

O'REILLY®

Wydanie II

Data science od podstaw

Analiza danych w Pythonie



Helion

Joel Grus

Tytuł oryginału: Data Science from Scratch: First Principles with Python, 2nd Edition

Tłumaczenie: Wojciech Bombik, z wykorzystaniem fragmentów książki „Data science od podstaw. Analiza danych w Pythonie” w przekładzie Konrada Matuka

ISBN: 978-83-8322-131-1

© 2020, 2022 Helion S.A.

Authorized Polish translation of the English edition of *Data Science from Scratch, 2nd Edition*
ISBN 9781492041139 © 2019 Joel Grus.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/dascp2.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/dasc2v>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa do drugiego wydania	11
Przedmowa do pierwszego wydania	14
1. Wprowadzenie	17
Znaczenie danych	17
Czym jest analiza danych?	17
Hipotetyczna motywacja	18
Określanie najważniejszych węzłów	19
Analitycy, których możesz znać	21
Wynagrodzenie i doświadczenie	23
Płatne konta	25
Tematy interesujące użytkowników	26
Co dalej?	27
2. Błyskawiczny kurs Pythona	29
Zasady tworzenia kodu Pythona	29
Skąd wziąć interpreter Pythona?	30
Środowiska wirtualne	30
Formatowanie za pomocą białych znaków	31
Moduły	32
Polskie znaki diakrytyczne	33
Funkcje	33
Łańcuchy	34
Wyjątki	35
Listy	35
Krotki	36
Słowniki	37
defaultdict	38

Counter	39
Zbiory	39
Przepływ sterowania	40
Wartości logiczne	41
Sortowanie	42
Składanie list	42
Testy automatyczne i instrukcja assert	43
Programowanie obiektowe	43
Obiekty iterowalne i generatory	45
Losowość	46
Wyrażenia regularne	47
Narzędzia funkcyjne	48
Funkcja zip i rozpakowywanie argumentów	48
Argumenty nazwane i nienazwane	49
Adnotacje typów	50
Jak pisać adnotacje typów	52
Witaj w firmie DataSciencester!	53
Dalsza eksploracja	53
3. Wizualizacja danych	55
Pakiet matplotlib	55
Wykres słupkowy	57
Wykresy liniowe	60
Wykresy punktowe	60
Dalsza eksploracja	63
4. Algebra liniowa	65
Wektory	65
Macierze	69
Dalsza eksploracja	71
5. Statystyka	73
Opis pojedynczego zbioru danych	73
Tendencje centralne	74
Dyspersja	76
Korelacja	78
Paradoks Simpsona	80
Inne pułapki związane z korelacją	81
Korelacja i przyczynowość	81
Dalsza eksploracja	82

6. Prawdopodobieństwo	83
Zależność i niezależność	83
Prawdopodobieństwo warunkowe	84
Twierdzenie Bayesa	85
Zmienne losowe	87
Ciągły rozkład prawdopodobieństwa	87
Rozkład normalny	89
Centralne twierdzenie graniczne	91
Dalsza eksploracja	93
7. Hipotezy i wnioski	95
Sprawdzanie hipotez	95
Przykład: rzut monetą	95
Wartości p	98
Przedziały ufności	99
Hakowanie wartości p	100
Przykład: przeprowadzanie testu A-B	101
Wnioskowanie bayesowskie	102
Dalsza eksploracja	105
8. Metoda gradientu prostego	107
Podstawy metody gradientu prostego	107
Szacowanie gradientu	108
Korzystanie z gradientu	111
Dobór właściwego rozmiaru kroku	111
Używanie metody gradientu do dopasowywania modeli	112
Metody gradientu prostego: stochastyczna i minibatch	113
Dalsza eksploracja	114
9. Uzyskiwanie danych	117
Strumienie stdin i stdout	117
Wczytywanie plików	119
Podstawowe zagadnienia dotyczące plików tekstowych	119
Pliki zawierające dane rozdzielone separatorem	120
Pobieranie danych ze stron internetowych	122
HTML i parsowanie	122
Przykład: wypowiedzi kongresmenów	124
Korzystanie z interfejsów programistycznych	126
Format JSON (i XML)	126
Korzystanie z interfejsu programistycznego bez uwierzytelniania	127
Poszukiwanie interfejsów programistycznych	128

Przykład: korzystanie z interfejsów programistycznych serwisu Twitter	128
Uzyskiwanie danych uwierzytelniających	129
Dalsza eksploracja	132
10. Praca z danymi	133
Eksploracja danych	133
Eksploracja danych jednowymiarowych	133
Dwa wymiary	135
Wiele wymiarów	136
Wykorzystanie klasy NamedTuple	137
Dekorator dataclass	139
Oczyszczanie i wstępne przetwarzanie danych	140
Przetwarzanie danych	141
Przeskalowanie	144
Dygresja: tqdm	145
Redukcja liczby wymiarów	146
Dalsza eksploracja	151
11. Uczenie maszynowe	153
Modelowanie	153
Czym jest uczenie maszynowe?	154
Nadmierne i zbyt małe dopasowanie	154
Poprawność	157
Kompromis pomiędzy wartością progową a wariancją	159
Ekstrakcja i selekcja cech	161
Dalsza eksploracja	162
12. Algorytm k najbliższych sąsiadów	163
Model	163
Przykład: dane dotyczące irysów	165
Przekleństwo wymiarowości	168
Dalsza eksploracja	171
13. Naiwny klasyfikator bayesowski	173
Bardzo prosty filtr antyspamowy	173
Bardziej zaawansowany filtr antyspamowy	174
Implementacja	175
Testowanie modelu	177
Używanie modelu	178
Dalsza eksploracja	180

14. Prosta regresja liniowa	181
Model	181
Korzystanie z algorytmu spadku gradientowego	184
Szacowanie maksymalnego prawdopodobieństwa	185
Dalsza eksploracja	185
15. Regresja wieloraka	187
Model	187
Dalsze założenia dotyczące modelu najmniejszych kwadratów	188
Dopasowywanie modelu	189
Interpretacja modelu	190
Poprawność dopasowania	191
Dygresja: ładowanie wstępne	192
Błędy standardowe współczynników regresji	193
Regularyzacja	194
Dalsza eksploracja	196
16. Regresja logistyczna	197
Problem	197
Funkcja logistyczna	199
Stosowanie modelu	201
Poprawność dopasowania	202
Maszyny wektorów nośnych	203
Dalsza eksploracja	206
17. Drzewa decyzyjne	207
Czym jest drzewo decyzyjne?	207
Entropia	209
Entropia podziału	211
Tworzenie drzewa decyzyjnego	211
Łączenie wszystkiego w całość	214
Lasy losowe	216
Dalsza eksploracja	217
18. Sztuczne sieci neuronowe	219
Perceptrony	219
Jednokierunkowe sieci neuronowe	221
Propagacja wsteczna	224
Przykład: Fizz Buzz	226
Dalsza eksploracja	228

19. Uczenie głębokie	229
Tensor	229
Abstrakcja Layer	231
Warstwa Linear	233
Sieci neuronowe jako sekwencje warstw	235
Abstrakcja Loss i optymalizacja	235
Przykład: kolejne podejście do bramki XOR	237
Inne funkcje aktywacji	238
Przykład: kolejne podejście do gry Fizz Buzz	239
Funkcja softmax i entropia krzyżowa	240
Dropout	242
Przykład: MNIST	243
Zapisywanie i wczytywanie modeli	246
Dalsza eksploracja	247
20. Grupowanie	249
Idea	249
Model	250
Przykład: spotkania	252
Wybór wartości parametru k	253
Przykład: grupowanie kolorów	255
Grupowanie hierarchiczne z podejściem aglomeracyjnym	257
Dalsza eksploracja	261
21. Przetwarzanie języka naturalnego	263
Chmury wyrazowe	263
Modele n-gram	264
Gramatyka	267
Na marginesie: próbkowanie Gibbsa	269
Modelowanie tematu	271
Wektory słów	275
Rekurencyjne sieci neuronowe	283
Przykład: używanie rekurencyjnej sieci neuronowej na poziomie pojedynczych znaków	285
Dalsza eksploracja	288
22. Analiza sieci społecznościowych	289
Pośrednictwo	289
Centralność wektorów własnych	294
Mnożenie macierzy	294
Centralność	295
Grafy skierowane i metoda PageRank	297
Dalsza eksploracja	299

23. Systemy rekomendujące	301
Ręczne rozwiązywanie problemu	301
Rekomendowanie tego, co jest popularne	302
Filtrowanie kolaboratywne oparte na użytkownikach	303
Filtrowanie kolaboratywne oparte na zainteresowaniach	305
Faktoryzacja macierzy	307
Dalsza eksploracja	311
24. Bazy danych i SQL	313
Polecenia CREATE TABLE i INSERT	313
Polecenie UPDATE	316
Polecenie DELETE	316
Polecenie SELECT	317
Polecenie GROUP BY	319
Polecenie ORDER BY	321
Polecenie JOIN	322
Zapytania składowe	324
Indeksy	324
Optymalizacja zapytań	325
Bazy danych NoSQL	326
Dalsza eksploracja	326
25. Algorytm MapReduce	327
Przykład: liczenie słów	327
Dlaczego warto korzystać z algorytmu MapReduce?	329
Algorytm MapReduce w ujęciu bardziej ogólnym	330
Przykład: analiza treści statusów	331
Przykład: mnożenie macierzy	332
Dodatkowe informacje: zespaland	334
Dalsza eksploracja	334
26. Etyka przetwarzania danych	335
Czym jest etyka danych?	335
Ale tak naprawdę to czym jest etyka danych?	336
Czy powinienem przejmować się etyką danych?	336
Tworzenie złych produktów wykorzystujących dane	337
Kompromis między dokładnością a uczciwością	337
Współpraca	339
Interpretowalność	339
Rekomendacje	340
Tendencyjne dane	340

Ochrona danych	341
Podsumowanie	342
Dalsza eksploracja	342
27. Praktyka czyni mistrza	343
IPython	343
Matematyka	343
Korzystanie z gotowych rozwiązań	344
NumPy	344
pandas	344
scikit-learn	344
Wizualizacja	345
R	345
Uczenie głębokie	346
Szukanie danych	346
Zabierz się za analizę	346
Hacker News	347
Wozy straży pożarnej	347
Koszulki	347
Tweety na kuli ziemskiej	348
A Ty?	348

Przetwarzanie języka naturalnego

Byli na wielkiej uczcie języków i pokradli okruszyny.

— William Shakespeare

Przetwarzanie języka naturalnego (NLP — ang. *natural language processing*) jest określeniem technik przetwarzania danych związanych z językiem naturalnym. To bardzo szerokie pole, ale przyjrzymy się kilku technikom — zaczniemy od prostszych, aby później przejść do bardziej skomplikowanych.

Chmury wyrazowe

W rozdziale 1. obliczaliśmy liczbę wystąpień słów w zainteresowaniach użytkowników. Częstotliwość pojawiania się słów można zilustrować za pomocą chmury wyrazowej, w której słowa mają rozmiar zależny od liczby ich wystąpień w analizowanym tekście.

Ogólnie rzecz biorąc, analitycy danych rzadko korzystają z chmur słów między innymi dlatego, że punkty umieszczenia tych słów zwykle nie mają żadnego znaczenia, liczy się tylko ich wielkość.

Jeżeli kiedykolwiek spotkasz się z koniecznością utworzenia chmury wyrazowej, zastanów się nad tym, czy osie, wzdłuż których umieszczane są słowa, mogą mieć jakieś znaczenie. Załóżmy, że dysponujemy zbiorem słów związanych z nauką o danych, które są opisane dwoma wartościami znajdującymi się w zakresie od 0 do 100. Pierwsza z nich określa częstotliwość pojawiania się słowa w ofertach pracy, a druga w życiorysach zawodowych:

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),  
         ("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 60),  
         ("data science", 60, 70), ("analytics", 90, 3),  
         ("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0),  
         ("actionable insights", 40, 30), ("think out of the box", 45, 10),  
         ("self-starter", 30, 50), ("customer focus", 65, 15),  
         ("thought leadership", 35, 35)]
```

Utworzenie chmury wyrazowej polega tylko na ułożeniu słów w dostępnej przestrzeni w interesujący sposób (zobacz rysunek 21.1).



Rysunek 21.1. Chmura wyrazowa

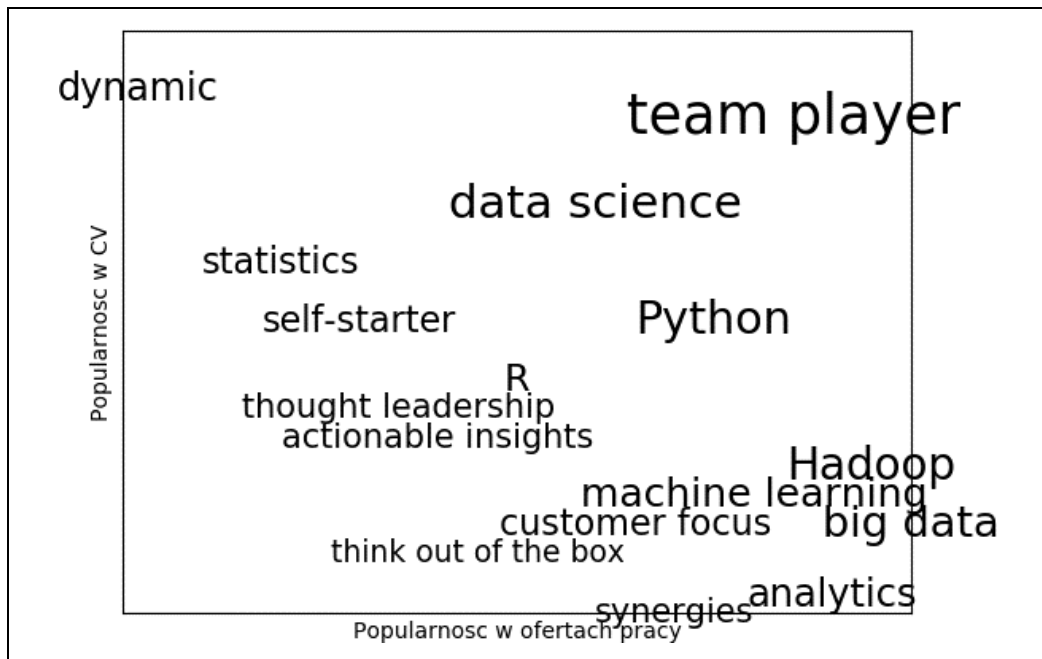
Chmura wyrazowa wygląda ładnie, ale tak naprawdę przekazuje mało informacji. Ciekawszym rozwiązaniem byłoby umieszczenie wyrazów tak, aby położenie wyrazów w płaszczyźnie poziomej było zależne od częstotliwości ich występowania w ofertach pracy, a położenie w płaszczyźnie pionowej było zależne od częstotliwości ich występowania w życiorysach zawodowych. Z takiej wizualizacji można wyciągnąć więcej wniosków (zobacz rysunek 21.2):

```
from matplotlib import pyplot as plt
def text_size(total: int) -> float:
    """Wynosi 8, jeżeli liczba wystąpień jest równa 0
    lub 28, jeżeli liczba wystąpień jest równa 200."""
    return 8 + total / 200 * 20

for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
plt.xlabel("Popularność w ofertach pracy")
plt.ylabel("Popularność w CV")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

Modele n-gram

Szefowa działu marketingu chce utworzyć tysiące stron internetowych związanych z nauką o danych w celu przesunięcia serwisu DataSciencester wyżej w wynikach wyszukiwania słów kluczowych związanych z analizą danych. Pomimo usilnych prób przekonania jej, że algorytmy wyszukiwarek są na tyle sprytne, że to rozwiązanie nic nie da, nie udało Ci się jej przekonać.



Rysunek 21.2. Chmura wyrazowa, która jest mniej atrakcyjna wizualnie, ale przekazuje więcej informacji

Oczywiście szefowa działu marketingu nie chce tworzyć samodzielnie tysiąca stron internetowych ani nie chce wynajmować hordy osób do tworzenia treści. Zdecydowała się poprosić Ciebie o programistyczne wygenerowanie takich stron. Aby to zrobić, musisz dysponować jakimś sposobem modelowania języka.

Jednym z możliwych rozwiązań jest zebranie korpusu dokumentów i użycie go w celu nauczania modelu statystycznego. Zaczniemy od napisanego przez Mike'a Loukidesa tekstu *What is data science?* (<http://oreil.ly/1Cd6ykN>), wyjaśniającego, czym jest nauka o danych.

Podobnie jak robiliśmy w rozdziale 9., będziemy korzystać z pakietów `requests` i `BeautifulSoup` w celu uzyskania danych, ale tym razem musimy zwrócić uwagę na kilka problemów.

Pierwszym z nich jest to, że apostrofy znajdujące się w tekście są tak naprawdę znakiem Unicode `u"\"u2019"`. W związku z tym potrzebujemy funkcji pomocniczej zastępującej je standardowymi apostrofami:

```
def fix_unicode(text: str) -> str:
    return text.replace(u"\"u2019", "'')
```

Drugim problemem jest to, że po pobraniu tekstu ze strony internetowej musimy podzielić go na sekwencję słów i kropek (kropki wskazują koniec zdania). Operacje te możemy wykonać za pomocą funkcji `re.findall()`:

```
import re
from bs4 import BeautifulSoup
import requests

url = "https://www.oreilly.com/ideas/what-is-data-science/"
```

```
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')

content = soup.find("div", "article-body") # Znajdź element div oznaczony etykietą article-body.
regex = r"[w']+|[\\.]" # Wybiera słowa i kropki.

document = []

for paragraph in content("p"):
    words = re.findall(regex, fix_unicode(paragraph.text))
    document.extend(words)
```

Oczywiście moglibyśmy (a nawet powinniśmy) oczyścić dane w większym stopniu. Zawierają one wciąż zbędne fragmenty tekstu (pierwszym słowem dokumentu jest np. *Section*), a dodatkowo dokonaliśmy podziału na kropkach znajdujących się w środku zdania (dotyczy to np. wyrażenia *Web 2.0*). Nasze dane zawierają ponadto sporo przypisów i list. Pomimo tego będziemy pracować z nimi w niezmienionej formie.

Podzieliśmy tekst na sekwencje słów, a więc możemy modelować język w następujący sposób: zakładając jakieś słowo początkowe (takie jak np. *book* — z ang. książka), szukamy w dokumencie wszystkich słów znajdujących się za nim. Wybieramy losowo jedno z nich i powtarzamy ten proces aż do uzyskania kropki sygnalizującej koniec zdania. Model taki określamy mianem **bigramu**, ponieważ zależy on w pełni od częstotliwości pojawiania się bigramów (par słów) w danych, na których model był uczony.

Co z pierwszym słowem? Możemy je wybrać losowo na podstawie słów *występujących po kropce*. Na początek zacznijmy od utworzenia możliwych przejść składających się ze słów. Przypominam, że funkcja `zip` przerywa działanie po zakończeniu przetwarzania jednego z obiektów wejściowych, a więc polecenie `zip(document, document[1:])` utworzy pary kolejnych elementów dokumentu `document`:

```
from collections import defaultdict
transitions = defaultdict(list)
for prev, current in zip(document, document[1:]):
    transitions[prev].append(current)
```

Teraz możemy przystąpić do generowania zdań:

```
def generate_using_bigrams() -> str:
    current = "." # Kolejne słowo rozpoczyna nowe zdanie.
    result = []
    while True:
        next_word_candidates = transitions[current] # bigramy (current, _)
        current = random.choice(next_word_candidates) # Wylusuj.
        result.append(current) # Dołącz do wyniku.
        if current == ".": return " ".join(result) # Zakończ pracę w przypadku kropki.
```

Wygenerowane zdania są bełkotem, ale bełkot ten można umieścić na stronie internetowej mającej zawierać treści, które mają brzmieć tak, jakby dotyczyły nauki o danych. Oto przykład wygenerowanego zdania:

If you may know which are you want to data sort the data feeds web friend someone on trending topics as the data in Hadoop is the data science requires a book demonstrates why visualizations are but we do massive correlations across many commercial disk drives in Python language and creates more tractable form making connections then use and uses it to solve a data.

— model bigram

Jeżeli skorzystamy z trigramów (trzech występujących po sobie słów), to generowany tekst będzie mniej bełkotliwy. Ogólnie rzecz biorąc, możesz korzystać z n -gramów — n występujących po sobie słów, ale w naszym przypadku wystarczy, że będziemy korzystać z trzech takich słów. Teraz przejście będzie zależeć od dwóch poprzednich słów:

```
trigram_transitions = defaultdict(list)
starts = []

for prev, current, next in zip(document, document[1:], document[2:]):

    if prev == ".":          # Jeżeli poprzednim „słowem” była kropka,
        starts.append(current) # to zacznij od tego słowa

    trigram_transitions[(prev, current)].append(next)
```

Zwróć uwagę na to, że teraz musimy dysponować oddzielnym mechanizmem śledzenia słów początkowych. Zdania generowane są w bardzo podobny sposób:

```
def generate_using_trigrams() -> str:
    current = random.choice(starts) # Wylosuj pierwsze słowo.
    prev = "."                      # Umieść przed nim kropkę.
    result = [current]
    while True:
        next_word_candidates = trigram_transitions[(prev, current)]
        next_word = random.choice(next_word_candidates)

        prev, current = current, next_word
        result.append(current)

    if current == ".":
        return " ".join(result)
```

W ten sposób możemy wygenerować sensowniejsze zdania. Oto przykład takiego zdania:

In hindsight MapReduce seems like an epidemic and if so does that give us new insights into how economies work That's not a question we could even have asked a few years there has been instrumented.

— model trigram

Zdania brzmią lepiej, ponieważ ograniczyliśmy liczbę opcji dostępnych na każdym etapie ich generowania, a w wielu sytuacjach algorytm nie miał żadnych dodatkowych opcji, spośród których mógłby dokonać wyboru. W związku z tym często generowane będą zdania (lub przynajmniej długie frazy), które dokładnie w takiej samej formie występowały w oryginalnych danych. Problem ten rozwiązałoby rozszerzenie zbioru danych — zbiór ten warto byłoby rozszerzyć o n -gramy pochodzące z różnych prac naukowych dotyczących analizy danych.

Gramatyka

Język może być modelowany również na podstawie *zasad gramatyki* — reguł tworzenia akceptowalnych zdań. W szkole podstawowej uczono Cię, czym są części mowy i jak należy je ze sobą łączyć. Jeżeli miałeś kiepskiego nauczyciela od angielskiego, to z pewnością słyszałeś twierdzenie, że zdanie musi się składać z *rzeczownika*, po którym ma znaleźć się *czasownik*. Z reguły tej moglibyśmy skorzystać podczas generowania zdań, gdybyśmy dysponowali listą rzeczowników i czasowników.

Zdefiniujemy nieco bardziej złożone zasady gramatyczne:

```
from typing import List, Dict

# alias typu
Grammar = Dict[str, List[str]]

grammar = {
    "_S" : ["_NP", "_VP"],
    "_NP" : ["_N",
            "_A", "_NP", "_P", "_A", "_N"],
    "_VP" : ["_V",
            "_V", "_NP"],
    "_N" : ["data science", "Python", "regression"],
    "_A" : ["big", "linear", "logistic"],
    "_P" : ["about", "near"],
    "_V" : ["learns", "trains", "tests", "is"]
}
```

Przyjąłem konwencję, według której nazwy rozpoczynające się od znaków podkreślenia odwołują się do *reguł* wymagających dalszego rozszerzenia (reguły o innych nazwach nie wymagają tego).

Reguła `_S` jest regułą „zdania”, które wymaga skorzystania z reguły `_NP` (umieszczenia frazy rzeczownikowej) i reguły `_VP` (umieszczenia frazy czasownikowej po frazie rzeczownikowej).

Reguła frazy rzeczownikowej może prowadzić do reguły `_V` (rzeczownika) lub reguły rzeczownika, po której zostanie umieszczona reguła frazy rzeczownikowej.

Zauważ, że reguła `_NP` zawiera samą siebie na liście własnych reguł składowych. Zasady gramatyki mogą mieć charakter rekurencyjny, co pozwala na generowanie nieskończenie wielu różnych zdań.

Jak można skorzystać z tych zasad w celu wygenerowania zdań? Zaczniemy od utworzenia listy zawierającej regułę zdania `["_S"]`, a następnie będziemy tworzyć kolejne listy, rozbudowując ją o reguły losowo wybrane ze zbioru dozwolonych reguł. Proces rozszerzania budowy list zakończymy po uzyskaniu listy zawierającej same węzły końcowe.

Oto przykład progresywnej rozbudowy takich list:

```
['_S']
['_NP', '_VP']
['_N', '_VP']
['Python', '_VP']
['Python', '_V', '_NP']
['Python', 'trains', '_NP']
['Python', 'trains', '_A', '_NP', '_P', '_A', '_N']
['Python', 'trains', 'logistic', '_NP', '_P', '_A', '_N']
['Python', 'trains', 'logistic', '_N', '_P', '_A', '_N']
['Python', 'trains', 'logistic', 'data science', '_P', '_A', '_N']
['Python', 'trains', 'logistic', 'data science', 'about', '_A', '_N']
['Python', 'trains', 'logistic', 'data science', 'about', 'logistic', '_N']
['Python', 'trains', 'logistic', 'data science', 'about', 'logistic', 'Python']
```

Jak możemy to zaimplementować? Zaczniemy od utworzenia prostej funkcji identyfikującej reguły, które można rozbudować:

```
def is_terminal(token: str) -> bool:
    return token[0] != "_"
```

Potrzebujemy jeszcze funkcji zwracającej listę znaczników token do zdania. Będziemy szukać pierwszego niedającego się rozbudować znacznika. Znalezienie go oznacza zbudowanie całego zdania.

W przypadku znalezienia znacznika niekońcowego należy wylosować jeden z jego produktów. Jeżeli znacznik jest końcowy, to należy zamiast niego umieścić słowo. W innym przypadku mamy do czynienia z sekwencją rozdzielonych spacjami znaczników niekońcowych, które musimy rozdzielić, a następnie zastąpić bieżącymi znacznikami. Tak czy inaczej proces jest powtarzany na nowym zestawie znaczników.

Implementując te wszystkie mechanizmy, uzyskujemy następujący kod:

```
def expand(grammar: Grammar, tokens: List[str]) -> List[str]:
    for i, token in enumerate(tokens):

        # Ignoruj węzły końcowe.
        if is_terminal(token): continue

        # Wylosuj element zastępujący (znaleziono znacznik niekońcowy).
        replacement = random.choice(grammar[token])

        if is_terminal(replacement):
            tokens[i] = replacement
        else:
            # Replacement może być np. "_NP_VP", więc musimy podzielić po spacjach
            tokens = tokens[:i] + replacement.split() + tokens[(i+1):]

        # Wywołaj rozszerzanie nowej listy znaczników.
        return expand(grammar, tokens)

    # W tym momencie wszystkie węzły zostały obsłużone.
    return tokens
```

Teraz możemy rozpocząć proces generowania zdań:

```
def generate_sentence(grammar: Grammar) -> List[str]:
    return expand(grammar, ["_S"])
```

Poeksperymentuj — dodaj więcej słów, utwórz więcej reguł, wprowadź kolejne części mowy i wygeneruj tyle stron internetowych, ilu potrzebuje Twoja firma.

Zasady gramatyczne są o wiele bardziej interesujące, gdy używa się ich do wykonywania odwrotnej operacji — **parsowania** zdań. Wówczas zasady gramatyczne pozwalają na identyfikację podmiotów i czasowników, a także określenie sensu zdania.

Generowanie tekstu za pomocą technik analitycznych to dość ciekawe zagadnienie, ale próba *zrozumienia* tekstu jest czymś jeszcze bardziej interesującym (informacje o bibliotekach przeznaczonych do analizy tekstu znajdziesz w podrozdziale „Dalsza eksploracja” znajdującym się na końcu tego rozdziału).

Na marginesie: próbkowanie Gibbsa

Generowanie próbek z niektórych rozkładów jest łatwe. W celu wygenerowania losowych zmiennych z rozkładu jednostajnego wystarczy użyć funkcji:

```
random.random()
```

W celu wygenerowania losowych wartości zmiennych z rozkładu normalnego wystarczy użyć kodu:

```
inverse_normal_cdf(random.random())
```

Niestety generowanie próbek z niektórych rozkładów jest o wiele trudniejsze. **Próbkowanie Gibbsa** (ang. *Gibbs sampling*) jest techniką generowania próbek z wielowymiarowych rozkładów wtedy, gdy znamy tylko niektóre uwarunkowania takich rozkładów.

Załóżmy, że np. rzucamy dwoma kostkami. Niech x oznacza liczbę oczek wyrzuconą na pierwszej kostce, a y oznacza sumę liczby oczek wyrzuconych na obu kostkach. Załóżmy, że chcemy wygenerować dużo par wartości (x, y) . W takim przypadku bezpośrednie generowanie par nie stanowi trudności:

```
from typing import Tuple
import random

def roll_a_die() -> int:
    return random.choice([1,2,3,4,5,6])

def direct_sample() -> Tuple[int, int]:
    d1 = roll_a_die()
    d2 = roll_a_die()
    return d1, d1 + d2
```

A co, gdybyśmy znali tylko rozkłady warunkowe? Rozkład y jest uwarunkowany od x . Znając wartość x , można stwierdzić, że prawdopodobieństwo tego, że y jest równe $x + 1$, $x + 2$, $x + 3$, $x + 4$, $x + 5$ lub $x + 6$, jest w każdym przypadku takie samo:

```
def random_y_given_x(x: int) -> int:
    """Może wynosić x + 1, x + 2, ..., x + 6"""
    return x + roll_a_die()
```

Pójście w drugą stronę nie byłoby już takie łatwe. Gdybyśmy wiedzieli, że $y = 2$, to x musi być równe 1 (obie wartości muszą dać w sumie 2, a na obu kostkach musi wypaść po jednym oczku). Jeżeli wiemy, że $y = 3$, to x może równie dobrze być równe 1, jak i 2. Z taką samą sytuacją mamy do czynienia, jeżeli $y = 11$ (wtedy x może być równe 5 lub 6):

```
def random_x_given_y(y: int) -> int:
    if y <= 7:
        # Jeżeli suma jest równa 7 lub mniej, to na pierwszej kostce mogły wypaść wartości 1, 2, ..., (suma - 1).
        return random.randrange(1, y)
    else:
        # Jeżeli suma jest większa od 7, to na pierwszej kostce
        # mogły równie prawdopodobnie wypaść wartości (suma - 6), (suma - 5), ..., 6.
        return random.randrange(y - 6, 7)
```

Próbkowanie Gibbsa działa tak, że zaczynamy od pierwszej (poprawnej) wartości dla x i y , a następnie powtarzamy zastępowanie x losową wartością wybraną pod warunkiem y i zastępowanie y losową wartością wybraną pod warunkiem x . Po kilku iteracjach uzyskane wartości x i y będą reprezentowały próbkę z rozkładu bezwarunkowo połączonego:

```
def gibbs_sample(num_iters: int = 100) -> Tuple[int, int]:
    x, y = 1, 2 # To tak naprawdę nie ma znaczenia.
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

Możemy sprawdzić, że uzyskamy wynik podobny do bezpośredniego próbkowania:

```
def compare_distributions(num_samples: int = 1000) -> Dict[int, List[int]]:
    counts = defaultdict(lambda: [0, 0])
    for _ in range(num_samples):
```

```
counts[gibbs_sample()][0] += 1
counts[direct_sample()][1] += 1
return counts
```

Z techniki tej skorzystamy w kolejnym podrozdziale.

Modelowanie tematu

Tworząc mechanizm rekomendacyjny w rozdziale 1., szukaliśmy dokładnie pasujących do siebie elementów na listach zainteresowań deklarowanych przez użytkowników.

Bardziej zaawansowanym sposobem na zrozumienie zainteresowań użytkowników może być próba zidentyfikowania *tematów* z nimi związanych. Często w tego typu przypadkach korzysta się z **analizy LDA** (ang. *Latent Dirichlet Analysis*). Analiza ta jest stosowana w celu określania tematów zbioru dokumentów. Użyjemy jej do przetworzenia dokumentów składających się z zainteresowań każdego z użytkowników.

Analiza LDA przypomina nieco naiwny klasyfikator bayesowski, który zbudowaliśmy w rozdziale 13. — zakłada model probabilistyczny dokumentów. Nie będziemy zagłębiać się zbyt głęboko w szczegóły matematyczne, ale możemy przyjąć, że model zakłada, iż:

- Istnieje określona liczba K tematów.
- Istnieje zmienna losowa przypisująca każdemu tematowi rozkład probabilistyczny słów. Rozkład ten należy traktować jako rozkład prawdopodobieństwa wystąpienia słowa w w temacie k .
- Istnieje inna zmienna losowa przypisująca każdemu dokumentowi rozkład prawdopodobieństwa tematów. Rozkład ten należy traktować jako mieszanie tematów dokumentu d .
- Każde słowo w dokumencie zostało wygenerowane poprzez losowy wybór tematu (z rozkładu tematów dokumentu), a następnie losowy wyraz słowa (z rozkładu słów określonego wcześniej tematu).

Dysponujemy zbiorem dokumentów (`documents`). Każdy z tych dokumentów jest listą słów. Dysponujemy również odpowiadającym temu zbiorowi zbiorem tematów (`documents_topics`), który przypisuje temat (w tym przypadku liczbę z zakresu od 0 do $K - 1$) do każdego słowa znajdującego się w każdym dokumencie.

W związku z tym piąte słowo w czwartym dokumencie można odczytać za pomocą kodu:

```
documents[3][4]
```

Temat, z którego to słowo zostało wybrane, można określić za pomocą kodu:

```
document_topics[3][4]
```

Mamy więc do czynienia z bardzo jawną definicją rozkładu dokumentów według tematów i niejawną definicją rozkładu tematów według słów.

Możemy oszacować prawdopodobieństwo tego, że temat 1 zwróci określone słowo, porównując liczbę określającą, ile razy to słowo zostało wygenerowane przez ten temat, z liczbą wszystkich operacji generowania słów z tego tematu. (Podobne rozwiązanie stosowaliśmy w rozdziale 13. — porównywaliśmy liczbę wystąpień danego słowa w spamie z całkowitą liczbą słów pojawiających się w spamie).

Tematy są tylko liczbami, ale możemy nadać im również nazwy opisowe poprzez analizę słów, do których przywiązują one największą wagę. Potrzebujemy sposobu wygenerowania zbioru `document_topics`. Do tego przyda się nam właśnie próbkowanie Gibbsa.

Zaczynamy od losowego przypisania tematu do wszystkich słów znajdujących się we wszystkich dokumentach. Na razie będziemy przetwarzać każdy dokument słowo po słowie. Dla danego słowa występującego w danym dokumencie będziemy konstruować wagi każdego tematu, który zależy od (bieżącego) rozkładu tematów w tym dokumencie i (bieżącego) rozkładu słów tego tematu. Następnie używamy tych wag do próbkowania nowego tematu dla danego słowa. Jeżeli powtórzymy ten proces wielokrotnie, to uzyskamy łączną próbkę z rozkładu temat-słowo i rozkładu dokument-temat.

Na początek musimy utworzyć funkcję losującą indeks na podstawie dowolnego zestawu wag:

```
def sample_from(weights: List[float]) -> int:
    """zwraca i z prawdopodobieństwem weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random()      # Rozkład jednostajny od 0 do sumy.
    for i, w in enumerate(weights):
        rnd -= w                       # Zwraca najmniejszą wartość i spełniającą warunek:
        if rnd <= 0: return i          # weights[0] + ... + weights[i] >= rnd.
```

Po przekazaniu do tej funkcji wag [1, 1, 3] będzie ona zwracała przez jedną piątą czasu 0, przez jedną piątą czasu 1, a przez trzy piąte czasu będzie zwracała 2. Możemy to sprawdzić:

```
from collections import Counter

# wylosuj 1000 razy i policz
draws = Counter(sample_from([0.1, 0.1, 0.8]) for _ in range(1000))
assert 10 < draws[0] < 190 # powinno być ~10%
assert 10 < draws[1] < 190 # powinno być ~10%
assert 650 < draws[2] < 950 # powinno być ~80%
assert draws[0] + draws[1] + draws[2] == 1000
```

Naszymi dokumentami są zainteresowania użytkowników. Mają one następującą formę:

```
documents = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

Spróbujemy znaleźć $K = 4$ tematów.

W celu obliczenia wag próbkowania musimy śledzić kilka wartości. Musimy utworzyć strukturę danych, w której je umieścimy.

- Ile razy każdy temat jest przypisywany do każdego dokumentu:

```
#Lista liczników; po jednym dla każdego dokumentu.  
document_topic_counts = [Counter() for _ in documents]
```

- Ile razy każde słowo jest przypisywane do każdego tematu:

```
#Lista liczników; po jednym dla każdego tematu.  
topic_word_counts = [Counter() for _ in range(K)]
```

- Suma słów przypisanych do każdego tematu:

```
#Lista liczb; po jednej dla każdego tematu.  
topic_counts = [0 for _ in range(K)]
```

- Całkowita liczba słów znajdujących się w każdym dokumencie:

```
#Lista liczb; po jednej dla każdego dokumentu.  
document_lengths = [len(document) for document in documents]
```

- Liczba unikalnych słów:

```
distinct_words = set(word for document in documents for word in document)  
W = len(distinct_words)
```

- Liczba dokumentów:

```
D = len(documents)
```

Po określeniu tych wartości możemy ustalić np. liczbę słów występujących w dokumencie `documents[3]` dotyczących tematu nr 1:

```
document_topic_counts[3][1]
```

Ponadto możemy określić, ile razy słowo *nlp* pojawiło się w związku z tematem nr 2:

```
topic_word_counts[2]["nlp"]
```

Teraz możemy zdefiniować nasze funkcje prawdopodobieństwa warunkowego. Każda z nich, podobnie jak w rozdziale 13., będzie miała czynnik wygładzający, zapewniający to, że każdy temat ma niezerową szansę na bycie wybranym w dowolnym dokumencie, oraz to, że każde słowo ma niezerową szansę na bycie wybranym w dowolnym temacie:

```
def p_topic_given_document(topic: int, d: int, alpha: float = 0.1) -> float:  
    """Ułamek słów w dokumencie _d_, które są przypisane  
    do tematu _topic_ plus wartość wygładzająca."""  
  
    return ((document_topic_counts[d][topic] + alpha) /  
            (document_lengths[d] + K * alpha))  
  
def p_word_given_topic(word: str, topic: int, beta: float = 0.1) -> float:  
    """Ułamek słów _word_ przypisanych do tematu _topic_,  
    plus wartość wygładzająca."""  
  
    return ((topic_word_counts[topic][word] + beta) /  
            (topic_counts[topic] + W * beta))
```

Z funkcji tych będziemy korzystać w celu utworzenia wag do aktualizacji tematów:

```
def topic_weight(d: int, word: str, k: int) -> float:  
    """Zwróć wagę k-tego tematu na podstawie  
    dokumentu i występującego w nim słowa"""  
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)
```

```
def choose_new_topic(d: int, word: str) -> int:
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

Sposób definicji funkcji `topic_weight` obliczającej wagi tematów wynika z pewnych zasad matematycznych, ale niestety jest to zbyt krótka książka, aby wyjaśnić w niej szczegóły matematyczne tej implementacji. Na szczęście działanie tej funkcji jest intuicyjne — prawdopodobieństwo wybrania dowolnego tematu na podstawie słowa i dokumentu zależy od tego, jakie jest prawdopodobieństwo wystąpienia wybranego tematu w danym dokumencie, a także od prawdopodobieństwa wystąpienia danego słowa w wybranym temacie.

Dysponujemy już wszystkimi niezbędnymi mechanizmami. Zaczniemy od przypisania każdego słowa do wybranego losowo tematu, a następnie wygenerujemy odpowiednie wartości liczników:

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                    for document in documents]

for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

Naszym celem jest uzyskanie połączonej próbki rozkładu tematy-słowa i rozkładu dokumenty-tematy. Zrobimy to za pomocą próbkowania Gibbsa korzystającego ze zdefiniowanych wcześniej prawdopodobieństw warunkowych:

```
import tqdm
for iter in tqdm.trange(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):

            # Odejmij parę słowo-temat od sumy, aby nie wpływała ona na wagę.
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1

            # Wybierz nowy temat na podstawie wag.
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic

            # Dodaj go z powrotem do sumy.
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

Czym są tematy? Są one liczbami 0, 1, 2 i 3. Jeżeli chcemy nadać im nazwy, to musimy je określić samodzielnie. Przyjrzyjmy się pięciu słowom, które uzyskały największą wagę w każdym z tematów (zobacz tabelę 21.1):

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0: print(k, word, count)
```

Tabela 21.1. Najpopularniejsze słowa występujące w poszczególnych tematach

Temat 0	Temat 1	Temat 2	Temat 3
<i>Java</i>	<i>R</i>	<i>HBase</i>	<i>regression</i>
<i>Big Data</i>	<i>statistics</i>	<i>Postgres</i>	<i>libsvm</i>
<i>Hadoop</i>	<i>Python</i>	<i>MongoDB</i>	<i>scikit-learn</i>
<i>deep learning</i>	<i>probability</i>	<i>Cassandra</i>	<i>machine learning</i>
<i>artificial intelligence</i>	<i>pandas</i>	<i>NoSQL</i>	<i>neural networks</i>

Przeglądając się słowom przypisanym do tematów, przypisałbym im następujące nazwy:

```
topic_names = ["Big Data and programming languages", # Duże zbiory danych i języki programowania
               "Python and statistics",           # Python i statystyka
               "databases",                       # Bazy danych
               "machine learning"]                # Uczenie maszynowe
```

Teraz możemy sprawdzić, jak model przypisuje tematy do zainteresowań użytkowników:

```
for document, topic_counts in zip(documents, document_topic_counts):
    print(document)
    for topic, count in topic_counts.most_common():
        if count > 0:
            print(topic_names[topic], count)
    print()
```

Po uruchomieniu tego kodu uzyskamy następujące dane:

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4 databases 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5 machine learning 1
```

Nazwy tematów zawierają spójnik „i”, a więc prawdopodobnie można wydzielić większą liczbę tematów, ale niestety dysponujemy zbyt małym treningowym zbiorem danych, aby wytrenować model w celu rozpoznawania większej liczby tematów.

Wektory słów

Duża część ostatnich postępów w przetwarzaniu języka naturalnego ma związek z uczeniem głębokim. W dalszej części tego rozdziału przyjrzymy się kilku z nich, używając narzędzi, które napisaliśmy w rozdziale 19.

Jedną z ważnych innowacji jest przedstawianie słów jako kilkowymiarowych wektorów. Można z nimi zrobić wiele rzeczy — porównywać, dodawać do siebie lub wykorzystywać w modelach uczenia maszynowego. Wektory takie mają kilka przydatnych cech. Na przykład podobne słowa są zazwyczaj przedstawiane podobnymi wektorami. Często więc wektor słowa „duży” będzie zbliżony do wektora słowa „wielki”, dzięki czemu modele działające na takich wektorach będą mogły (do pewnego stopnia) radzić sobie z synonimami niskim kosztem.

Wektory wykazują też ciekawe właściwości arytmetyczne. Na przykład, jeżeli w niektórych modelach od wektora słowa „król” odejmiemy wektor słowa „mężczyzna” i dodamy wektor słowa „kobieta”, to

możemy uzyskać wektor zbliżony do słowa „królowa”. Rozważania na temat tego, co to dokładnie oznacza, mogą być interesujące, ale nie będziemy się nimi teraz zajmować.

Stworzenie takich wektorów dla dużego słownika jest trudnym zadaniem, więc zazwyczaj będziemy korzystać z korpusów językowych. Jest kilka sposobów działania, ale z grubsza wszystkie wyglądają tak:

1. Pobierz tekst.
2. Utwórz zbiór danych tak, aby spróbować przewidywać słowa na podstawie słów sąsiadujących (lub na odwrót — na podstawie słowa przewidywać jego sąsiadów).
3. Wytrenuj sieć neuronową, aby potrafiła poradzić sobie z tym zadaniem.
4. Utwórz wektory słów z wewnętrznych stanów wytrenowanych neuronów.

Naszym celem jest przewidywanie słów na podstawie wyrazów znajdujących się w pobliżu. Słowa, które pojawiają się w podobnych kontekstach (a więc również w towarzystwie podobnych wyrazów), powinny mieć podobne stany wewnętrzne, więc ich wektory również będą zbliżone do siebie.

Podobieństwo wektorów będziemy mierzyć przy użyciu tzw. podobieństwa kosinusowego, które przyjmuje wartości z przedziału od -1 do 1 i określa stopień, w jakim dwa wektory wskazują ten sam kierunek:

```
from scratch.linear_algebra import dot, Vector
import math

def cosine_similarity(v1: Vector, v2: Vector) -> float:
    return dot(v1, v2) / math.sqrt(dot(v1, v1) * dot(v2, v2))

assert cosine_similarity([1., 1, 1], [2., 2, 2]) == 1, "ten sam kierunek"
assert cosine_similarity([-1., -1], [2., 2]) == -1, "przeciwny kierunek"
assert cosine_similarity([1., 0], [0., 1]) == 0, "ortogonalne"
```

Wytrenujemy kilka wektorów, aby zobaczyć, jak to działa.

Na początek będziemy potrzebować przykładowego zbioru danych. Zazwyczaj wektory słów są uzyskiwane poprzez modele trenowane na milionach lub nawet miliardach słów. Ponieważ nasze programy nie poradzą sobie z tak dużymi ilościami danych, stworzymy sztuczny zbiór danych o określonej strukturze:

```
colors = ["red", "green", "blue", "yellow", "black", ""]
nouns = ["bed", "car", "boat", "cat"]
verbs = ["is", "was", "seems"]
adverbs = ["very", "quite", "extremely", ""]
adjectives = ["slow", "fast", "soft", "hard"]

def make_sentence() -> str:
    return " ".join([
        "The",
        random.choice(colors),
        random.choice(nouns),
        random.choice(verbs),
        random.choice(adverbs),
        random.choice(adjectives),
        "."
    ])
])
```



```
NUM_SENTENCES = 50

random.seed(0)
sentences = [make_sentence() for _ in range(NUM_SENTENCES)]
```

W ten sposób wygenerujemy wiele zdań o podobnej strukturze, ale składających się z różnych słów, takich jak na przykład: „The green boat seems quite slow”. W utworzonych zdaniach kolory będą zazwyczaj pojawiać się w podobnych kontekstach, podobnie jak rzeczowniki i pozostałe części mowy. Jeżeli więc przypiszemy wektory słów w odpowiedni sposób, słowa z podobnych kategorii powinny uzyskać podobne wektory.



Podczas prawdziwej analizy wektorów prawdopodobnie korzystalibyśmy ze zbioru mającego miliony zdań. W takim przypadku uzyskalibyśmy określone konteksty z samego tekstu. Analizując jedynie 50 zdań, musimy niestety narzucić konteksty w nieco sztuczny sposób.

Do dalszej analizy zamienimy słowa na identyfikatory. Wykorzystamy do tego celu klasę Vocabulary:

```
from scratch.deep_learning import Tensor

class Vocabulary:
    def __init__(self, words: List[str] = None) -> None:
        self.w2i: Dict[str, int] = {} # mapowanie słowa na jego identyfikator
        self.i2w: Dict[int, str] = {} # mapowanie identyfikatora na słowo

        for word in (words or []): # jeżeli na wejściu były jakieś słowa, to je dodajemy.
            self.add(word)

    @property
    def size(self) -> int:
        """ile słów jest w słowniku"""
        return len(self.w2i)

    def add(self, word: str) -> None:
        if word not in self.w2i:
            word_id = len(self.w2i) # jeżeli słowo jest nowe,
            self.w2i[word] = word_id # znajdź kolejny identyfikator.
            self.i2w[word_id] = word # dodaj do mapowania słów na identyfikatory
            # dodaj do mapowania identyfikatorów na słowa

    def get_id(self, word: str) -> int:
        """zwraca identyfikator słowa (lub None)"""
        return self.w2i.get(word)

    def get_word(self, word_id: int) -> str:
        """zwraca słowo o określonym identyfikatorze (lub None)"""
        return self.i2w.get(word_id)

    def one_hot_encode(self, word: str) -> Tensor:
        word_id = self.get_id(word)
        assert word_id is not None, f"nieznane słowo {word}"

        return [1.0 if i == word_id else 0.0 for i in range(self.size)]
```

Wszystkie te rzeczy można by zrobić ręcznie, ale wygodniej jest je mieć zebrane w jednej klasie. Możemy ją teraz przetestować:

```
vocab = Vocabulary(["a", "b", "c"])
assert vocab.size == 3, "w słowniku są 3 słowa"
assert vocab.get_id("b") == 1, "b powinno mieć identyfikator 1"
assert vocab.one_hot_encode("b") == [0, 1, 0]
```

```

assert vocab.get_id("z") is None, "w słowniku nie ma z"
assert vocab.get_word(2) == "c", "identyfikator 2 powinien oznaczać c"
vocab.add("z")
assert vocab.size == 4, "teraz w słowniku są 4 słowa"
assert vocab.get_id("z") == 3, "teraz z powinno mieć identyfikator 3"
assert vocab.one_hot_encode("z") == [0, 0, 0, 1]

```

Przydałaby się również pomocnicza funkcja do zapisywania i odczytywania słownika, podobnie jak robiliśmy w przypadku modeli uczenia głębokiego:

```

import json

def save_vocab(vocab: Vocabulary, filename: str) -> None:
    with open(filename, 'w') as f:
        json.dump(vocab.w2i, f) # potrzebujemy zapisać jedynie w2i

def load_vocab(filename: str) -> Vocabulary:
    vocab = Vocabulary()
    with open(filename) as f:
        # odczytaj w2i i na jego podstawie wygeneruj i2w
        vocab.w2i = json.load(f)
        vocab.i2w = {id: word for word, id in vocab.w2i.items()}
    return vocab

```

Będziemy korzystać z modelu wektorów słów, nazywanego *skip-gram*, który otrzymuje na wejściu słowo, a następnie określa prawdopodobieństwo, z jakim inne słowa mogą znaleźć się w jego sąsiedztwie. Dostarczymy mu pary treningowe (*word*, *nearby_word*) i spróbujemy zminimalizować funkcję straty *SoftmaxCrossEntropy*.



Inny popularny model, tzw. CBOW (ang. *continuous bag-of-words*), na podstawie sąsiadujących słów przewiduje konkretne słowo.

Zajmiemy się teraz zaprojektowaniem sieci neuronowej. W jej sercu będzie znajdować się warstwa *Embedding* przekształcająca identyfikator słowa na jego wektor. Możemy w tym celu użyć zwykłej tabeli słownikowej.

Następnie prześlemy wektor słów do warstwy *Linear*, która ma tyle samo wyjść, ile jest słów w słowniku. Podobnie jak wcześniej, użyjemy funkcji *softmax* do przekonwertowania wartości wyjściowych na prawdopodobieństwa wystąpień słów sąsiadujących. Używając metody gradientu, będziemy aktualizować wektory w tabeli słownikowej. Po wytrenowaniu modelu tabela ta będzie zawierała odpowiednie wektory słów.

Na początek stwórzmy warstwę *Embedding*. Możemy jej użyć również do obiektów innych niż słowa, więc napiszemy ogólną klasę, do której później dopiszemy podklasę *TextEmbedding* przeznaczoną jedynie dla wektorów słów.

W konstruktorze klasy określimy liczbę i wymiary wektorów, które mają zostać utworzone (początkowo będą to losowe wektory o rozkładzie normalnym).

```

from typing import Iterable
from scratch.deep_learning import Layer, Tensor, random_tensor, zeros_like

class Embedding(Layer):
    def __init__(self, num_embeddings: int, embedding_dim: int) -> None:
        self.num_embeddings = num_embeddings

```

```

self.embedding_dim = embedding_dim

# jeden wektor o rozmiarze size embedding_dim dla każdego przekształcenia
self.embeddings = random_tensor(num_embeddings, embedding_dim)
self.grad = zeros_like(self.embeddings)

# zapisz ostatni input id
self.last_input_id = None

```

W naszym przypadku warstwa ta będzie przetwarzała jedno słowo w danym momencie. Istnieją modele, w których można przetwarzać sekwencje słów i tworzyć z nich sekwencje wektorów (na przykład w opisanym wcześniej modelu CBOW), jednak dla uproszczenia pozostaniemy przy przetwarzaniu pojedynczych słów.

```

def forward(self, input_id: int) -> Tensor:
    """Wybiera wektor odpowiadający wejściowemu identyfikatorowi"""
    self.input_id = input_id # zapamiętaj, aby użyć przy propagacji wstecznej

    return self.embeddings[input_id]

```

Podczas propagacji wstecznej będziemy potrzebować gradientu dla wybranego wektora, więc skonstruujemy odpowiedni gradient dla `self.embeddings`, który wynosi zero dla każdego wektora z wyjątkiem wybranego:

```

def backward(self, gradient: Tensor) -> None:
    # korzystamy z gradientu z poprzedniego przetwarzania
    # jest to znacznie szybsze niż tworzenie zerowego tensora za każdym razem.
    if self.last_input_id is not None:
        zero_row = [0 for _ in range(self.embedding_dim)]
        self.grad[self.last_input_id] = zero_row

    self.last_input_id = self.input_id
    self.grad[self.input_id] = gradient

```

Musimy jeszcze nadpisać odpowiednie metody dla parametrów i gradientów:

```

def params(self) -> Iterable[Tensor]:
    return [self.embeddings]

def grads(self) -> Iterable[Tensor]:
    return [self.grad]

```

Jak wspomnieliśmy wcześniej, napiszemy jeszcze podklasę obsługującą tylko wektory słów. W tym przypadku liczba wektorów jest określona przez słownik, więc możemy wykorzystać go jako parametr wejściowy:

```

class TextEmbedding(Embedding):
    def __init__(self, vocab: Vocabulary, embedding_dim: int) -> None:
        # wywołaj konstruktor klasy Embedding
        super().__init__(vocab.size, embedding_dim)

        # umieść słownik w obiekcie klasy
        self.vocab = vocab

```

Pozostałe funkcje klasy `Embedding` będą działały bez zmian, ale dodamy jeszcze kilka nowych, specyficznych dla pracy z tekstem. Chcielibyśmy na przykład mieć możliwość uzyskiwania wektora dla konkretnego słowa (to nie jest część interfejsu `Layer`, ale zawsze możemy dodawać nowe funkcje do konkretnych warstw według naszych potrzeb).

```
def __getitem__(self, word: str) -> Tensor:
    word_id = self.vocab.get_id(word)
    if word_id is not None:
        return self.embeddings[word_id]
    else:
        return None
```

Dzięki tej funkcji specjalnej będziemy mogli uzyskiwać wektory przy użyciu indeksów:

```
word_vector = embedding["black"]
```

Chcielibyśmy mieć również możliwość uzyskiwania informacji na temat słowa będącego najbliższym danemu wyrazu:

```
def closest(self, word: str, n: int = 5) -> List[Tuple[float, str]]:
    """Zwraca n najbliższych słów na podstawie podobieństwa kosinusowego"""
    vector = self[word]

    # wyznacz pary (similarity, other_word) i posortuj według największego podobieństwa
    scores = [(cosine_similarity(vector, self.embeddings[i]), other_word)
               for other_word, i in self.vocab.w2i.items()]
    scores.sort(reverse=True)

    return scores[:n]
```

Nasza warstwa Embedding zwraca wektory, które możemy przekazać do warstwy Linear.

Teraz możemy już przygotować dane treningowe. Dla każdego słowa wejściowego wybierzemy jako wynik dwa słowa po jego lewej i dwa po jego prawej stronie.

Zacznijmy od zamiany wszystkich liter na małe i rozbicia zdań na słowa:

```
import re

# To wyrażenie regularne nie jest szczególnie zaawansowane, ale w naszym przypadku wystarczy.
tokenized_sentences = [re.findall("[a-z]+|[.]", sentence.lower())
                        for sentence in sentences]
```

a następnie zbudujemy słownik:

```
# Tworzenie słownika (czyli mapowania słowo -> identyfikator słowa) na podstawie tekstu.
vocab = Vocabulary(word

                    for sentence_words in tokenized_sentences
                    for word in sentence_words)
```

Możemy teraz wygenerować dane treningowe:

```
from scratch.deep_learning import Tensor, one_hot_encode

inputs: List[int] = []
targets: List[Tensor] = []

for sentence in tokenized_sentences:
    for i, word in enumerate(sentence): # dla każdego słowa
        for j in [i - 2, i - 1, i + 1, i + 2]: # weź najbliższe otoczenie,
            if 0 <= j < len(sentence): # które znajduje się w tekście,
                nearby_word = sentence[j] # i pobierz z niego słowa.

            # dodaje input, czyli identyfikator word_id pierwotnego słowa.
            inputs.append(vocab.get_id(word))

            # dodaje target, czyli identyfikatory najbliższych słów.
            targets.append(vocab.one_hot_encode(nearby_word))
```

Używając stworzonych przez nas narzędzi, możemy zbudować model:

```
from scratch.deep_learning import Sequential, Linear

random.seed(0)
EMBEDDING_DIM = 5 # wydaje się być dobrą wielkością

# tworzymy warstwę embedding osobno.
embedding = TextEmbedding(vocab=vocab, embedding_dim=EMBEDDING_DIM)

model = Sequential([
    # Mając słowo na wejściu (jako wektor identyfikatorów word_ids), dołącz jego wektor.
    embedding,
    # użyj warstwy linear do obliczenia wartości dla najbliższych słów.
    Linear(input_dim=EMBEDDING_DIM, output_dim=vocab.size)
])
```

Do wytrenowania modelu możemy użyć narzędzi napisanych w rozdziale 19.:

```
from scratch.deep_learning import SoftmaxCrossEntropy, Momentum, GradientDescent

loss = SoftmaxCrossEntropy()
optimizer = GradientDescent(learning_rate=0.01)

for epoch in range(100):
    epoch_loss = 0.0
    for input, target in zip(inputs, targets):
        predicted = model.forward(input)
        epoch_loss += loss.loss(predicted, target)
        gradient = loss.gradient(predicted, target)
        model.backward(gradient)
        optimizer.step(model)
    print(epoch, epoch_loss) # wyświetl wartość straty
    print(embedding.closest("black")) # oraz kilka najbliższych słów
    print(embedding.closest("slow")) # aby było widać
    print(embedding.closest("car")) # jak przebiega trenowanie modelu.
```

W miarę trenowania modelu możesz zobaczyć, że kolory coraz bardziej zbliżają się do siebie, podobnie przymiotniki i rzeczowniki.

Gdy model jest już gotowy, warto popatrzeć na najbliższe sobie słowa:

```
pairs = [(cosine_similarity(embedding[w1], embedding[w2]), w1, w2)
          for w1 in vocab.w2i
          for w2 in vocab.w2i
          if w1 < w2]
pairs.sort(reverse=True)
print(pairs[:5])
```

U mnie wyniki były takie:

```
[(0.9980283554864815, 'boat', 'car'),
 (0.9975147744587706, 'bed', 'cat'),
 (0.9953153441218054, 'seems', 'was'),
 (0.9927107440377975, 'extremely', 'quite'),
 (0.9836183658415987, 'bed', 'car')]
```

Możemy także wyznaczyć pierwsze dwie główne składowe i przedstawić je na wykresie:

```
from scratch.working_with_data import pca, transform
import matplotlib.pyplot as plt

# Wyznacz pierwsze dwie główne składowe i przetransformuj wektory słów.
components = pca(embedding.embeddings, 2)
```

```
transformed = transform(embedding.embeddings, components)

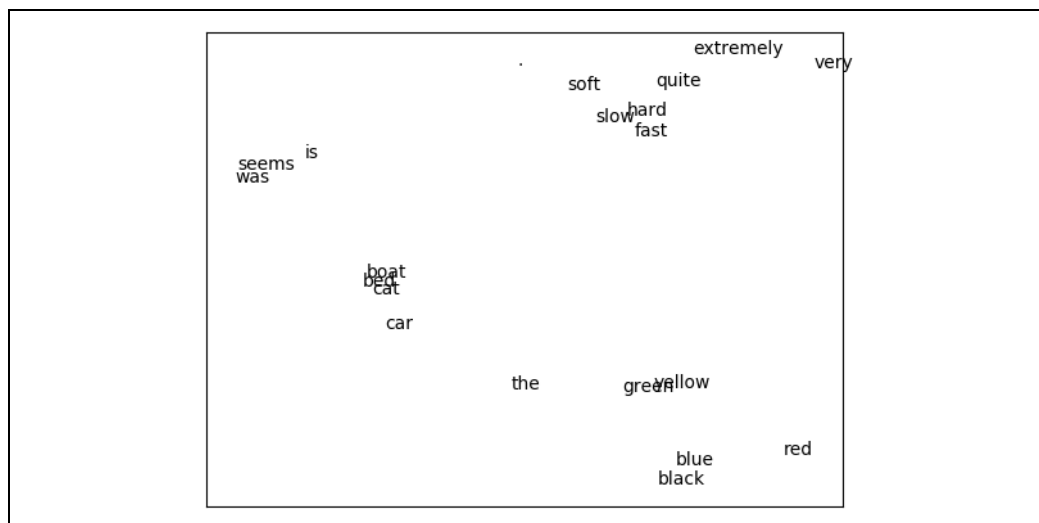
# Narysuj punkty na wykresie i pokoloruj je na bialo, aby byly niewidoczne.
fig, ax = plt.subplots()
ax.scatter(*zip(*transformed), marker='.', color='w')

# Dodaj opis do kazdego punktu.
for word, idx in vocab.w2i.items():
    ax.annotate(word, transformed[idx])

# Ukryj osie.
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

plt.show()
```

Widać, że podobne słowa faktycznie znajdują się blisko siebie (zobacz rysunek 21.3).



Rysunek 21.3. Wektory słów

Jeżeli jesteś zainteresowany wytrenowaniem modelu CBOW, to wiedz, że nie jest to takie trudne. Na początek należy zmodyfikować nieco warstwę Embedding, tak by na wejściu przyjmowała listę identyfikatorów i zwracała listę wektorów. Następnie należy stworzyć nową warstwę (Sum?), która pobiera listę wektorów i zwraca ich sumę.

Każde słowo reprezentuje przykład treningowy, gdzie na wejściu mamy identyfikatory otaczających słów, a wartością wyjściową jest identyfikator samego słowa.

Zmodyfikowana warstwa Embedding przetwarza otaczające słowa na listę wektorów, nowa warstwa Sum przekształca listę w pojedynczy wektor, a warstwa Linear wylicza wyniki, które mogą być przekształcone przy użyciu funkcji softmax, aby uzyskać rozkład oznaczający „najbardziej prawdopodobne słowa w tym kontekście”.

Wydaje mi się, że model CBOW jest trudniejszy do trenowania niż *skip-gram*, ale zachęcam, byś spróbował.

Rekurencyjne sieci neuronowe

Wektory słów, które tworzyliśmy w poprzednim podrozdziale, często są używane na wejściu sieci neuronowych. Jednym z problemów, jakie pojawiają się w tej sytuacji, jest to, że zdania mają różną liczbę słów. Zdanie składające się z trzech słów można przedstawić w formie tensora [3, embedding_dim], a zdanie zawierające dziesięć słów jako tensor [10, embedding_dim]. Aby przekazać takie zdania na przykład do warstwy Linear, musimy zrobić coś z tą zmienną.

Można na przykład wykorzystać warstwę Sum (lub jej wariant, używający średniej), jednak kolejność słów w zdaniu ma zazwyczaj duże znaczenie. Prostym przykładem mogą być zdania „Dzieci zjadają kurczaki” i „Kurczaki zjadają dzieci”.

Innym sposobem jest wykorzystanie **rekurencyjnych sieci neuronowych** (RNN — ang. *recurrent neural network*), które przechowują **ukryte stany** pomiędzy kolejnymi przetwarzanymi wartościami. W najprostszym przypadku każda wartość wejściowa jest przetwarzana razem z ukrytym stanem, aby wygenerować wartość wyjściową, która jest następnie używana jako nowy ukryty stan. Pozwala to sieciom „zapamiętywać” dotychczas przetworzone wartości i generować wynik, który zależy od wszystkich wartości na wejściu oraz od ich kolejności.

Stworzymy najprostszą możliwą warstwę RNN, która przyjmuje pojedyncze wartości na wejściu (na przykład pojedyncze słowa ze zdania lub pojedyncze znaki ze słowa) i przechowuje ich ukryty stan pomiędzy wywołaniami.

Jak pewnie pamiętasz, nasza warstwa Linear wykorzystuje pewne wagi w oraz wartość progową (bias) b. Na wejściu przyjmuje wektor i generuje wektor na wyjściu, używając logiki:

```
output[o] = dot(w[o], input) + b[o]
```

Ponieważ będziemy wykorzystywać ukryty stan, będą potrzebne *dwa* zestawy wag — jeden dla wartości wejściowych i jeden dla poprzedniego ukrytego stanu:

```
output[o] = dot(w[o], input) + dot(u[o], hidden) + b[o]
```

Następnie użyjemy wektora wyjściowego jako nowej wartości *ukrytej*. Nie jest to duża zmiana, ale pozwoli ona naszej sieci robić niezwykłe rzeczy.

```
from scratch.deep_learning import tensor_apply, tanh
```

```
class SimpleRnn(Layer):
    """Praktycznie najprostsza warstwa rekurencyjna."""
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        self.w = random_tensor(hidden_dim, input_dim, init='xavier')
        self.u = random_tensor(hidden_dim, hidden_dim, init='xavier')
        self.b = random_tensor(hidden_dim)

        self.reset_hidden_state()

    def reset_hidden_state(self) -> None:
        self.hidden = [0 for _ in range(self.hidden_dim)]
```

Jak widać, na początku przypisujemy stanowi ukrytemu wektor zerowy. Dodaliśmy też funkcję, którą można zresetować stan ukryty.

Funkcja forward jest w miarę prosta, o ile pamiętasz i rozumiesz, jak działa warstwa Linear:

```
def forward(self, input: Tensor) -> Tensor:
    self.input = input          # zachowaj zarówno wartość wejściową, jak i poprzedni
    self.prev_hidden = self.hidden # stan ukryty, aby użyć ich w propagacji wstecznej.

    a = [(dot(self.w[h], input) +          # wagi wejściowe
           dot(self.u[h], self.hidden) +   # wagi stanu ukrytego
           self.b[h])                     # wartość progowa
          for h in range(self.hidden_dim)]

    self.hidden = tensor_apply(tanh, a)    # Zastosuj tanh jako funkcję aktywacji
    return self.hidden                    # i zwróć wynik.
```

Funkcja backward jest podobna do tej w warstwie Linear, ale dodatkowo wylicza gradienty dla wag u:

```
def backward(self, gradient: Tensor):
    # propagacja wsteczna przez funkcję tanh
    a_grad = [gradient[h] * (1 - self.hidden[h] ** 2)
               for h in range(self.hidden_dim)]

    # b ma ten sam gradient co a
    self.b_grad = a_grad

    # Każda wartość w[h][i] jest mnożona przez input[i] i dodawana do a[h],
    # więc każdy w_grad[h][i] = a_grad[h] * input[i]
    self.w_grad = [[a_grad[h] * self.input[i]
                     for i in range(self.input_dim)]
                    for h in range(self.hidden_dim)]

    # Każda wartość u[h][h2] jest mnożona przez hidden[h2] i dodawana do a[h],
    # więc każdy u_grad[h][h2] = a_grad[h] * prev_hidden[h2]
    self.u_grad = [[a_grad[h] * self.prev_hidden[h2]
                     for h2 in range(self.hidden_dim)]
                    for h in range(self.hidden_dim)]

    # Każda wartość input[i] jest mnożona przez każdą wartość w[h][i] i dodawana do a[h],
    # więc każdy input_grad[i] = sum(a_grad[h] * w[h][i] for h in ...)
    return [sum(a_grad[h] * self.w[h][i] for h in range(self.hidden_dim))
            for i in range(self.input_dim)]
```

Na koniec musimy nadpisać metody params i grads:

```
def params(self) -> Iterable[Tensor]:
    return [self.w, self.u, self.b]

def grads(self) -> Iterable[Tensor]:
    return [self.w_grad, self.u_grad, self.b_grad]
```



Ten prosty przykład rekurencyjnej sieci neuronowej jest tak uproszczony, że prawdopodobnie nie będzie nadawał się do praktycznych zastosowań.

Nasza klasa SimpleRnn ma kilka niepożądanych cech. Jedną z nich jest to, że cały ukryty stan jest wykorzystywany do przeliczenia wartości wejściowej przy każdym wywołaniu. Ponadto przy każdym wywołaniu cały stan ukryty jest nadpisywany. Te dwie cechy powodują, że trenowanie modelu może być trudne, w szczególności przy długich zależnościach.

Z tego względu tego typu sieci rekurencyjne nie są używane. Zamiast nich wykorzystuje się bardziej skomplikowane warianty, takie jak LSTM (ang. — *long short-term memory*) lub GRU (ang. — *gated*

recurrent unit), które mają znacznie więcej parametrów i korzystają z parametryzowanych „bramek” umożliwiających wykorzystywanie i aktualizowanie tylko niektórych stanów w danym momencie.

Warianty te nie są szczególnie *skomplikowane*, jednak wymagają napisania znacznie więcej kodu, który w mojej opinii nie dałby nam wystarczająco dużo nowej wiedzy. Programy z tego rozdziału dostępne pod adresem <https://ftp.helion.pl/przyklady/dascp2.zip> zawierają implementację sieci LSTM. Zachęcam do ich przeanalizowania, ale jest to zadanie nieco żmudne, więc nie będziemy się tym teraz zajmować.

W naszej implementacji sieć przetwarza tylko jeden „krok” w danym momencie, a ponadto musimy ręcznie resetować stan ukryty. Bardziej praktycznie rozwiązania sieci rekurencyjnych mogą dopuszczać sekwencje wartości wejściowych, resetować stan ukryty na początku każdej sekwencji i generować sekwencje wartości wyjściowych. Nasz program mógłby oczywiście być poprawiony w ten sposób, ale to również wymagałoby znacznie więcej kodu, który wniósłby niewiele, jeżeli chodzi o zrozumienie działania sieci.

Przykład: używanie rekurencyjnej sieci neuronowej na poziomie pojedynczych znaków

Nowo zatrudnionemu dyrektorowi do spraw marki nie podoba się nazwa *DataSciencester* i sądzi, że lepsza nazwa mogłaby pomóc w rozwoju firmy. Prosi Cię, abyś przy użyciu narzędzi do analizy danych wygenerował kilka nowych nazw do rozważenia.

Jedno z ciekawych zastosowań rekurencyjnych sieci neuronowych polega na używaniu pojedynczych *znaków* (zamiast słów) na wejściu, uczeniu sieci subtelnych wzorców językowych na podstawie zbioru danych, a następnie generowaniu fikcyjnych słów w oparciu o te wzorce.

Możesz na przykład wytrenować sieć RNN na nazwach zespołów grających muzykę alternatywną, wygenerować nowe nazwy przy użyciu tego modelu, a następnie wybrać kilka najśmieszniejszych i opublikować na Twitterze.

Znając to zastosowanie sieci rekurencyjnych, decydujesz się wykorzystać je do znalezienia nowej nazwy dla serwisu *DataSciencester*.

Po krótkich poszukiwaniach znajdujesz stronę akceleratora startupów YCombinator, która publikuje listę 100 (tak naprawdę na liście jest ich 101) najlepszych startupów (<https://www.ycombinator.com/topcompanies/>). W źródle strony wszystkie nazwy firm są umieszczone w tagu `<b class="h4">`, dzięki czemu łatwo jest je pobrać:

```
from bs4 import BeautifulSoup
import requests

url = "https://www.ycombinator.com/topcompanies/"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')

# Pobieramy nazwy dwukrotnie, więc użyjemy zbioru, aby usunąć duplikaty.
companies = list({b.text
                   for b in soup("b")
                   if "h4" in b.get("class", ())})

assert len(companies) == 101
```

Jak zwykle, jeżeli strona będzie już nieaktualna lub niedostępna, kod nie zadziała prawidłowo. Możesz w takim wypadku użyć swoich nowo nabytych umiejętności analitycznych i naprawić kod albo pobrać listę firm pod adresem <https://ftp.helion.pl/przyklady/dascp2.zip>.

Nasz plan działania wygląda tak, że wytrenujemy model do przewidywania następnego znaku w nazwie na podstawie bieżącego znaku oraz ukrytego stanu zawierającego wszystkie znaki, które widzieliśmy do tej pory.

Jak zwykle będziemy przewidywać tak naprawdę rozkład prawdopodobieństwa znaków i trenować nasz model, aby minimalizował funkcję straty SoftmaxCrossEntropy.

Gdy tylko model będzie wytrenowany, możemy użyć go do generowania prawdopodobieństw, losowo wybierać znak na podstawie tych prawdopodobieństw i używać go jako kolejnej wartości wejściowej. Pozwoli nam to *generować* nazwy firm przy użyciu wyuczonych wag.

Na początek powinniśmy zbudować obiekt Vocabulary ze znaków w nazwach firm:

```
vocab = Vocabulary([c for company in companies for c in company])
```

Ponadto będziemy używać specjalnych tokenów, aby oznaczyć początek i koniec nazwy firmy. To pomoże modelowi nauczyć się, jakie znaki powinny znaleźć się na początku nazwy oraz kiedy nazwa jest już skończona.

Użyjemy w tym celu oznaczeń z wyrażeń regularnych, które na szczęście nie pojawiają się w nazwach firm:

```
START = "^"  
STOP = "$"  
  
# Należy je dodać do słownika.  
vocab.add(START)  
vocab.add(STOP)
```

W naszym modelu zakodujemy każdy znak, przepuścimy go przez dwie warstwy SimpleRnn, a następnie użyjemy warstwy Linear do wygenerowania wyniku dla każdego możliwego następnego znaku:

```
HIDDEN_DIM = 32 # Powinieneś poeksperymentować z różnymi rozmiarami!  
  
rnn1 = SimpleRnn(input_dim=vocab.size, hidden_dim=HIDDEN_DIM)  
rnn2 = SimpleRnn(input_dim=HIDDEN_DIM, hidden_dim=HIDDEN_DIM)  
linear = Linear(input_dim=HIDDEN_DIM, output_dim=vocab.size)  
  
model = Sequential([  
    rnn1,  
    rnn2,  
    linear  
)
```

Wyobraź sobie przez chwilę, że już wytrenowaliśmy ten model. Teraz przy użyciu funkcji `sample` ↪ `from` z podrozdziału „Modelowanie tematu” napiszemy funkcję, która używa naszego modelu do generowania nowych nazw firm:

```
from scratch.deep_learning import softmax  
  
def generate(seed: str = START, max_len: int = 50) -> str:  
    rnn1.reset_hidden_state() # Zresetuj obydwa ukryte stany  
    rnn2.reset_hidden_state()  
    output = [seed] # rozpocznij od podstawienia pod output określonej wartości seed
```

```

# Kontynuuj, aż trafisz na znak STOP lub do osiągnięcia maksymalnej długości
while output[-1] != STOP and len(output) < max_len:
    # Użyj ostatniego znaku na wejściu
    input = vocab.one_hot_encode(output[-1])

    # Wygeneruj wyniki, używając modelu
    predicted = model.forward(input)

    # Przekonwertuj je na prawdopodobieństwa i pobierz losowy char_id
    probabilities = softmax(predicted)
    next_char_id = sample_from(probabilities)

    # Dodaj odpowiedni znak do wartości wyjściowej
    output.append(vocab.get_word(next_char_id))

# usuń znaki START i END i zwróć wygenerowane słowo
return ''.join(output[1:-1])

```

Teraz jesteśmy gotowi, aby wytrenować naszą sieć rekurencyjną. Może to zająć chwilę.

```

loss = SoftmaxCrossEntropy()
optimizer = Momentum(learning_rate=0.01, momentum=0.9)

for epoch in range(300):
    random.shuffle(companies) # Za każdym przebiegiem zmieniamy kolejność.
    epoch_loss = 0 # śledzenie wartości straty.
    for company in tqdm.tqdm(companies):
        rnn1.reset_hidden_state() # Zresetuj obydwa ukryte stany.
        rnn2.reset_hidden_state()
        company = START + company + STOP # Add START and STOP characters.

        # Reszta działa jak typowa pętla treningowa, z tą różnicą, że wartości wejściowe oraz target są zakodowanym
        # poprzednim i następnym znakiem.
        for prev, next in zip(company, company[1:]):
            input = vocab.one_hot_encode(prev)
            target = vocab.one_hot_encode(next)
            predicted = model.forward(input)
            epoch_loss += loss.loss(predicted, target)
            gradient = loss.gradient(predicted, target)
            model.backward(gradient)
            optimizer.step(model)

    # Przy każdym przebiegu wyświetl wartość straty oraz wygeneruj nazwę.
    print(epoch, epoch_loss, generate())

    # Zmniejszenie wartości learning rate na ostatnie 100 przebiegów.
    # Nie ma dobrze określonego powodu, by tak robić, ale wygląda na to, że działa to dobrze.
    if epoch == 200:
        optimizer.lr *= 0.1

```

Po wytrenowaniu model generuje niektóre nazwy, które już znajdują się na liście (co nie jest niespodzianką ze względu na ograniczone możliwości modelu oraz niewielką ilość danych treningowych), nazwy tylko nieznacznie różniące się od treningowych (Scripe, Lionbare, Pozium), nazwy, które wyglądają naprawdę kreatywnie (Benuus, Clepto, Equite, Vivest) oraz kilka nazw, które należałoby odrzucić, chociaż w pewnym sensie wyglądają jak nazwy firm (SFitreasy, Sint ocanelp, GliyOx, Doorboronelhav).

Niestety, podobnie jak większość wyników generowanych przez sieci rekurencyjne używane na poziomie pojedynczych znaków, te nie są idealne i dyrektor do spraw marki rezygnuje ze swojego pomysłu.

Jeżeli zwiększyłbym liczbę ukrytych wymiarów do 64, dostałbym więcej nazw z oryginalnej listy. Jeżeli zmniejszyłbym ją do 8, dostałbym w większości bezsensowne wyniki. Słownik oraz końcowe wagi dla wszystkich tych rozmiarów modeli są dostępne pod adresem <https://ftp.helion.pl/przyklady/dascp2.zip>. Możesz użyć funkcji `load_weights` oraz `load_vocab`, by przetestować je samemu.

Jak wspomniałem wcześniej, kod dołączony do tego rozdziału zawiera również implementację modelu LSTM, który możesz wykorzystać zamiast `SimpleRnn` w swoim programie.

Dalsza eksploracja

- Natural Language Toolkit (<http://www.nltk.org/>) to popularna (i dość rozbudowana) biblioteka Pythona zawierająca narzędzia przeznaczone do przetwarzania języka naturalnego. Na jej temat napisano całą książkę, której treść umieszczono na stronie <http://www.nltk.org/book/>.
- Zamiast korzystać z naszego modelu napisanego od podstaw, lepiej jest używać biblioteki `gensim` (<http://radimrehurek.com/gensim/>), która jest przeznaczona do modelowania tematów.
- Inną popularną biblioteką Pythona do przetwarzania języka naturalnego jest `spaCy` (<https://spacy.io/>).
- Warto zapoznać się z artykułem *The Unreasonable Effectiveness of Recurrent Neural Networks*, który na swoim blogu zamieścił Andrej Karpathy (<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>).
- W mojej codziennej pracy zajmuję się tworzeniem biblioteki `AllenNLP` (<https://allennlp.org/>) do badań nad przetwarzaniem języka naturalnego (przynajmniej tym zajmowałem się w chwili oddawania książki do druku). Wykracza ona poza zakres tej książki, ale może być interesująca. Poza tym ma ciekawe interaktywne demo prezentujące najnowsze modele NLP.

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Nauka o danych: bazuj na solidnych podstawach!

Analityka danych jest uważana za wyjątkowo obiecującą dziedzinę wiedzy. Rozwija się błyskawicznie i znajduje coraz to nowsze zastosowania. Profesjonaliści biegli w eksploracji danych i wydobywaniu z nich pożytecznych informacji mogą liczyć na interesującą pracę i bardzo atrakcyjne warunki zatrudnienia. Jednak aby zostać analitykiem danych, trzeba znać matematykę i statystykę, a także nauczyć się programowania. Umiejętności w zakresie uczenia maszynowego i uczenia głębokiego również są ważne. W przypadku tak specyficznej dziedziny, jaką stanowi nauka o danych, szczególnie istotne jest zdobycie gruntownych podstaw i dogłębne ich zrozumienie.

W tym przewodniku opisano zagadnienia związane z podstawami nauki o danych. Wyjaśniono niezbędne elementy matematyki i statystyki. Przedstawiono także techniki budowy potrzebnych narzędzi i sposoby działania najistotniejszych algorytmów. Książka została skonstruowana tak, aby poszczególne implementacje były jak najbardziej przejrzyste i zrozumiałe. Zamieszczone tu przykłady napisano w Pythonie: jest to język dość łatwy do nauki, a pracę na danych ułatwia szereg przydatnych bibliotek. W drugim wydaniu znalazły się nowe tematy, takie jak uczenie głębokie, statystyka i przetwarzanie języka naturalnego, a także działania na ogromnych zbiorach danych. Zagadnienia te często pojawiają się w pracy współczesnego analityka danych.

W książce między innymi:

- elementy algebry liniowej, statystyki i rachunku prawdopodobieństwa
- zbieranie, oczyszczanie i eksploracja danych
- algorytmy modeli analizy danych
- podstawy uczenia maszynowego
- systemy rekomendacji i przetwarzanie języka naturalnego
- analiza sieci społecznościowych i algorytm MapReduce

Joel Grus jest inżynierem oprogramowania, analitykiem danych i autorem świetnie sprzedających się książek. Obecnie zajmuje się pracą badawczą w Allen Institute for Artificial Intelligence w Seattle. Angażuje się w rozwój projektu AllenNLP. Wcześniej był zatrudniony w firmie Google, pracował też w kilku start-upach.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!



ISBN 978-83-8322-131-1



Cena: 79,00 zł