100, Strata: 2.2656
Epoka 70/100, Strata: 2.2534
Epoka 80/100, Strata: 2.2445
Epoka 90/100, Strata: 2.2379
Epoka 100/100, Strata: 2.2328
Trening zakonczony.
```

Przebieg zmian funkcji kosztu podczas treningu został również zwizualizowany na wykresie (Rysunek 3.1).



Rys. 3.1: Wykres wartości funkcji kosztu (NLL + Regularyzacja L2) w trakcie treningu sieci neuronowej dla modelu bigramowego.

Zarówno logi, jak i wykres pokazują systematyczny spadek wartości funkcji kosztu podczas 100 epok treningu, co świadczy o tym, że sieć neuronowa uczyła się zależności bigramowych obecnych w danych. Wartość straty ustabilizowała się na poziomie około 2.23, co jest wartością lepszą niż losowe przewidywanie (ok. 3.29), ale nieco gorszą niż w przypadku modelu opartego na zliczeniach z wygładzaniem (gdzie koszt dla 'ZUZANNA' wynosił ok. 2.09, a średni koszt dla danych prawdopodobnie byłby zbliżony).

3.4. Generowanie Imion z Modelu Neuronowego

Po zakończeniu treningu przetestowano zdolność modelu do generowania nowych imion poprzez iteracyjne próbkowanie kolejnych znaków na podstawie rozkładu prawdopodobieństwa wynikającego z macierzy wag 'W'.

Listing 3.6. Output: Imiona wygenerowane przez wytrenowaną sieć neuronową

```
Generowanie 10 imion z wytrenowanej sieci neuronowej:
IAMIAROSYYLEKANAWIA
MA
AVIZJAM
L
ALELEKJUSZLIARIAN
 HAGANAROG
MA
MIANTOFRILELANIALARZYKAPOLHANANIWIUELIGANTAJEN
 FESANDERB
O
```

Wygenerowane imiona, podobnie jak w przypadku modelu opartego na zliczeniach, wykazują pewne cechy języka źródłowego, ale często zawierają nietypowe sekwencje lub są bardzo krótkie. Wyniki te ilustrują, że nawet prosta sieć neuronowa może nauczyć się modelować zależności bigramowe, choć jakość generacji zależy od architektury, hiperparametrów i długości treningu.

4. Zadanie 4: Analiza Dialogów "The Simpsons"z Wykorzystaniem Gensim Word2Vec

4.1. Wstęp i Cel Analizy

W czwartym zadaniu przeprowadzono analizę semantyczną transkrypcji dialogów z serialu animowanego "The Simpsons", wykorzystując model Word2Vec zaimplementowany w bibliotece Gensim. Celem było zrozumienie działania tej techniki oraz ocena jej zdolności do wychwytywania relacji między słowami (w tym postaciami) na podstawie dużego korpusu tekstowego.

4.2. Wczytywanie i Wstępne Przetwarzanie Danych

Proces rozpoczął się od pobrania archiwum ‘simpsons_script_lines.csv.zip‘ i jego rozpakowania. Wczytano dane przy użyciu biblioteki ‘pandas‘, koncentrując się na kolumnie ‘normalized_text‘. Po usunięciu brakujących wpisów uzyskano zbiór danych o następującym rozmiarze:

Listing 4.1. Output: Rozmiar danych po wczytaniu

```
Liczba dialogow po wczytaniu i usunieciu NaN: 132087
```

4.3. Zaawansowany Preprocessing Tekstu

Kolejnym krokiem był zaawansowany preprocessing tekstu, obejmujący usuwanie niepożądanych znaków, lematyzację i usunięcie słów stopu za pomocą ‘spacy‘. Proces ten trwał określony czas i zredukował liczbę unikalnych dialogów:

Listing 4.2. Output: Wyniki czyszczenia tekstu

```
Czas czyszczenia: 2.2 min
Liczba unikalnych, niepustych dialogow po czyszczeniu: 105650
```

Efektywność czyszczenia została zilustrowana na przykładzie:

Listing 4.3. Output: Przykład czyszczenia tekstu

```
Przyklad - po brief_cleaning: 'no actually it was a little of both sometimes'
Przyklad - po pelnym cleaning: 'actually little'
```

4.4. Tokenizacja i Tworzenie Bigramów

Przygotowane dane poddano tokenizacji. Zastosowano również mechanizm tworzenia bigramów (‘gensim.models.phrases‘), co pozwoliło połączyć często współwystępujące słowa:

Listing 4.4. Output: Przykład działania bigramów

```
Przyklad zdania po dodaniu bigramow (jesli jakies powstaly):
Oryginalne (tokenizowane): ['think', 's', 'take', 'train', 'capital', 'city']
Z bigramami: ['think', 's', 'take', 'train', 'capital_city']
```

4.5. Trenowanie Modelu Word2Vec

Następnie przystąpiono do treningu modelu ‘gensim.models.Word2Vec‘ z ustalonymi parametrami (m.in. ‘vector_size=300‘, ‘window=2‘, ‘min_count=20‘). Informacje o budowie słownika i czasie treningu przedstawiono poniżej:

Listing 4.5. Output: Budowa słownika i czas treningu

```
Liczba dostepnych rdzeni CPU: 2
Czas budowania slownika: 0.01 min
Rozmiar slownika (liczba unikalnych tokenow): 3389
Czas trenowania modelu: 1.24 min
```

4.6. Ewaluacja Semantyczna Modelu

Po wytrenowaniu modelu przeprowadzono jego ewaluację semantyczną, wykorzystując metodę ‘most_similar‘. Wyniki dla głównych bohaterów pokazują interesujące zależności semantyczne wyuczone przez model:

Listing 4.6. Output: Wyniki most_similar dla 'homer'

```
Slowa najbardziej podobne do 'homer':
[('marge', 0.7032139301300049), ('depressed', 0.6795114278793335), ('upside',
    0.6730273962020874), ('mindy', 0.6652554869651794), ('reverend_lovejoy',
    0.6531610488891602), ('dr_hibbert', 0.6517014503479004), ('humiliate',
    0.6505870819091797), ('becky', 0.6470052003860474), ('snuggle',
    0.6467839479446411), ('bongo', 0.6447693109512329)]
```

Listing 4.7. Output: Wyniki most_similar dla 'marge'

```
Slowa najbardziej podobne do 'marge':
[('homer', 0.7032139897346497), ('abe', 0.6526325941085815), ('envy',
    0.6521787643432617), ('humiliate', 0.6515457630157471), ('attract',
    0.6468246579170227), ('rapture', 0.6434506177902222), ('husband',
    0.640910267829895), ('depressed', 0.6390814781188965), ('mindy',
    0.6296670436859131), ('mmmhmm', 0.6285890340805054)]
```

Listing 4.8. Output: Wyniki most_similar dla 'bart'

```
Slowa najbardziej podobne do 'bart':
[('lisa', 0.7766444683074951), ('pay_attention', 0.6911911964416504), ('dr_hibbert',
    0.6909650564193726), ('mom_dad', 0.6876555681228638), ('convince',
    0.6845778226852417), ('embarrassing', 0.6840355396270752), ('videogame',
    0.6822503209114075), ('behave', 0.6805838346481323), ('miss_hoover',
    0.6733542680740356), ('dad', 0.6709942817687988)]
```

Analiza za pomocą metody ‘doesnt_match‘ potwierdziła zdolność modelu do rozumienia kontekstu społecznego:

Listing 4.9. Output: Wynik doesnt_match

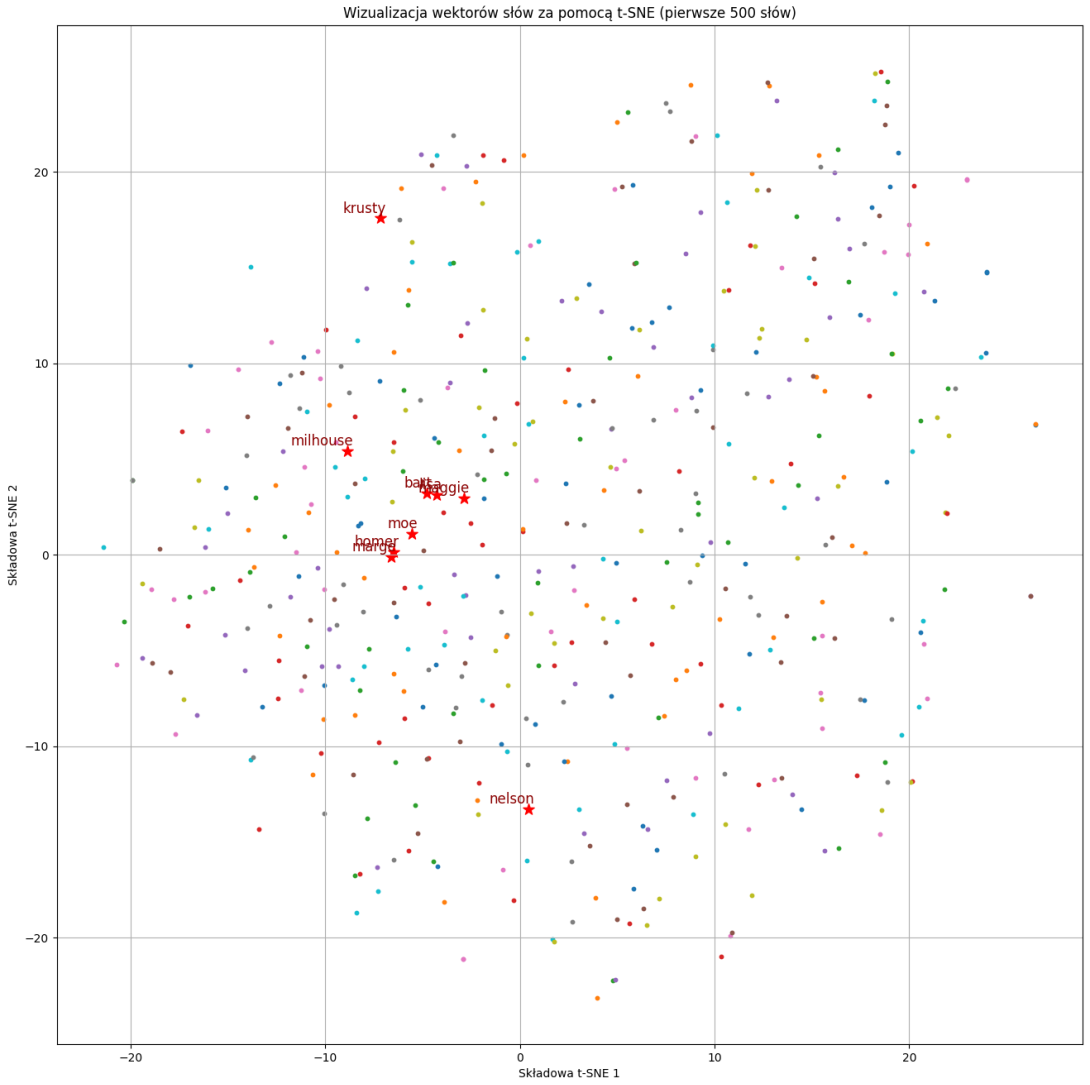
```
Sprawdzanie, ktore slowo nie pasuje do listy: ['jimbo', 'milhouse', 'kearney']
Wedlug modelu, sowo  ktre  najmniej pasuje to: 'milhouse'
```

4.7. Podsumowanie Wyników Semantycznych

- Dla 'homer': najbliższe okazały się 'marge', ale także terminy związane z emocjami ('depressed') i inne postacie ('mindy', 'reverend_lovejoy').
- Dla 'marge': najbliższy był 'homer', a także 'abe' (Abraham Simpson), 'husband' oraz słowa nacechowane emocjonalnie ('envy', 'humiliate').
- Dla 'bart': najwyższe podobieństwo wykazywała 'lisa', co odzwierciedla ich relację rodzeństwa, ale blisko znalazły się też 'dad', 'mom_dad' oraz inne postacie i terminy szkolne/behawioralne.

4.8. Wizualizacja Wektorów Słów

Ostatnim etapem była wizualizacja nauczonych reprezentacji wektorowych. Wykorzystano algorytm t-SNE do redukcji wymiarowości wektorów dla 500 najczęstszych słów do przestrzeni 2D. Wynikowa wizualizacja (Rysunek 4.1) przedstawia rozkład tych słów, z podświetlonymi na czerwono głównymi postaciami znalezionymi w modelu. Chociaż postacie są rozproszone, można zaobserwować pewne bliskości, np. między Bartem a Lisą.



Rys. 4.1. Wizualizacja t-SNE wektorów słów (top 500) dla dialogów z "The Simpsons", z podświetlonymi wybranymi postaciami.

5. Zadanie 5: Implementacja CBOW w PyTorch - Przygotowanie Danych

Implementacja modelu CBOW (Continuous Bag of Words) w bibliotece PyTorch rozpoczęła się od przygotowania danych treningowych. Jako podstawę wykorzystano krótki korpus tekstowy w języku angielskim.

Pierwszym krokiem w zadaniu 5 było wczytanie niezbędnych bibliotek, w tym ‘torch‘, ‘numpy‘ oraz narzędzi z ‘sklearn‘. Następnie zdefiniowany korpus został poddany podstawowemu preprocessingowi: konwersji na małe litery i tokenizacji (podziałowi na słowa). Na podstawie przetworzonego tekstu zbudowano słownik unikalnych tokenów oraz mapowania pozwalające na konwersję między słowem a jego unikalnym indeksem liczbowym (‘word_to_ix‘ i ‘ix_to_word‘). Wyniki tej operacji przedstawiono poniżej:

Listing 5.1. Output: Rozmiar i próbka słownika

```
Rozmiar slownika: 41
Probka slownika (word_to_ix): [('word', 0), ('embeddings', 1), ('are', 2), ('a', 3),
    ('type', 4), ('of', 5), ('representation', 6), ('that', 7), ('allows', 8), ('words
    ', 9)]
```

Kluczowym etapem było wygenerowanie danych w formacie odpowiednim dla modelu CBOW. Dla każdego słowa w korpusie (z wyjątkiem słów na krawędziach, dla których nie można było utworzyć pełnego kontekstu), jako kontekst przyjęto ‘context_size=2‘ słowa występujące bezpośrednio przed nim oraz 2 słowa występujące bezpośrednio po nim. Słowo centralne stanowiło cel (target), który model miał przewidywać na podstawie kontekstu. W ten sposób utworzono zbiór par treningowych (lista indeksów słów kontekstowych, indeks słowa docelowego).

Listing 5.2. Output: Wyniki generowania danych treningowych CBOW

```
Liczba probek treningowych: 59
Przykladowa probka (indeksy kontekstu, indeks celu): ([1, 0, 3, 4], 2)
Odpowiadajace slowa: (['embeddings', 'word', 'a', 'type'], 'are')
```

Jak widać na przykładzie, dla słowa ’are’ (indeks 2), model będzie uczył się przewidywać je na podstawie kontekstu składającego się ze słów ’embeddings’, ’word’, ’a’ oraz ’type’ (indeksy 1, 0, 3, 4). Tak przygotowany zbiór danych został następnie wykorzystany do treningu sieci neuronowej CBOW.

5.1. Definicja Modelu CBOW w PyTorch

Po przygotowaniu danych treningowych zdefiniowano architekturę sieci neuronowej CBOW przy użyciu biblioteki PyTorch. Stworzono klasę ‘CBOW‘ dziedziczącą po ‘torch.nn.Module‘, która enkapsuluje logikę modelu.

Kluczowe elementy architektury to:

- Warstwa ‘torch.nn.Embedding‘, odpowiedzialna za przechowywanie i naukę wektorowych reprezentacji słów (embeddingów). Jej rozmiar został ustalony na podstawie rozmiaru słownika (‘vocab_size=41‘) oraz zdefiniowanego wymiaru embeddingu (‘EMBEDDING_DIM=10‘).
- Warstwa ‘torch.nn.Linear‘, która przetwarza zagregowany wektor kontekstu (uzyskany przez zsumowanie embeddingów słów kontekstowych w metodzie ‘forward‘) na wektor logitów o rozmiarze słownika.
- Metoda ‘forward‘, która realizuje przepływ danych przez model, włącznie z agregacją kontekstu i zastosowaniem funkcji ‘log_softmax‘ na wyjściu w celu uzyskania logarytmów prawdopodobieństw słów docelowych.

Poniżej przedstawiono kod definicji klasy modelu:

Listing 5.3. Definicja klasy modelu CBOW (PyTorch)

```
class CBOW(nn.Module):
    def __init__(self, vocab_size, embedding_dim, context_size):
        super(CBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim, vocab_size)
    def forward(self, inputs):
        embeds = self.embeddings(inputs)
        summed_embeds = torch.sum(embeds, dim=1)
        out = self.linear1(summed_embeds)
        log_probs = nn.functional.log_softmax(out, dim=1)
        return log_probs
    def get_embeddings(self):
      return self.embeddings.weight.data.cpu().numpy()
```

Następnie zainicjalizowano model, podając rozmiar słownika (41) i wymiar embeddingu (10). Jako funkcję kosztu wybrano ‘nn.NLLLoss‘, a jako optymalizator ‘optim.Adam‘ z szybkością uczenia (learning rate) 0.01. Zdefiniowano również liczbę epok treningowych na 50. Struktura zainicjalizowanego modelu została wyświetlona w konsoli:

Listing 5.4. Output: Struktura zainicjalizowanego modelu CBOW

```
Model CBOW zdefiniowany:
CBOW(
  (embeddings): Embedding(41, 10)
  (linear1): Linear(in_features=10, out_features=41, bias=True)
)
```

Output potwierdza poprawną konfigurację warstw modelu, gotowego do rozpoczęcia procesu treningu.

5.2. Trening Modelu CBOW

Po zdefiniowaniu architektury modelu CBOW oraz przygotowaniu danych, przystąpiono do procesu treningu. Wykorzystano standardową pętlę treningową w PyTorch, iterującą przez zadane 50 epok. W każdej epoce przetwarzano kolejno wszystkie 59 próbek treningowych.

Dla każdej próbki wykonywano następujące kroki:

1) Przygotowanie danych wejściowych (indeksy słów kontekstowych) i docelowych (indeks słowa centralnego) jako tensorów PyTorch.

2) Wyzerowanie gradientów modelu.

3) Wykonanie przejścia w przód (forward pass) w celu uzyskania logarytmów prawdopodobieństw dla słowa docelowego.

4) Obliczenie wartości funkcji kosztu (‘nn.NLLLoss‘) między predykcją modelu a rzeczywistym słowem docelowym.

5) Wykonanie przejścia wstecz (backward pass) w celu obliczenia gradientów funkcji kosztu względem parametrów modelu.

6) Aktualizacja parametrów (wag) modelu przy użyciu optymalizatora Adam (‘optimizer.step()‘).

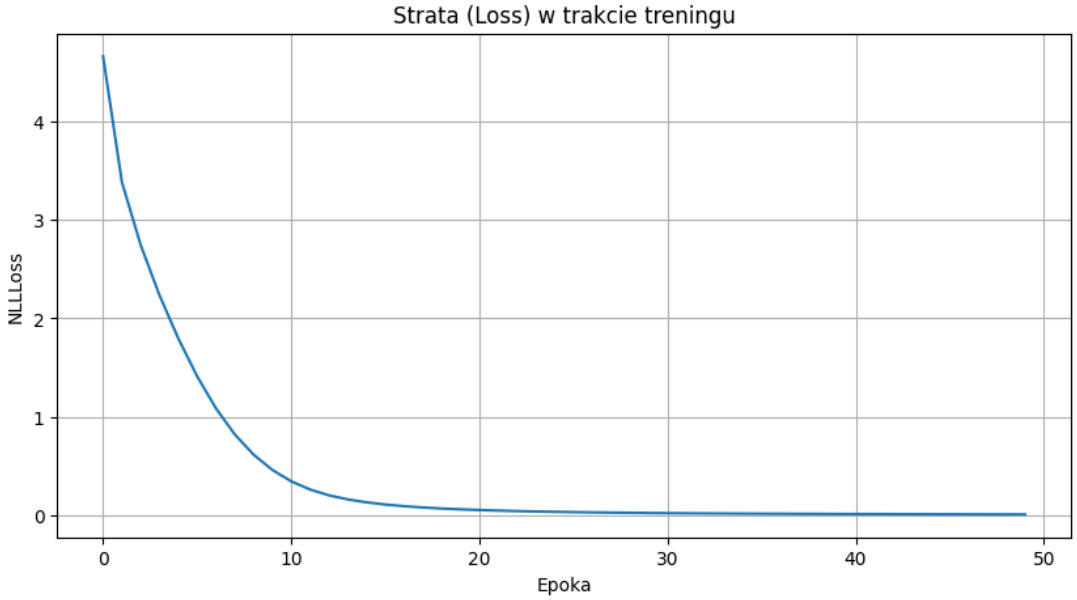
Średnia wartość funkcji kosztu była obliczana i zapisywana po każdej epoce. Postęp treningu monitorowano, wyświetlając średnią stratę co 10 epok:

Listing 5.5. Output: Postęp treningu modelu CBOW

```
Trenowanie na urzadzeniu: cpu
Epoka: 10/50, Srednia strata: 0.4573
Epoka: 20/50, Srednia strata: 0.0566
Epoka: 30/50, Srednia strata: 0.0198
Epoka: 40/50, Srednia strata: 0.0096
```

```
Epoka: 50/50, Srednia strata: 0.0054

Trening zakonczony.
```

Jak widać w logach, strata systematycznie malała, co wskazuje na skuteczny proces uczenia. Zostało to również zwizualizowane na wykresie funkcji kosztu (Rysunek 5.1).



Rys. 5.1. Wykres wartości funkcji kosztu (NLLLoss) w trakcie treningu modelu CBOW.

Wykres pokazuje gwałtowny spadek straty w początkowych epokach i jej stabilizację na niskim poziomie w dalszej części treningu.

Po zakończeniu treningu, z modelu wyekstrahowano nauczone reprezentacje wektorowe słów (embeddingi). Kształt uzyskanej macierzy embeddingów był zgodny z oczekiwaniami:

Listing 5.6. Output: Kształt macierzy embeddingów

```
Ksztalt macierzy embeddingow: (41, 10)
```

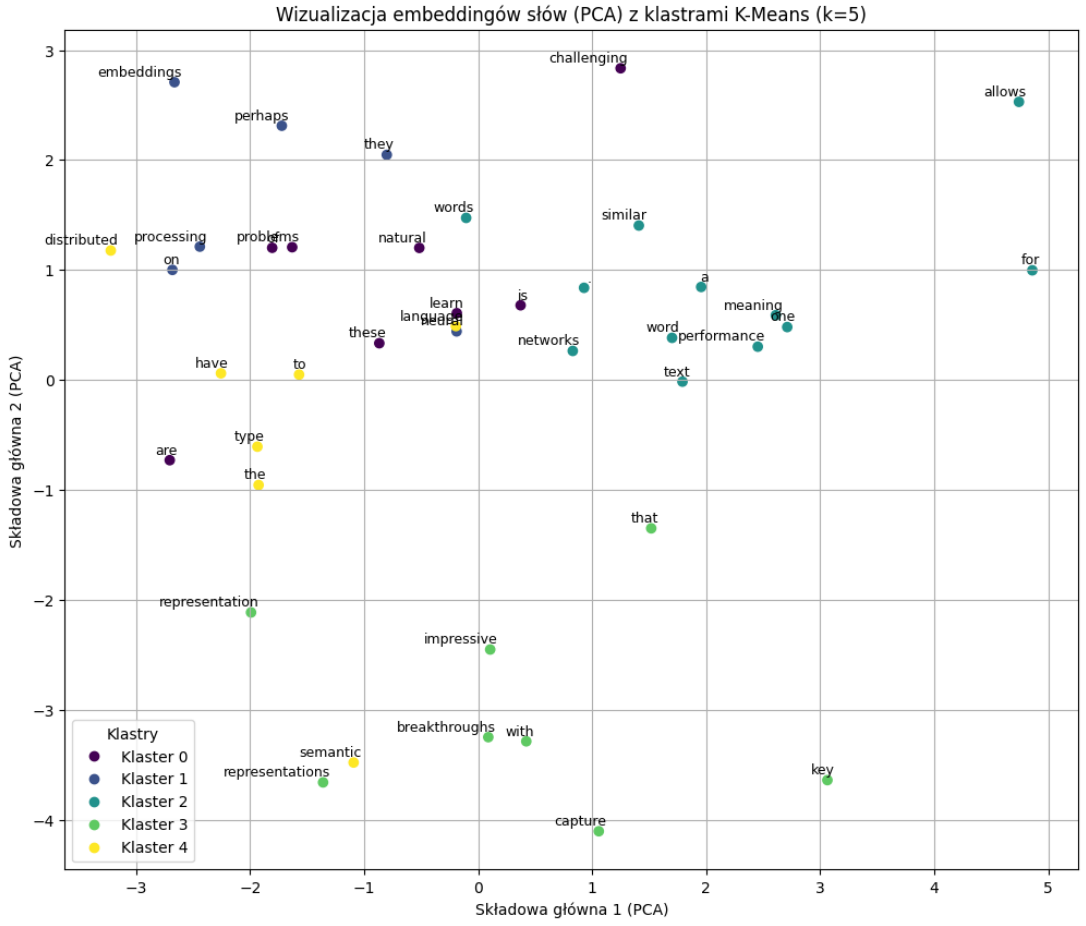
Potwierdza to uzyskanie 10-wymiarowych wektorów dla każdego z 41 słów w słowniku. Embeddingi te zostały następnie wykorzystane do klasteryzacji i wizualizacji.

5.3. Klasteryzacja i Wizualizacja Embeddingów

Ostatnim krokiem w ramach Zadania 5 była analiza i wizualizacja wektorów słów (embeddingów) nauczonych przez model CBOW. W tym celu wykorzystano algorytm grupowania K-Means oraz technikę redukcji wymiarowości PCA.

Najpierw zastosowano algorytm K-Means z biblioteki ‘scikit-learn‘, aby podzielić 10-wymiarowe wektory embeddingów na ‘n_clusters = 5‘ grup. Każdemu słowu została przypisana etykieta klastra na podstawie jego położenia w przestrzeni embeddingów.

Następnie, w celu umożliwienia wizualizacji, wymiarowość wektorów została zredukowana z 10 do 2 za pomocą Analizy Głównych Składowych (PCA). Wynikowa, dwuwymiarowa reprezentacja słów została przedstawiona na wykresie punktowym (Rysunek 5.2). Na wykresie każdy punkt odpowiada jednemu słowu ze słownika, a jego kolor wskazuje na przynależność do jednego z pięciu klastrów znalezionych przez K-Means. Punkty zostały również opisane etykietami słownymi.



Rys. 5.2. Wizualizacja (PCA) wektorów słów z modelu CBOW z zaznaczonymi klastrami K-Means (k=5).

Wykres wizualizuje przestrzenne rozmieszczenie słów w zredukowanej przestrzeni, pokazując tendencję do grupowania się słów z tych samych klastrów.

Dodatkowo, przeanalizowano zawartość poszczególnych klastrów, wypisując należące do nich słowa:

Listing 5.7. Output: Słowa pogrupowane w klastry K-Means (k=5)

```
Slowa pogrupowane w klastry:
Klaster 0: are, of, is, challenging, natural, problems, learn, these
Klaster 1: embeddings, they, perhaps, neural, on, processing
Klaster 2: word, a, allows, words, similar, meaning, ., for, text, one, performance,
    networks
Klaster 3: representation, that, with, key, breakthroughs, impressive, capture,
    representations
Klaster 4: type, to, have, distributed, the, language, semantic
```

Analiza zawartości klastrów (np. Klaster 1 zawierający terminy 'embeddings', 'neural', 'processing'; Klaster 2 grupujący słowa 'word', 'words', 'text', 'meaning') sugeruje, że model CBOW, mimo treningu na bardzo małym zbiorze danych, był w stanie nauczyć się pewnych spójnych zależności semantycznych i kontekstowych, co znalazło odzwierciedlenie w strukturze przestrzeni embeddingów ujawnionej przez klasteryzację i wizualizację.

6. Zadanie 6: Analiza 4-gramów (Gensim Word2Vec)

Zadanie 6 polegało na zastosowaniu metody Word2Vec (przy użyciu biblioteki Gensim) do analizy sekwencji 4-gramów. Jako dane wejściowe wykorzystano plik ‘raw_sentences.txt‘.

6.1. Pobranie i Wstępna Inspekcja Danych

Pierwszym krokiem było pobranie wymaganego pliku danych. Kod sprawdzał, czy plik ‘raw_sentences.txt‘ już istnieje lokalnie, a jeśli nie, pobierał go z podanego adresu URL za pomocą narzędzia ‘wget‘. Logi wykonania potwierdzają pomyślne pobranie pliku:

Listing 6.1. Output: Log pobierania pliku raw_sentences.txt

```
Pobieranie pliku raw_sentences.txt...
Pobieranie zakonczone.
```

Po pobraniu pliku wyświetlono jego pierwsze pięć linii za pomocą polecenia ‘head‘, aby zapoznać się z formatem danych:

Listing 6.2. Output: Pierwsze 5 linii pliku raw_sentences.txt

```
Pierwsze 5 linii pliku:
 No ,  he says now .
And what did he do ?
 The money 's there .
That was less than a year ago .
But he made only the first .
```

Podgląd potwierdził, że plik zawiera tekst podzielony na linie, który będzie dalej przetwarzany w celu ekstrakcji 4-gramów lub analizy jako sekwencje słów.

6.2. Przetwarzanie Danych 4-gramowych

Po pobraniu pliku ‘raw_sentences.txt‘, kolejnym krokiem było jego przetworzenie do struktury danych odpowiedniej dla modelu Word2Vec. Kod wczytał plik linia po linii, a następnie każdą niepustą linię podzielił na listę słów (tokenów), usuwając jednocześnie wiodące i końcowe białe znaki. Wszystkie te listy słów zostały zgromadzone w jednej głównej liście, tworząc format "listy list".

Proces ten objął cały plik, a jego wynik podsumowano w konsoli:

Listing 6.3. Output: Wyniki przetwarzania pliku raw_sentences.txt

```
Liczba wczytanych 4-gramw (zdan): 97162

```

```
Przykadowe przetworzone 4-gramy (lista list):
['No', ',', 'he', 'says', 'now', '.']
['And', 'what', 'did', 'he', 'do', '?']
['The', 'money', "'s", 'there', '.']
['That', 'was', 'less', 'than', 'a', 'year', 'ago', '.']
['But', 'he', 'made', 'only', 'the', 'first', '.']

Rozmiar slownika (liczba unikalnych slow): 652
```

Jak widać, uzyskano ponad 97 tysięcy sekwencji słów. Przykładowe sekwencje zawierają zarówno słowa, jak i znaki interpunkcyjne. Całkowita liczba unikalnych tokenów w zbiorze wyniosła 652, co jest informacją istotną dla budowy słownika modelu Word2Vec w kolejnym etapie. Przygotowana lista list (‘word_list‘) posłużyła jako dane wejściowe do treningu modelu.

6.3. Trening Modelu Word2Vec (Gensim)

Po przygotowaniu danych w formacie listy list sekwencji (‘word_list‘), przystąpiono do treningu modelu Word2Vec przy użyciu biblioteki Gensim. Wybrano architekturę CBOW (‘sg=0‘). Ze względu na specyfikę danych (krótkie sekwencje, mały słownik) oraz charakterystykę zadania analizy 4-gramów, zdefiniowano następujące parametry modelu:

- Wymiar wektora embeddingu (‘vector_size‘): 50
- Rozmiar okna kontekstowego (‘window‘): 2
- Minimalna liczba wystąpień słowa (‘min_count‘): 1 (aby uwzględnić wszystkie 652 tokeny)

Model został zainicjalizowany, przekazując listę sekwencji ‘word_list‘. Na tym etapie automatycznie zbudowano słownik modelu. Następnie model został wytrenowany przez 20 epok. Informacje o rozmiarze słownika i czasie treningu zostały zarejestrowane:

Listing 6.4. Output: Informacje o treningu modelu Word2Vec dla 4-gramów

```
Liczba dostepnych rdzeni CPU: 2

Inicjalizowanie modelu Word2Vec...
[...]
Budowanie slownika zakonczone.
Rozmiar slownika modelu: 652

Rozpoczynanie treningu modelu...
Trening zakonczony w 24.08 sek.

Model Word2Vec dla 4-gramw jest gotowy.
```

Trening zakończył się pomyślnie w czasie około 24 sekund. Uzyskany model ‘w2v_model_4gram‘ zawiera nauczone 50-wymiarowe reprezentacje wektorowe dla 652 unikalnych tokenów (słów i znaków interpunkcyjnych) występujących w analizowanym zbiorze danych i jest gotowy do dalszej analizy semantycznej.

6.4. Analiza Podobieństwa Słów

Po wytrenowaniu modelu ‘w2v_model_4gram‘ przeprowadzono analizę nauczonych reprezentacji wektorowych, wykorzystując metodę ‘most_similar‘ z biblioteki Gensim. Celem było sprawdzenie, czy model uchwycił semantyczne lub gramatyczne podobieństwa między tokenami.

W pierwszej kolejności wyświetlono próbkę słownika modelu, aby zorientować się w jego zawartości. Następnie wybrano listę słów testowych, w tym popularne zaimki, czasowniki, określniki, przyimki oraz znaki interpunkcyjne. Dla każdego z tych słów wyszukano 5 najbardziej podobnych tokenów w wyuczonej przestrzeni embeddingów. Wyniki tej analizy przedstawiono poniżej:

Listing 6.5. Output: Próbka słownika i wyniki most_similar dla wybranych słów

```
Probka slw w slowniku modelu: ['.', ',', 'I', 'it', 'to', 'do', 'nt', '?', "'s", 'the
    ', 'is', 'that', 'he', 'said', 'not', 'It', 'you', 'But', 'know', 'was']

Slowa najbardziej podobne do 'he':
  - she (podobienstwo: 0.966)
  - She (podobienstwo: 0.594)
  - He (podobienstwo: 0.589)
  - though (podobienstwo: 0.559)
  - Still (podobienstwo: 0.548)

Slowa najbardziej podobne do 'was':
  - is (podobienstwo: 0.791)
  - 's (podobienstwo: 0.698)
  - Is (podobienstwo: 0.657)
  - Was (podobienstwo: 0.536)
  - were (podobienstwo: 0.455)

Slowa najbardziej podobne do 'the':
  - our (podobienstwo: 0.762)
  - The (podobienstwo: 0.710)
  - their (podobienstwo: 0.696)
  - my (podobienstwo: 0.686)
  - his (podobienstwo: 0.683)

 Sowa najbardziej podobne do 'in':
  - into (podobiestwo: 0.632)
  - for (podobiestwo: 0.594)
  - at (podobiestwo: 0.557)
  - In (podobiestwo: 0.549)
  - on (podobiestwo: 0.541)

Slowa najbardziej podobne do '.':
  - ; (podobienstwo: 0.721)
```

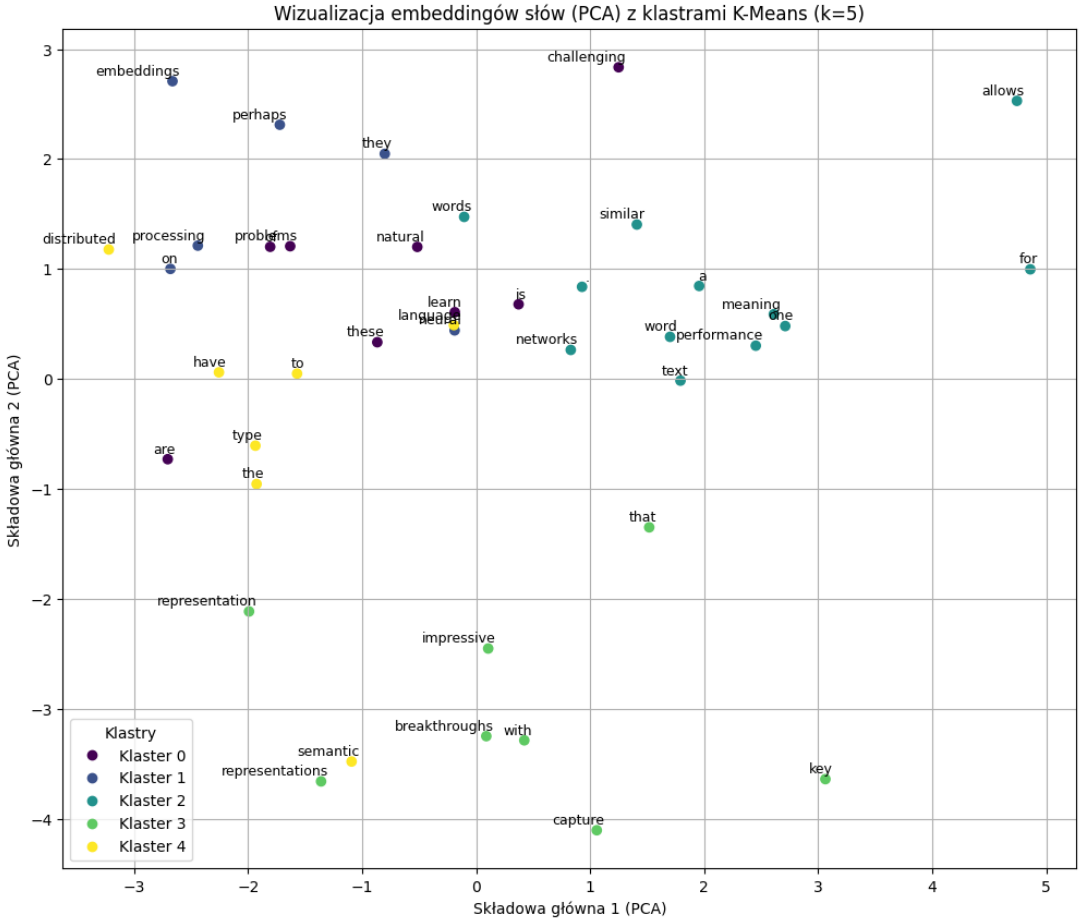
```
  - ? (podobienstwo: 0.707)
  - and (podobienstwo: 0.623)
  - , (podobienstwo: 0.614)
  - for (podobienstwo: 0.569)

Slowa najbardziej podobne do '?':
  - . (podobienstwo: 0.707)
  - ; (podobienstwo: 0.603)
  - and (podobienstwo: 0.601)
  - with (podobienstwo: 0.550)
  - When (podobienstwo: 0.499)
```

Obserwacje z powyższych wyników wskazują na poprawne działanie modelu. Widoczne jest grupowanie słów o podobnej funkcji gramatycznej: zaimki ('he', 'she'), formy czasownika 'być' ('was', 'is', 'were'), określniki ('the', 'our', 'my', 'his'), przyimki ('in', 'into', 'on', 'at'). Model rozróżnia wielkość liter (np. 'he' vs 'He'). Znaki interpunkcyjne ('.', '?') również wykazują podobieństwo do siebie oraz do często występujących w ich kontekście spójników ('and'). Potwierdza to, że model Word2Vec był w stanie nauczyć się zależności strukturalnych i kontekstowych na podstawie dostarczonych sekwencji 4-gramowych.

6.5. Wizualizacja Embeddingów 4-gramowych (PCA)

W celu wizualnej inspekcji relacji wyuczonych przez model Word2Vec na danych 4-gramowych, przeprowadzono redukcję wymiarowości uzyskanych 50-wymiarowych wektorów embeddingów do przestrzeni 2D. Wykorzystano do tego Analizę Głównych Składowych (PCA) zaimplementowaną w bibliotece 'scikit-learn'.

Kod pobrał macierz embeddingów oraz listę odpowiadających im tokenów bezpośrednio z wytrenowanego modelu 'w2v_model_4gram'. Następnie zastosowano PCA w celu transformacji danych do dwóch głównych składowych. Wynikowa reprezentacja 2D została wykorzystana do stworzenia wykresu punktowego (Rysunek 6.1), gdzie każdy punkt odpowiada jednemu tokenowi ze słownika, a jego pozycja na wykresie odzwierciedla jego położenie w zredukowanej przestrzeni embeddingów. Każdy punkt został dodatkowo opisany etykietą tekstową (tokenem).



Rys. 6.1: Wizualizacja (PCA) wektorów słów (embeddingów) nauczonych przez model Word2Vec (CBOW) na danych 4-gramowych.

Wygenerowany wykres przedstawia wszystkie 652 tokeny w przestrzeni 2D. Obserwuje się dużą gęstość punktów, szczególnie w centralnej części wykresu, co jest oczekiwane przy wizualizacji całego słownika. Mimo to, wizualizacja pozwala na ogólną ocenę struktury przestrzeni embeddingów. Analizując położenie poszczególnych etykiet, można próbować doszukiwać się grupowania słów o podobnych funkcjach gramatycznych (np. zaimków, przyimków) lub występujących w podobnych kontekstach w ramach 4-gramów, co mogłoby potwierdzać wnioski z analizy metodą ‘most_similar‘.

7. Zadanie 7: Analiza Implementacji Word2Vec "From Scrath"

Zadanie 7 polegało na analizie dostarczonego kodu implementacji algorytmu Word2Vec from scratch (zaimplementowanego w Pythonie z użyciem głównie biblioteki NumPy, bez wykorzystania gotowych bibliotek jak Gensim) oraz odpowiedzi na pytania dotyczące jego działania. Analiza została przeprowadzona na podstawie kodu udostępnionego w formie notatnika Colab.

7.1. Odpowiedzi na Pytania

Na podstawie analizy kodu uzyskano następujące odpowiedzi:

1) **Pytanie 1: Jaką miarę odległości/podobieństwa zastosowano w przykładzie word2vec?**

   Odpowiedź: W analizowanym przykładzie zastosowano **podobieństwo kosinusowe (Cosine Similarity)** do mierzenia bliskości (podobieństwa) między wektorami słów. Kod w metodach ‘vec_sim‘ oraz ‘word_sim‘ bezpośrednio implementuje wzór na podobieństwo kosinusowe, obliczając iloczyn skalarny wektorów i dzieląc go przez iloczyn ich norm (długości):

$$\text{Podobieństwo}(A, B) = \frac{A \cdot B}{\|A\|\|B\|}$$

2) **Pytanie 2: Jak wyszukiwane są najbliższe słowa?**

   Odpowiedź: Najbliższe słowa są wyszukiwane w sposób wyczerpujący. Dla danego słowa wejściowego (lub jego wektora), kod iteruje przez **wszystkie słowa** w słowniku modelu. W każdej iteracji obliczane jest podobieństwo kosinusowe między wektorem słowa wejściowego a wektorem bieżącego słowa ze słownika. Obliczone wartości podobieństwa są przechowywane (np. w słowniku), a następnie sortowane w porządku malejącym. Na koniec zwracane (lub wyświetlane) są słowa odpowiadające najwyższym wartościom podobieństwa (najbliżsi sąsiedzi).

3) **Pytanie 3: Ile słów ze słownika trzeba sprawdzić, aby odnaleźć najbliższe?**

Odpowiedź: Aby odnaleźć najbliższe słowa, analizowana implementacja musi porównać wektor słowa wejściowego z wektorami **wszystkich słów** znajdujących się w słowniku modelu. Potwierdza to obecność pętli ‘for i in range(self.v_count):‘ w metodach ‘vec_sim‘ i ‘word_sim‘, gdzie ‘self.v_count‘ oznacza całkowity rozmiar słownika. Oznacza to wykonanie V porównań (lub V-1, jeśli pominiemy porównanie słowa z samym sobą) dla słownika o rozmiarze V.

Analiza kodu potwierdza zastosowanie standardowych technik obliczania podobieństwa i wyszukiwania sąsiadów w podstawowej implementacji Word2Vec.

8. Zadanie z Notatnika Makemore: Klasteryzacja Embeddingów CBOW

W ramach analizy notatnika ‘LAB_makemore_part1_bigrams.ipynb‘ wykonano zadanie polegające na klasteryzacji i wizualizacji wektorów słów (embeddingów) uzyskanych z modelu CBOW (prawdopodobnie trenowanego na 4-gramach w tym samym notatniku).

Pierwszym krokiem była weryfikacja dostępnych danych: macierzy embeddingów oraz mapowania ‘word_to_ix‘. Na tym etapie ujawniono istotną niezgodność rozmiarów:

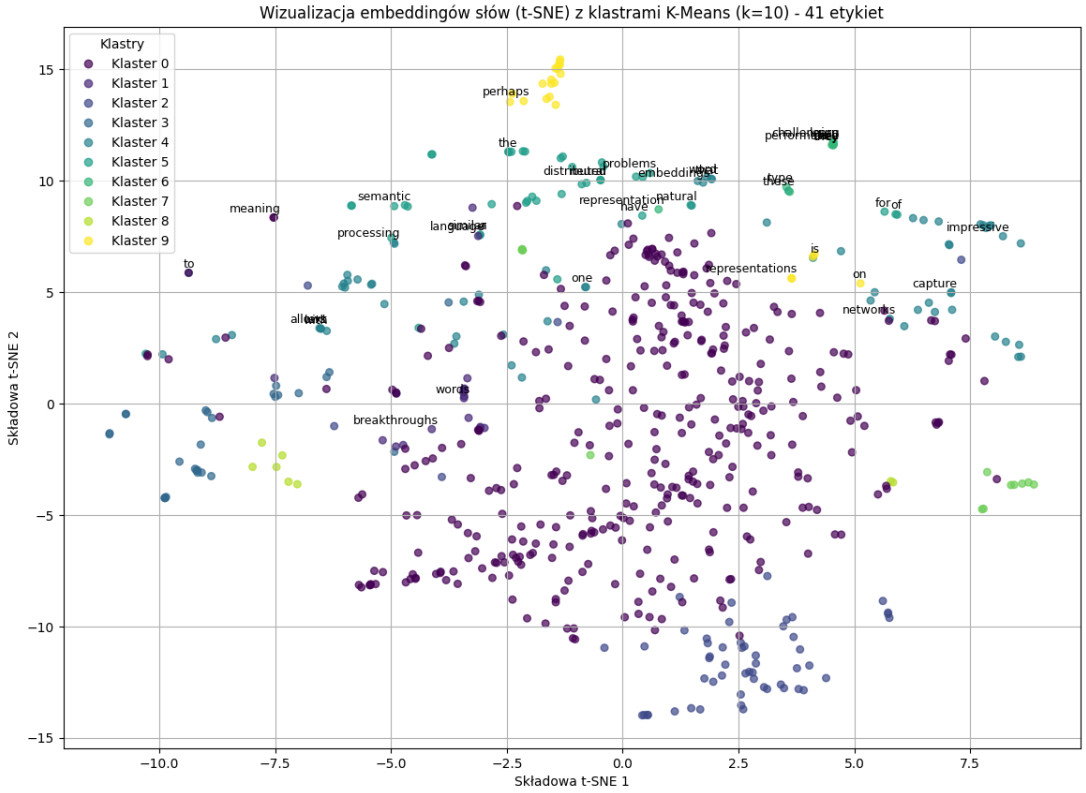
Listing 8.1. Output: Weryfikacja rozmiarów danych

```
Rozmiar slownika (ix_to_word): 41
Rozmiar macierzy embeddingow: 652
```

Istniało 652 wektorów embeddingów, podczas gdy słownik mapujący indeksy na słowa zawierał jedynie 41 pozycji. Mając świadomość tej niezgodności, przystąpiono do dalszych kroków, odpowiednio modyfikując proces wizualizacji i analizy klastrów.

Zastosowano algorytm K-Means (‘sklearn.cluster.KMeans‘) do pogrupowania wszystkich 652 wektorów embeddingów na ‘n_clusters = 10‘. Następnie, w celu wizualizacji, wymiarowość embeddingów zredukowano do 2D przy użyciu metody t-SNE (‘sklearn.manifold.TSNE‘).

Wygenerowano wykres punktowy (Rysunek 8.1), na którym przedstawiono wszystkie 652 punkty w przestrzeni 2D, pokolorowane zgodnie z przypisaniem do jednego z 10 klastrów. Kluczową modyfikacją w stosunku do pierwotnego kodu było dodanie etykiet tekstowych (słów) **jedynie do tych punktów (spośród 652), których indeksy (0-40) odpowiadały znanym słowom ze słownika ‘ix_to_word‘**. Punkty odpowiadające indeksom 41-651 pozostały bez etykiet.



Rys. 8.1. Wizualizacja (t-SNE) 652 wektorów embeddingów z modelu CBOW (4-gramy) z zaznaczonymi 10 klastrami K-Means. Etykiety tekstowe dodano tylko dla 41 znanych słów.

Wykres, mimo ograniczonej liczby etykiet, pozwala zaobserwować rozkład wszystkich punktów i ich podział na klastry. Można analizować wzajemne położenie tych 41 znanych słów.

Na koniec wyświetlono zawartość poszczególnych klastrów, uwzględniając jedynie słowa ze znanego, 41-elementowego słownika:

Listing 8.2. Output: Zawartość klastrów K-Means (tylko znane słowa)

```
 Sowa  pogrupowane w klastry (tylko te ze znanego sownika):
Klaster 0: meaning
Klaster 1: words, breakthroughs, language
Klaster 2: (brak znanych  sw )
Klaster 3: (brak znanych  sw )
Klaster 4: word, embeddings, that, allows, with, to, text, is, one, impressive,
    networks, natural, processing, capture
Klaster 5: a, of, similar, have, ., distributed, for, the, neural, problems, semantic,
     these
Klaster 6: are, type, representation, they, key, performance, challenging, learn
Klaster 7: (brak znanych  sw )
Klaster 8: (brak znanych  sw )
Klaster 9: perhaps, on, representations
```

Wyniki te pokazują, że wiele klastrów (2, 3, 7, 8) składało się wyłącznie z wektorów o nieznanych odpowiednikach słownych (indeksy >= 41). Pozostałe klastry grupowały znane słowa w sposób, który może odzwierciedlać pewne zależności semantyczne lub kontekstowe wyuczone przez model CBOW, pomimo problemów z rozmiarem embeddingów/słownika we wcześniejszych etapach.

9. Zadanie Bonusowe: Przewidywanie Ostatniego Słowa 4-gramu (PyTorch CBOW)

Zadanie bonusowe polegało na zaprojektowaniu i wytrenowaniu sieci neuronowej w podejściu CBOW, której celem było przewidzenie ostatniego (czwartego) słowa w sekwencji 4-gramowej na podstawie trzech poprzedzających słów. Wykorzystano do tego dane z pliku ‘raw_sentences.txt‘.

9.1. Przygotowanie Danych dla Zadania Bonusowego

Pierwszym krokiem było przygotowanie specyficznego zbioru danych treningowych. Kod wczytał plik ‘raw_sentences.txt‘, a następnie przefiltrował jego zawartość, zachowując jedynie te linie, które zawierały dokładnie cztery tokeny (słowa/znaki interpunkcyjne). Na podstawie tych poprawnych 4-gramów zbudowano dedykowany słownik oraz zestaw danych treningowych.

Listing 9.1. Output: Przetwarzanie danych dla zadania bonusowego

```
Liczba wczytanych poprawnych 4-gramow: 1445

Przykladowe 4-gramy:
[['People', 'still', 'do', '.'], ['Who', 'would', 'nt', '?'], ['Which', 'she', 'did',
    '.'], ['It', "'s", 'when', '.'], ['A', 'good', 'one', '?']]

Rozmiar slownika: 360

Liczba probek treningowych: 1445
Przykladowa probka (indeksy kontekstu[0:3], indeks celu[3]): ([0, 1, 2], 3)
Odpowiadajace slowa: (['People', 'still', 'do'], '.')
```

Jak pokazują wyniki, z pliku źródłowego wyodrębniono 1445 poprawnych 4-gramów. Słownik zbudowany wyłącznie na podstawie tych sekwencji zawiera 360 unikalnych tokenów (znacząco mniej niż w Zadaniu 6, gdzie analizowano cały plik). Wygenerowano 1445 próbek treningowych w formacie ‘([indeks_słowa1, indeks_słowa2, indeks_słowa3], indeks_słowa4)‘, co stanowi podstawę do treningu modelu przewidującego ostatni element sekwencji. Przykładowa próbka ilustruje to zadanie dla sekwencji ‘People still do .‘.

9.2. Definicja Modelu CBOW do Przewidywania Ostatniego Słowa

Po przygotowaniu danych zdefiniowano architekturę sieci neuronowej w PyTorch, dostosowaną do zadania przewidywania ostatniego słowa 4-gramu na podstawie pierwszych

trzech. Wykorzystano podejście CBOW, gdzie informacje z kontekstu (pierwsze trzy słowa) są agregowane w celu predykcji słowa docelowego (czwarte słowo).

Zdefiniowano klasę ‘PredictLastWordCBOW‘ zawierającą:

- Warstwę ‘torch.nn.Embedding‘ do nauki 50-wymiarowych wektorów dla 360 tokenów ze słownika.
- Metodę ‘forward‘, która pobiera embeddingi dla trzech słów kontekstowych, sumuje je, a następnie przepuszcza przez warstwę ‘torch.nn.Linear‘.
- Warstwę ‘torch.nn.Linear‘ przekształcającą 50-wymiarowy zagregowany wektor kontekstu na 360-wymiarowy wektor logitów (po jednym dla każdego słowa w słowniku).

W tej implementacji nie zastosowano funkcji aktywacji na wyjściu metody ‘forward‘, ponieważ standardowa funkcja kosztu ‘nn.CrossEntropyLoss‘ (wybrana do treningu) wewnętrznie zawiera już mechanizm Softmax.

Poniżej znajduje się fragment kodu definiującego model:

Listing 9.2. Definicja klasy modelu PredictLastWordCBOW (PyTorch)

```
class PredictLastWordCBOW(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(PredictLastWordCBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim, vocab_size)
    def forward(self, inputs):
        embeds = self.embeddings(inputs)
        context_vector = torch.sum(embeds, dim=1)
        out = self.linear1(context_vector)
        return out
```

Model został następnie zainicjalizowany z wymiarem embeddingu 50. Jako funkcję kosztu wybrano ‘nn.CrossEntropyLoss‘, a jako optymalizator ‘optim.Adam‘ z szybkością uczenia 0.01. Liczbę epok treningowych ustalono na 30. Struktura zainicjalizowanego modelu została potwierdzona przez output:

Listing 9.3. Output: Struktura zainicjalizowanego modelu PredictLastWordCBOW

```
Model PredictLastWordCBOW zdefiniowany:
PredictLastWordCBOW(
  (embeddings): Embedding(360, 50)
  (linear1): Linear(in_features=50, out_features=360, bias=True)
)
```

Model o takiej architekturze był gotowy do przeprowadzenia procesu treningu na przygotowanych wcześniej danych 4-gramowych.

9.3. Trening Modelu do Przewidywania Ostatniego Słowa

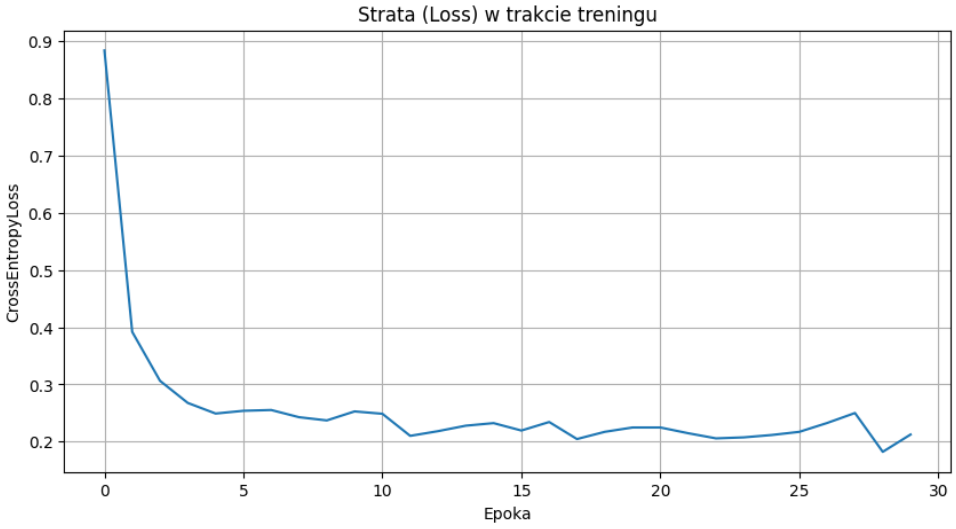
Po zdefiniowaniu modelu ‘PredictLastWordCBOW‘ przystąpiono do jego treningu na przygotowanym zbiorze 1445 próbek 4-gramowych. Proces uczenia zrealizowano w PyTorch, wykorzystując standardową pętlę treningową obejmującą 30 epok. W każdej epoce iterowano przez cały zbiór danych, wykonując dla każdej próbki kroki przejścia w przód, obliczenia straty (funkcja ‘nn.CrossEntropyLoss‘), przejścia wstecz oraz aktualizacji wag za pomocą optymalizatora Adam. Trening przeprowadzono na CPU.

Postęp uczenia monitorowano, zapisując średnią wartość funkcji kosztu po każdej epoce i wyświetlając ją co 5 epok:

Listing 9.4. Output: Postęp treningu modelu PredictLastWordCBOW

```
Trenowanie na urzadzeniu: cpu
Epoka: 5/30, rednia strata: 0.2491
Epoka: 10/30, rednia strata: 0.2528
Epoka: 15/30, rednia strata: 0.2324
Epoka: 20/30, rednia strata: 0.2248
Epoka: 25/30, rednia strata: 0.2116
Epoka: 30/30, rednia strata: 0.2125

Trening zakonczony.
```

Przebieg procesu uczenia został również zwizualizowany na wykresie funkcji kosztu w zależności od epoki (Rysunek 9.1).



Rys. 9.1. Wykres wartości funkcji kosztu (CrossEntropyLoss) w trakcie treningu modelu PredictLastWordCBOW.

Zarówno logi z konsoli, jak i wykres pokazują, że model uczył się bardzo szybko w początkowych epokach (strata gwałtownie malała). W późniejszych epokach strata ustabilizowała się na niskim poziomie (około 0.21-0.25), co wskazuje na osiągnięcie dobrej konwergencji na dostępnym zbiorze danych treningowych. Po 30 epokach model został uznany za wytrenowany i gotowy do ewentualnej ewaluacji lub wykorzystania do predykcji.

9.4. Testowanie Modelu - Predykcja Ostatniego Słowa

Aby sprawdzić działanie wytrenowanego modelu ‘PredictLastWordCBOW‘, zdefiniowano funkcję ‘predict_next_word‘. Funkcja ta przyjmuje model oraz sekwencję trzech słów kontekstowych, a następnie wykorzystuje model do przewidzenia najbardziej prawdopodobnego czwartego słowa. Funkcja zwraca również 5 najbardziej prawdopodobnych kandydatów wraz z ich prawdopodobieństwami, obliczonymi za pomocą funkcji Softmax na wyjściowych logitach modelu.

Funkcję przetestowano na dwóch przykładowych kontekstach pobranych bezpośrednio z danych treningowych:

Listing 9.5. Output: Wyniki predykcji dla przykładowych kontekstów

```
Kontekst: ['People', 'still', 'do']
Przewidywane nastepne slowo: '.' (Pewno: 1.000)
Top 5 przewidywan:
  - '.' (1.000)
  - 'Were' (0.000)
  - 'But' (0.000)
  - '?' (0.000)
  - 'life' (0.000)

Kontekst: ['That', "'s", 'right']
Przewidywane nastepne slowo: '.' (Pewno: 1.000)
Top 5 przewidywan:
  - '.' (1.000)
  - '?' (0.000)
  - 'put' (0.000)
  - 'today' (0.000)
  - 'Were' (0.000)
```

W obu testowych przypadkach model z bardzo dużą pewnością (praktycznie 100%) przewidział znak kropki (‘.‘) jako czwarte słowo sekwencji. Pozostałe możliwe słowa otrzymały znikome prawdopodobieństwa. Taki wynik sugeruje, że model silnie nauczył się wzorca kończenia tych konkretnych sekwencji znakiem interpunkcyjnym, co prawdopodobnie odzwierciedla strukturę danych treningowych (‘raw_sentences.txt‘). Choć predykcja

może być poprawna dla tych przykładów, tak ekstremalna pewność może wskazywać na potencjalne przeuczenie modelu na niewielkim zbiorze danych (1445 4-gramów).

10. Podsumowanie

Niniejsze laboratorium stanowiło praktyczne ćwiczenie z zakresu przetwarzania języka naturalnego, koncentrując się na metodach tworzenia wektorowych reprezentacji słów oraz podstawowych modelach sekwencyjnych. Głównym celem było zrozumienie i zastosowanie algorytmu Word2Vec w różnych wariantach (Gensim, PyTorch CBOW) do analizy semantycznej danych tekstowych, takich jak dialogi filmowe i sekwencje 4-gramowe.

Dodatkowo, zaimplementowano i porównano proste modele bigramowe (oparty na zliczeniach oraz neuronowy) do zadania generowania sekwencji na przykładzie polskich imion. Przeprowadzono również analizę implementacji Word2Vec "from scratch"oraz wykorzystano techniki redukcji wymiarowości (PCA, t-SNE) i klasteryzacji (K-Means) do wizualnej eksploracji i interpretacji uzyskanych przestrzeni embeddingów.

Ćwiczenia pozwoliły na zdobycie praktycznych umiejętności w zakresie przygotowania danych tekstowych, treningu modeli, ich oceny oraz wykorzystania kluczowych bibliotek, takich jak Gensim, PyTorch, Scikit-learn, Pandas i Spacy.