O'REILLY®

Data science od podstaw

ANALIZA DANYCH W PYTHONIE

Helion

Joel Grus

Tytuł oryginału: Data Science from Scratch: First Principles with Python

Tłumaczenie: Konrad Matuk

ISBN: 978-83-283-4602-4

HELION SA
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/dascpo.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/dascpo
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 12.

Algorytm k najbliższych sąsiadów

Jeżeli chcesz zdenerwować swoich sąsiadów, to powiedz o nich prawdę.

— Pietro Aretino

Załóżmy, że chcesz przewidzieć, na kogo oddam swój głos w kolejnych wyborach prezydenckich. Jeżeli nic o mnie nie wiesz, ale dysponujesz jakimiś danymi na mój temat, to możesz przyjrzeć się planom wyborczym moich *sąsiadów*. Mieszkam na przedmieściach Seattle, a moi sąsiedzi niezmiennie głosują na kandydata Partii Demokratycznej, zatem obstawienie tego, że również i ja będę głosował na kandydata tej partii, wydaje się dość rozsądne.

Załóżmy, że posiadasz nie tylko dane dotyczące miejsca, w którym mieszkam. Dysponujesz danymi dotyczącymi mojego wieku, dochodu, liczby dzieci itd. Są to rzeczy, które do pewnego stopnia wpływają na moje zachowanie. Przyglądanie się preferencjom wyborczym moich sąsiadów, którzy są mi bliscy również w płaszczyźnie tych dodatkowych wymiarów, wydaje się być jeszcze lepszym pomysłem. W taki sposób działa **metoda klasyfikacji najbliższych sąsiadów**.

Model

Metoda najbliższych sąsiadów to najprostszy istniejący model predyktywny. Nie wymaga ona tworzenia zadanych założeń matematycznych ani stosowania ciężkiej artylerii. Wymaga ona jedynie:

- miary odległości;
- założenia, że punkty znajdujące się bliżej siebie są do siebie bardziej podobne.

Większość technik, z których będziemy korzystać w tej książce, analizuje cały zbiór danych w celu znalezienia w nim pewnych prawidłowości. Algorytm najbliższych sąsiadów dość często ignoruje wiele informacji, ponieważ przewidywania parametrów nowego elementu zbioru danych zależą tylko od parametrów punktów, które są położone najbliżej.

Ponadto metoda najbliższych sąsiadów nie pomoże Ci zrozumieć czynników wpływających na analizowany fenomen. Przewidywanie moich preferencji wyborczych na podstawie preferencji moich sąsiadów nie mówi zbyt wiele o przyczynach moich preferencji. Przyczyny mógłby określić alternatywny model przewidujący moje preferencje wyborcze na podstawie np. mojego dochodu i stanu cywilnego.

Ogólnie rzecz biorąc, dysponujemy elementami zbioru danych i odpowiadającymi im etykietami. Etykiety te mogą mieć postać np. wartości `True` lub `False`, które określają, czy dana wartość wejściowa spełnia warunek taki jak np. „jest spamem", „jest trucizną" lub „jest filmem wartym obejrzenia". Etykiety mogą być również kategoriami — takimi jak np. kategorie wiekowe filmów (G, PG, PG-13, R i NC-17) — nazwiskami kandydatów startujących w wyborach prezydenckich lub nazwami ulubionego języka programowania.

W naszym przypadku elementy zbioru danych będą wektorami, a więc możemy korzystać z funkcji `distance` utworzonej w rozdziale 4.

Załóżmy, że wybraliśmy wartość *k* równą np. 3 lub 5 i chcemy dokonać klasyfikacji nowych obserwacji — znaleźć *k* najbliższych elementów zbioru danych, oznaczonych etykietami, i na ich podstawie ustalić etykietę nowej obserwacji.

W tym celu potrzebujemy funkcji liczącej głosy. Oto przykładowa implementacja mechanizmu liczenia głosów:

```
def raw_majority_vote(labels):
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner
```

Niestety mechanizm taki nie robi niczego inteligentnego z węzłami. Załóżmy, że określamy kategorię wiekową filmów i pięciu najbliższym filmom nadano kategorie G, G, PG, PG i R.W związku z tym mamy dwa głosy na kategorię G i dwa głosy na kategorię PG. W takim przypadku możemy skorzystać z kilku rozwiązań:

- Wybrać zwycięzcę w sposób losowy.
- Określić wagi głosów na podstawie odległości i wybrać zwycięzcę na podstawie wagi.
- Zmniejszać wartość parametru *k* aż do jednoznacznego określenia zwycięzcy.

Zaimplementujemy trzecie rozwiązanie:

```
def majority_vote(labels):
    """Funkcja zakłada, że etykiety są ustawione w kolejności
       od najbliższej do najdalszej."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])

    if num_winners == 1:
        return winner                     # Ustalono jednoznacznie zwycięzcę. Zwróć go.
    else:
        return majority_vote(labels[:-1]) # Odrzuć najdalszą obserwację i spróbuj ponownie.
```

Koniec końców metoda ta zwróci zwycięzcę. W najgorszym wypadku zostaniemy z jedną etykietą, która zostanie uznana za zwycięzcę.

Dysponując tą funkcją, możemy z łatwością utworzyć klasyfikator:

```
def knn_classify(k, labeled_points, new_point):
    """Każdemu elementowi zbioru danych powinna być przypisana etykieta."""

    # Ustaw punkty oznaczone etykietami w kolejności od najbliższego do najdalszego.
    by_distance = sorted(labeled_points,
                         key=lambda (point, _): distance(point, new_point))

    # Ustal etykiety k najbliższych punktów.
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # Wybierz zwycięzcę na podstawie tych etykiet.
    return majority_vote(k_nearest_labels)
```

Przyjrzyjmy się działaniu tego mechanizmu w praktyce.

Przykład: ulubione języki

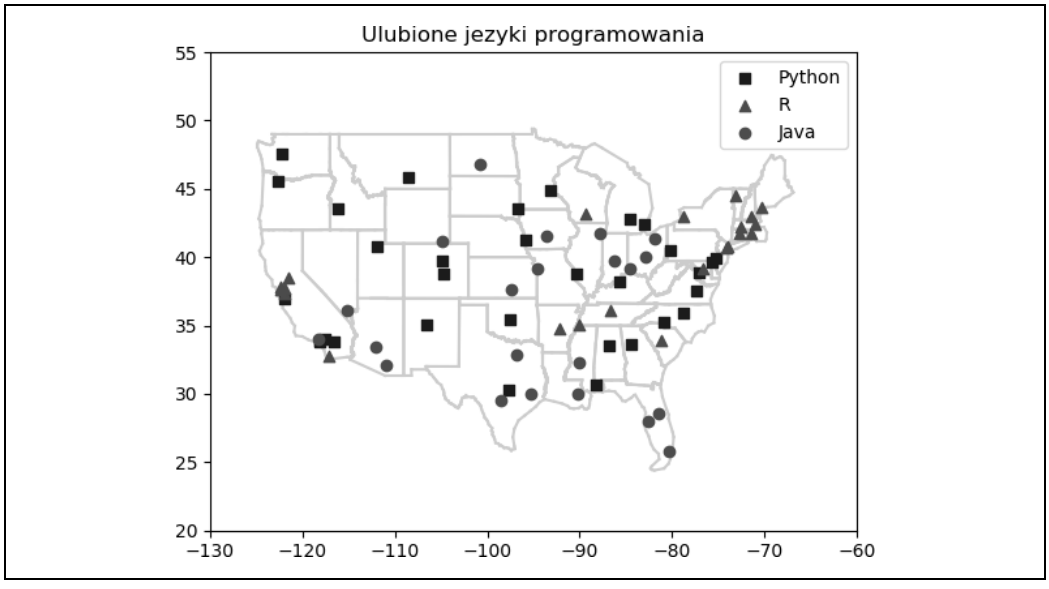
Właśnie udało Ci się uzyskać dostęp do pierwszej ankiety przeprowadzonej wśród użytkowników serwisu DataSciencester, która miała na celu ustalenie preferowanych przez nich języków programowania (użytkownicy biorący udział w ankiecie mieszkają tylko w większych miastach):

```
# Każdy element zbioru ma formę: ([długość geogr., szerokość geogr.], ulubiony_język).

cities = [([-122.3 , 47.53], "Python"), # Seattle
          ([ -96.85, 32.85], "Java"),   # Austin
          ([ -89.33, 43.13], "R"),      # Madison
          # I tak dalej…
]
```

Osoba zarządzająca działem aktywizacji społeczności chce dowiedzieć się, czy wyniki ankiety mogą zostać użyte w celu przewidzenia preferowanego języka programowania w miejscach, które nie były objęte ankietą.

Jak zwykle warto zacząć pracę od przedstawienia danych na wykresie (zobacz rysunek 12.1):

```
# Klucz definiuje język, a wartość jest parą (długość geogr., szerokość geogr.).
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

# Chcemy nadać każdemu językowy inny znak i kolor.
markers = { "Java" : "o", "Python" : "s", "R" : "^" }
colors  = { "Java" : "r", "Python" : "b", "R" : "g" }

for (longitude, latitude), language in cities:
    plots[language][0].append(longitude)
    plots[language][1].append(latitude)

# Utwórz wykres punktowy danych każdego języka.
for language, (x, y) in plots.iteritems():
    plt.scatter(x, y, color=colors[language], marker=markers[language],
                label=language, zorder=10)

plot_state_borders(plt)     # Załóżmy, że mamy funkcję generującą kontury stanów.

plt.legend(loc=0)           # Pozwól pakietowi matplotlib wybrać najlepsze miejsce na legendę.
plt.axis([-130,-60,20,55]) # Definicja osi.
plt.title("Ulubione jezyki programowania")
plt.show()
```



Rysunek 12.1. Ulubione języki programowania

W kodzie znajduje się odwołanie do niezdefiniowanej funkcji `plot_state_borders()`. Implementację tej funkcji znajdziesz w pliku *plot_state_borders.py* wchodzącym w skład archiwum z kodem znajdującym się na stronie *ftp://ftp.helion.pl/przyklady/dascpo.zip*, ale warto spróbować zaimplementować ją samodzielnie:

1. Poszukaj w internecie współrzędnych geograficznych granic pomiędzy poszczególnymi stanami.
2. Zamień dane, które uda Ci się znaleźć, na listę segmentów `[(dlug1, szer1), (dlug2, szer2)]`.
3. Nanieś segmenty na wykres za pomocą funkcji `plt.plot()`.

Wydaje się, że te same języki programowania są lubiane w zbliżonych do siebie miejscach, a więc model generujący przewidywania możemy oprzeć na algorytmie k najbliższych sąsiadów.

Na początek zobaczmy, co się stanie, jeżeli spróbujemy przewidzieć język preferowany w każdym mieście na podstawie etykiet sąsiadów, a nie etykiety wybranego miasta:

```
# Wypróbuj kilka wartości k.
for k in [1, 3, 5, 7]:
    num_correct = 0

    for city in cities:
        location, actual_language = city
        other_cities = [other_city
                        for other_city in cities
                        if other_city != city]

        predicted_language = knn_classify(k, other_cities, location)

        if predicted_language == actual_language:
            num_correct += 1

    print k, "sąsiad(ów):", num_correct, "poprawnie na", len(cities)
```

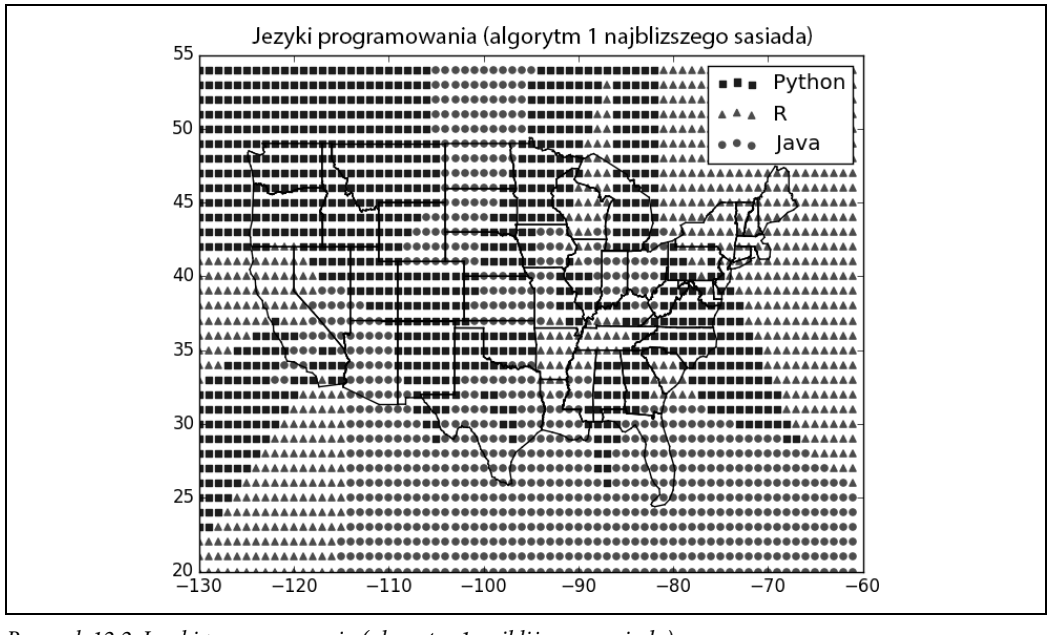
Wydaje się, że algorytm 3 najbliższych sąsiadów działa najlepiej — zwraca poprawne wyniki w około 59% przypadków:

```
1 sąsiad(ów): 40 poprawnie na 75
3 sąsiad(ów): 44 poprawnie na 75
5 sąsiad(ów): 41 poprawnie na 75
7 sąsiad(ów): 35 poprawnie na 75
```

Teraz możemy sprawdzić sposób, w jaki zostaną sklasyfikowane obszary w wyniku użycia każdej wartości parametru k. Zrobimy to, klasyfikując całą siatkę punktów, a następnie przedstawiając je na bazie wygenerowanego wcześniej wykresu miast:

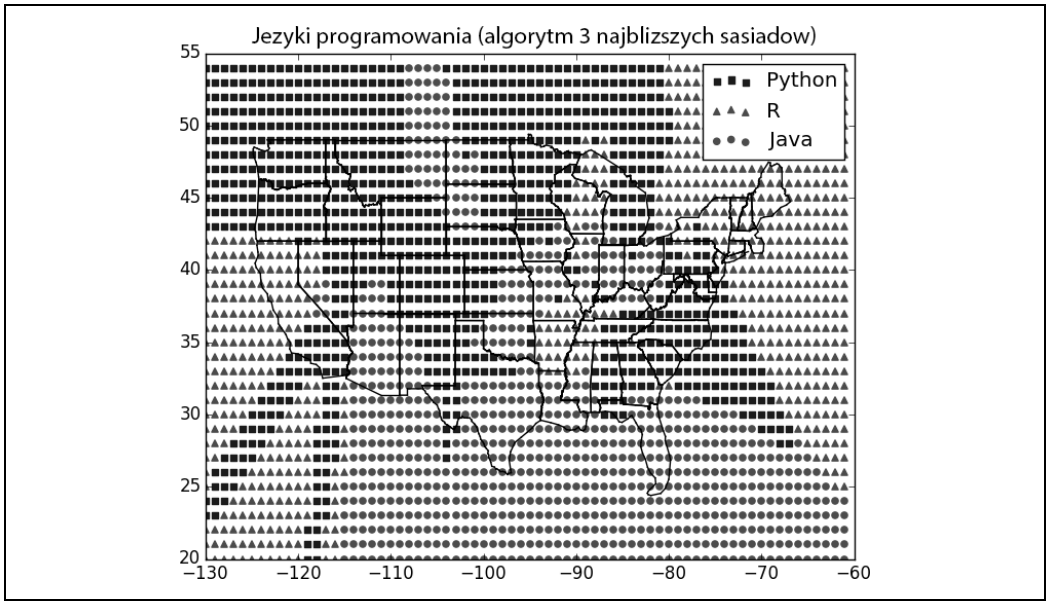
```
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

k = 1 # Lub k = 3, lub k = 5, lub…

for longitude in range(-130, -60):
    for latitude in range(20, 55):
        predicted_language = knn_classify(k, cities, [longitude, latitude])
        plots[predicted_language][0].append(longitude)
        plots[predicted_language][1].append(latitude)
```

Na rysunku 12.2 pokazano graficzną interpretację tego, co stanie się po przyjęciu parametru k równego 1 (przyglądamy się tylko jednemu najbliższemu sąsiadowi).



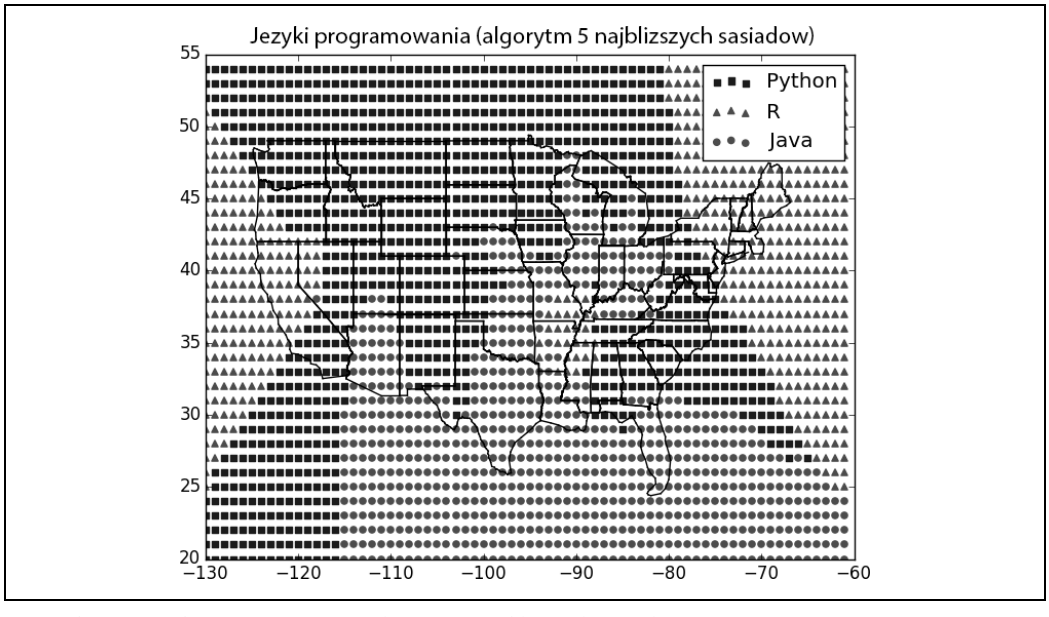
Rysunek 12.2. Języki programowania (algorytm 1 najbliższego sąsiada)

Widocznych jest wiele gwałtownych zmian języka — granice są ostre. Wzrost liczby analizowanych sąsiadów powoduje wygładzenie granic pomiędzy poszczególnymi językami (zobacz rysunek 12.3).



Rysunek 12.3. Języki programowania (algorytm 3 najbliższych sąsiadów)

Po zwiększeniu liczby analizowanych sąsiadów do pięciu uzyskujemy jeszcze bardziej gładkie granice (zobacz rysunek 12.4).



Rysunek 12.4. Języki programowania (algorytm 5 najbliższych sąsiadów)

W tym przypadku dane poszczególnych wymiarów można ze sobą porównywać, ale gdyby nie dało się ich porównywać, to należałoby wówczas skorzystać z technik skalowania opisanych w rozdziale 10., w podrozdziale „Przeskalowanie”.

Przekleństwo wymiarowości

Tzw. przekleństwo wymiarowości sprawia, że algorytm k najbliższych sąsiadów ma problemy z przetwarzaniem danych o większej liczbie wymiarów. Wynika to z *ogromu* przestrzeni wielowymiarowych. Punkty w przestrzeniach wielowymiarowych są zwykle od siebie bardzo oddalone. Można to sprawdzić poprzez generowanie losowych par punktów znajdujących się w d-wymiarowej przestrzeni, a następnie pomiar dzielących je odległości.

Generowanie wartości losowych nie powinno już sprawiać Ci problemów:

```
def random_point(dim):
    return [random.random() for _ in range(dim)]
```

To samo dotyczy tworzenia funkcji obliczającej odległość:

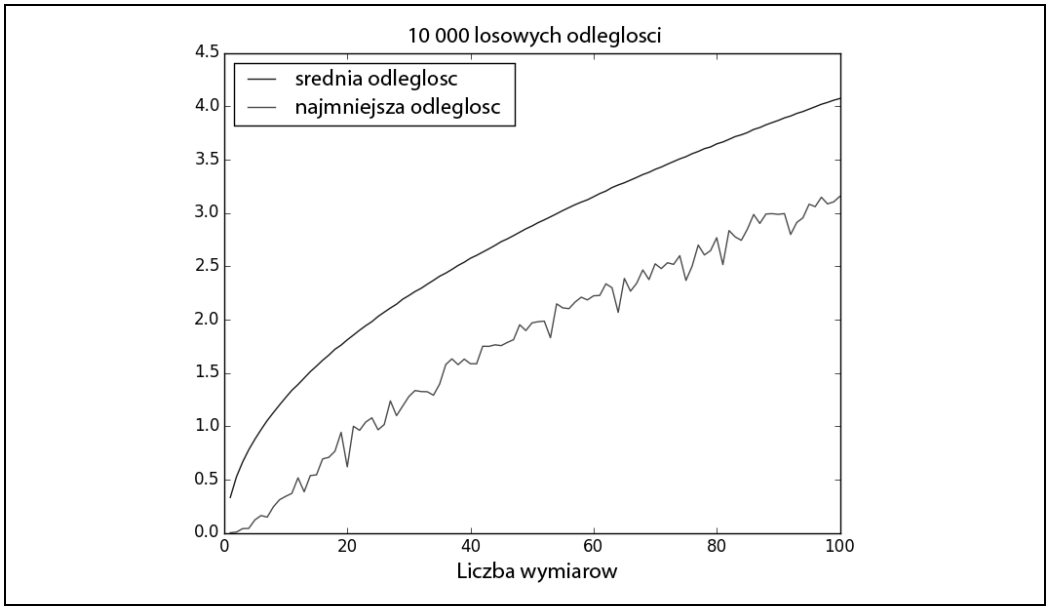
```
def random_distances(dim, num_pairs):
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

Dla każdej liczby wymiarów (od 1 do 100) będziemy obliczać 10 000 wartości odległości, które zostaną użyte w celu określenia średniej odległości pomiędzy punktami, a także minimalnej odległości pomiędzy punktami przy danej liczbie wymiarów (zobacz rysunek 12.5):

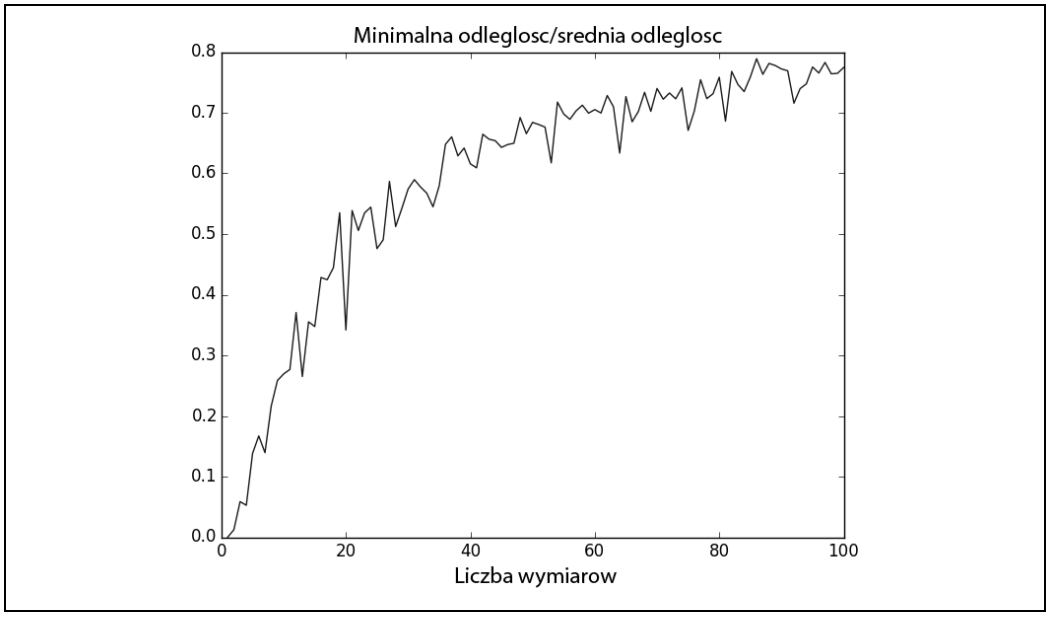
```
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in dimensions:
    distances = random_distances(dim, 10000) # 10,000 losowych par.
    avg_distances.append(mean(distances))    # Określ wartość średnią.
    min_distances.append(min(distances))     # Określ wartość najmniejszą.
```

Wzrost liczby wymiarów sprawia, że średnia odległość pomiędzy punktami wzrasta, ale największym problemem jest zmiana stosunku pomiędzy najmniejszą odległością i średnią odległością (zobacz rysunek 12.6):

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```



Rysunek 12.5. Przekleństwo wymiarowości

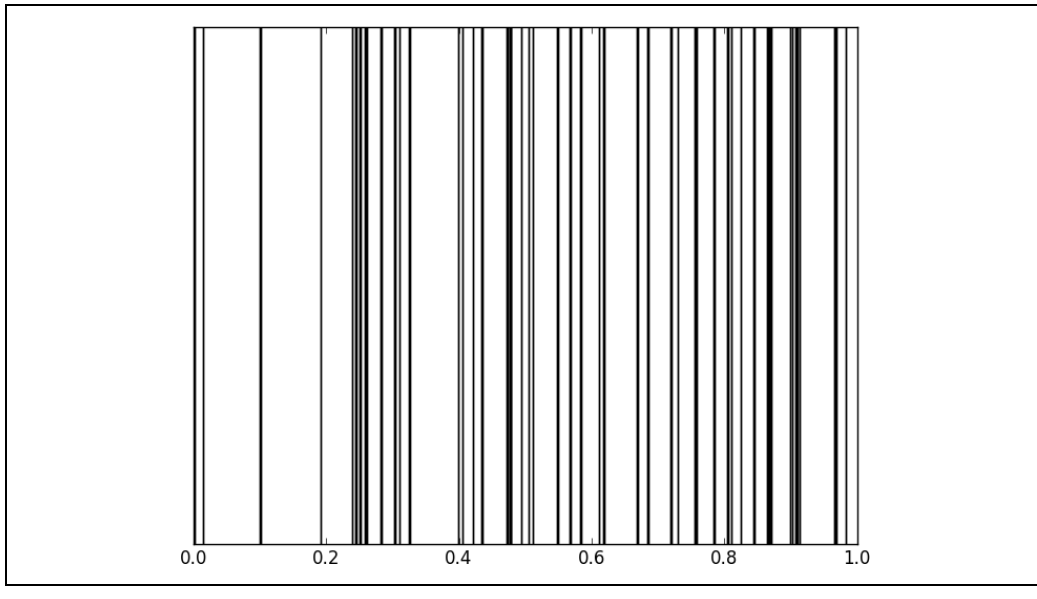


Rysunek 12.6. Kolejne skutki przekleństwa wymiarowości

W zbiorach danych o małej liczbie wymiarów najbliższe punkty są zwykle do siebie bardziej zbliżone. Dwa punkty znajdują się blisko siebie, jeżeli są bliskie na płaszczyźnie każdego wymiaru. Dodanie do zbioru każdego kolejnego wymiaru (nawet takiego, który zawiera tylko szum) może doprowadzić do zwiększenia odległości pomiędzy punktami. W przypadku dużej liczby wymiarów punkty reprezentujące elementy zbioru danych charakteryzują się zwykle większą średnią odległością od siebie, a więc bliskość dwóch punktów nie jest tak wyraźna jak w przypadku zbioru o mniejszej liczbie wymiarów, o ile dane nie zawierają wielu struktur, które sprawiają, że zbiór można analizować tak, jakby składał się z mniejszej liczby wymiarów.

Problem ten można również powiązać z gęstością przestrzeni o większej liczbie wymiarów.

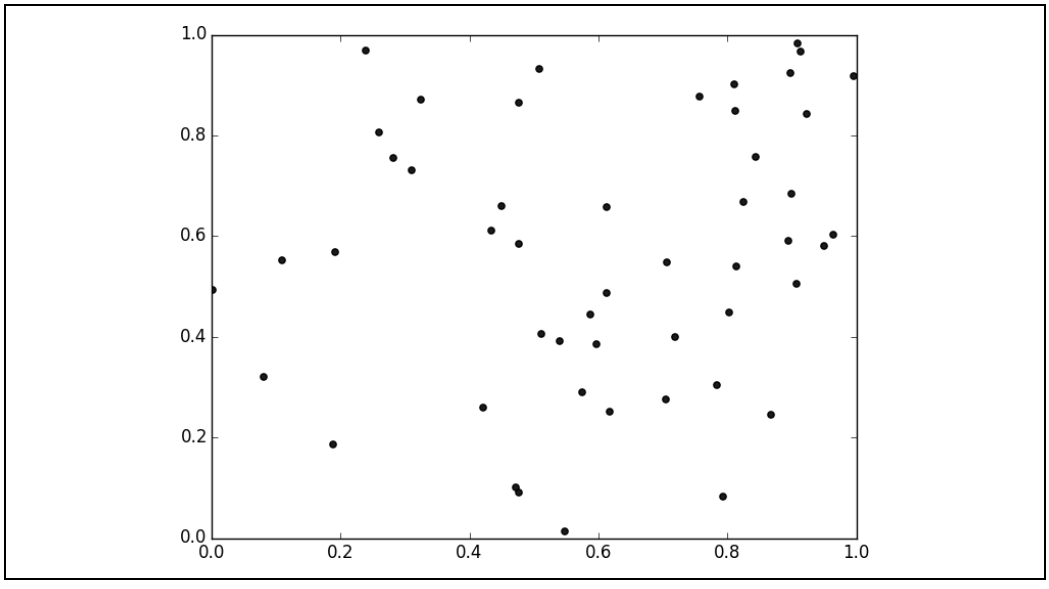
Jeżeli wybierzemy 50 losowych liczb z zakresu od 0 do 1, to prawdopodobnie uzyskamy dość dobrą próbkę interwału jednostkowego (zobacz rysunek 12.7).



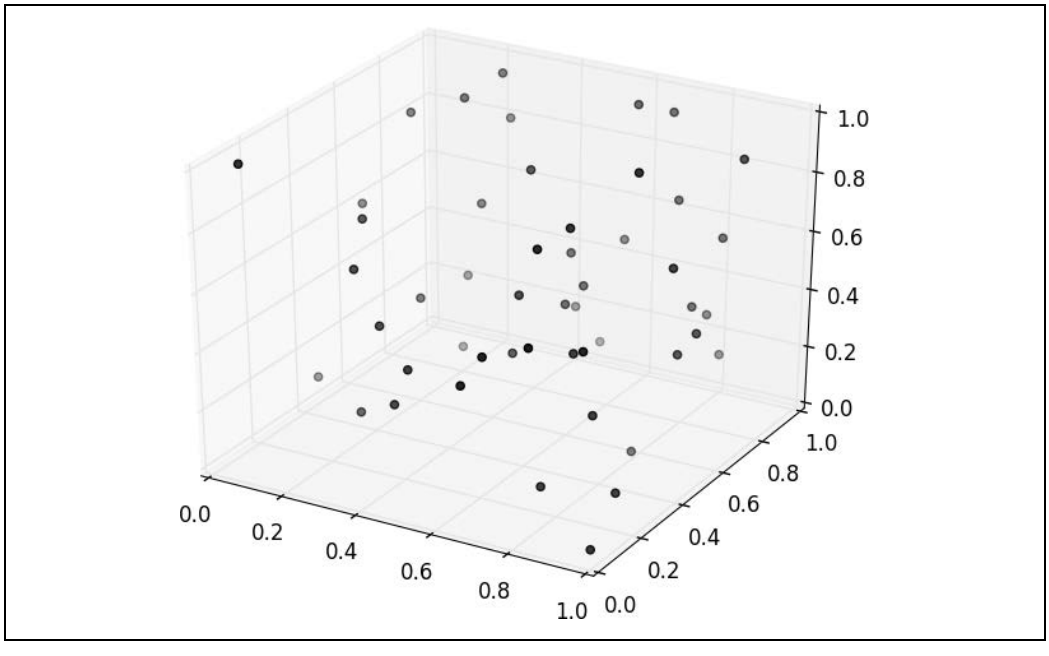
Rysunek 12.7. Pięćdziesiąt losowych wartości w przestrzeni jednowymiarowej

Jeżeli w przestrzeni dwuwymiarowej wybierzemy 50 losowo rozmieszczonych punktów, to będą one zajmowały mniejszy obszar przestrzeni (zobacz rysunek 12.8).

Pięćdziesiąt punktów umieszczonych w losowych miejscach przestrzeni trójwymiarowej będzie od siebie oddalonych jeszcze bardziej (zobacz rysunek 12.9).



Rysunek 12.8. Pięćdziesiąt punktów rozmieszczonych losowo w przestrzeni dwuwymiarowej



Rysunek 12.9. Pięćdziesiąt punktów rozmieszczonych losowo w przestrzeni trójwymiarowej

Pakiet `matplotlib` nie tworzy zbyt dobrych wizualizacji przestrzeni czterowymiarowych, a więc nie będę generował bardziej rozbudowanych wykresów, ale z trzech przedstawionych dotychczas wykresów wynika, że wzrost liczby wymiarów zwiększa ilość pustej przestrzeni pomiędzy wartościami. Wzrost liczby wymiarów bez wykładniczego przyrostu liczby punktów doprowadza do wzrostu pustych przestrzeni pomiędzy punktami, które mają być użyte do generowania przewidywań.

Jeżeli chcesz korzystać z algorytmu k najbliższych sąsiadów w przestrzeniach wielowymiarowych, to lepiej jest zacząć pracę z takimi danymi od zastosowania jednej z technik redukcji wymiarów.

Dalsza eksploracja

- Pakiet scikit-learn zawiera implementacje wielu modeli algorytmów najbliższych sąsiadów (*http://bit.ly/1ycP5rj*).

Skorowidz

N

O

P

R

S

Ś

T

U

W

Z

Notatki

O'REILLY®

Python. Wyciśniesz z danych każdą kroplę wiedzy!

Współczesne ogromne zbiory danych zawierają odpowiedzi na prawie każde pytanie. Równocześnie nauka o danych jest dziedziną, która cokolwiek onieśmiela. Znajduje się gdzieś pomiędzy subtelnymi umiejętnościami hakerskimi, twardą wiedzą z matematyki i ze statystyki a merytoryczną znajomością zagadnień z danej branży. Co więcej, dziedzina ta niezwykle dynamicznie się rozwija. Jednak trud włożony w naukę o danych niewątpliwie się opłaca: biegły analityk danych może liczyć na dobrze płatną, inspirującą i bardzo atrakcyjną pracę.

Dzięki tej książce opanujesz najważniejsze zagadnienia związane z matematyką i ze statystyką, będziesz także rozwijać umiejętności hakerskie. W ten sposób zyskasz podstawy pozwalające na rozpoczęcie przygody z analizą danych. Gruntownie zapoznasz się z potrzebnymi narzędziami i algorytmami. Pozwoli Ci to lepiej zrozumieć ich działanie. Poszczególne przykłady, którymi zilustrowano omawiane zagadnienia, są przejrzyste, dobrze opisane i zrozumiałe. Podczas lektury książki poznasz biblioteki, które umożliwią zaimplementowanie omówionych technik podczas analizy dużych zbiorów danych. Szybko się przekonasz, że aby zostać analitykiem danych, wystarczy odrobina ciekawości, sporo chęci, mnóstwo ciężkiej pracy i... ta książka.

Najważniejsze zagadnienia:

- Praktyczne wprowadzenie do Pythona
- Podstawy algebry liniowej, statystyki i rachunku prawdopodobieństwa w analizie danych
- Podstawy uczenia maszynowego
- Implementacje algorytmów modeli, w tym naiwny klasyfikator bayesowski, regresja liniowa, regresja logistyczna, drzewa decyzyjne, sieci neuronowe i grupowanie, MapReduce
- Systemy rekomendacji i mechanizmy przetwarzania języka naturalnego

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. Wcześniej był zatrudniony w firmie Google i kilku startupach. Mieszka w Seattle, gdzie regularnie uczestniczy w spotkaniach lokalnej społeczności analityków danych.

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 978-83-283-4602-4

INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 57,00 zł