OKIEM EKSPERTA

Matematyka w uczeniu maszynowym

Opanuj algebrę liniową, rachunek różniczkowy i całkowy oraz rachunek prawdopodobieństwa

Tivadar Danka

Helion

packt

Tytuł oryginału: Mathematics of Machine Learning: Master linear algebra, calculus, and probability for machine learning

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-289-3325-5

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

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

CZĘŚĆ 2. Rachunek różniczkowy i całkowy

CZĘŚĆ 4. Teoria prawdopodobieństwa

Dodatki

Wektory i przestrzenie wektorowe

Rozdział 1

Chciałbym zwrócić uwagę, że klasa abstrakcyjnych przestrzeni liniowych nie jest większa niż klasa przestrzeni, których elementami są tablice. Co zatem zyskujemy dzięki abstrakcji? Przede wszystkim swobodę używania pojedynczego symbolu dla tablicy. W ten sposób możemy traktować wektory jako podstawowe elementy konstrukcyjne bez konieczności myślenia o ich składowych. Podejście abstrakcyjne prowadzi do prostych i przejrzystych dowodów twierdzeń.

Peter D. Lax, rozdział 1. książki *Linear Algebra and its Applications*

Matematyka uczenia maszynowego opiera się na trzech filarach: *algebrze liniowej*, *rachunku różniczkowym* i *teorii prawdopodobieństwa*. Algebra liniowa opisuje, jak reprezentować dane i nimi operować, rachunek różniczkowy pomaga dopasowywać modele, natomiast teoria prawdopodobieństwa pomaga je interpretować.

Kolejne dziedziny bazują na wcześniejszych, dlatego należy zacząć od podstaw: *reprezentacji danych i operowania nimi*.

W przykładach w tym podrozdziale korzystam ze słynnego zbioru danych dotyczącego irysów (*https://en.wikipedia.org/wiki/Iris_flower_data_set*). Zawiera on pomiary długości i szerokości działek kielicha i płatków trzech gatunków irysów. Każdy punkt danych obejmuje więc cztery pomiary wraz z gatunkiem danego kwiatu: *Iris setosa* (kosaciec szczecinkowy), *Iris virginica* (kosaciec wirginijski) lub *Iris versicolor* (kosaciec różnobarwny). Działki kielicha to zwykle zielone, liściaste struktury u podstawy kwiatu, które chronią rozwijający się pąk, zanim się otworzy. Płatki to kolorowe, miękkie części kwiatu, które przyciągają zapylacze, takie jak owady czy ptaki.

Wspomniany zbiór danych można łatwo wczytać bezpośrednio z biblioteki *scikit-learn* (*https://scikit-learn.org/*), więc przyjrzyj mu się bliżej.

```
from sklearn.datasets import load_iris
data = load_iris()
X, y = data["data"], data["target"]
X[:10]
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
```

```
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1]])
```

Zanim przejdę do definicji matematycznych, należy ustalić wspólne słownictwo. Dane z pomiarów są przechowywane w formacie tabelarycznym. Wiersze reprezentują próbki, a kolumny reprezentują pomiary. Konkretny typ pomiaru często nazywany jest *cechą*. Za pomocą instrukcji `X.shape` można stwierdzić, że zbiór danych Iris zawiera 150 punktów danych i cztery cechy:

```
X.shape
(150, 4)
```

(Nie martw się, jeśli nie znasz NumPy. W odpowiednim czasie dowiesz się więcej o tej bibliotece. Na razie zapamiętaj, że instrukcja `shape` tablicy pozwala sprawdzić jej wymiary).

Dla danej próbki odpowiadający jej gatunek nazywany jest *etykietą*. W omawianym przykładzie etykiety to *Iris setosa*, *Iris virginica* lub *Iris versicolor*. Są one zakodowane za pomocą liczb 0, 1 i 2:

```
y
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2])
```

W ujęciu matematycznym zbiór danych Iris tworzy *macierz*, a poszczególne punkty danych tworzą *wektory*. Mówiąc prościej, macierze to *tabele*, a wektory to *krotki* (czyli skończone i uporządkowane sekwencje obiektów, na przykład `(1.297, -2.35, 32.3, 29.874)`). Jednak to uproszczone ujęcie nie pokazuje pełnego obrazu. Wektory i macierze mają cudowną strukturę geometryczną i algebraiczną, a zgłębienie teorii matematycznej dotyczącej tych koncepcji pozwala dostrzec wzorce kryjące się w danych.

Jak to się dzieje? Załóżmy, że oprócz zapisania punktów danych w zwięzłej postaci chcesz wykonywać na nich operacje, takie jak dodawanie i mnożenie przez skalar. Po co dodawać do siebie punkty danych? Oto prosty przykład: często korzystne jest, gdy cechy są wyrażone w tej samej skali. Jeśli dana cecha ma mniejszy zakres wartości niż pozostałe, będzie miała mniejszy wpływ na prognozy.

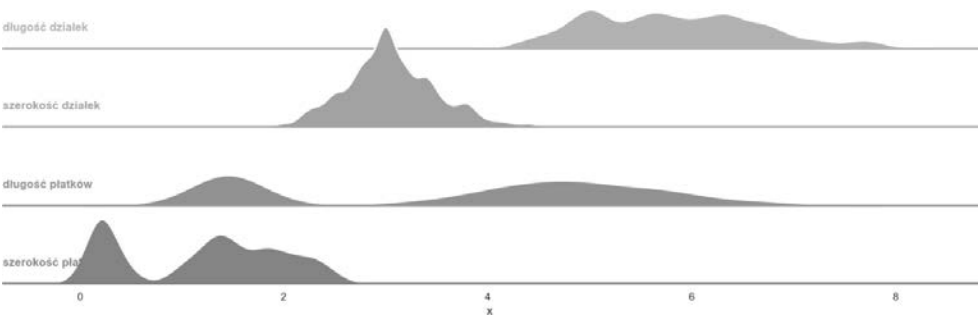
Pomyśl o tym w ten sposób: jeśli ktoś szepcze do Ciebie z sąsiedniego pokoju, podczas gdy tuż obok Twojego ucha grają głośniki, nie usłyszysz nic z tego, co mówi ta osoba. Cechy o dużej skali są jak głośna muzyka, a te o mniejszej skali przypominają szept. Możesz uzyskać znacznie więcej informacji z szeptu, ale najpierw musisz ściszyć muzykę.

Aby zobaczyć to zjawisko w praktyce, przyjrzyj się rozkładowi cech w omawianym zbiorze danych:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

sns.set_theme(style="white", rc={"axes.facecolor": (0, 0, 0, 0)})
```

```
# Przygotowywanie danych
x = X.ravel()
labels = ["Długość działek", "Szerokość działek", "Długość płatków",
          "Szerokość płatków"]
g = np.tile(labels, len(X))
df = pd.DataFrame(dict(x=x, g=g))

# Inicjalizowanie obiektu FacetGrid
pal = sns.cubehelix_palette(10, rot=-.25, light=.7)
g = sns.FacetGrid(df, row="g", hue="g", aspect=10, height=1.5, palette=pal)

# Generowanie wykresu gęstości
g.map(sns.kdeplot, "x", bw_adjust=.5, clip_on=False, fill=True, alpha=1,
↪linewidth=1.5)
g.map(sns.kdeplot, "x", clip_on=False, color="w", lw=2, bw_adjust=.5)

# Dodawanie linii odniesienia
g.refline(y=0, linewidth=2, linestyle="-", color=None, clip_on=False)

# Dodawanie etykiety do każdego wykresu
g.map(lambda x, color, label: plt.gca().text(0, .2, label, fontweight="bold",
      color=color,ha="left", va="center", transform=plt.gca().transAxes), "x")

# Dodawanie podwykresów i aspektów estetycznych
g.figure.subplots_adjust(hspace=-.25)
g.set_titles("")
g.set(yticks=[], ylabel="")
g.despine(bottom=True, left=True)

plt.show
```

Na rysunku 1.1 widać, że wartości niektórych cech mają szerszy zakres (na przykład długość działki kielicha), podczas gdy inne węższy (na przykład szerokość działki kielicha). W zastosowaniach praktycznych może to negatywnie wpływać na skuteczność predykcyjną algorytmów.



Rysunek 1.1. Surowe cechy ze zbioru danych Iris

Aby rozwiązać ten problem, można usunąć *średnią* i *odchylenie standardowe* ze zbioru danych. Jeśli zbiór danych składa się z wektorów $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., $\boldsymbol{x}_{150}$, można obliczyć ich średnią za pomocą następującego wzoru:

$$\boldsymbol{\mu} = \frac{1}{150}\sum_{i=1}^{150} x_i \in \mathbb{R}^4.$$

A oto wzór na odchylenie standardowe:

$$\sigma = \sqrt{\frac{1}{150}\sum_{i=1}^{150} (x_i - \mu)^2} \in \mathbb{R}^4,$$

gdzie odejmowanie i podnoszenie do kwadratu w $(\boldsymbol{x}_i - \boldsymbol{\mu})^2$ wykonywane jest po współrzędnych.

Składowe $\boldsymbol{\mu} = (\mu_1, \mu_2, \mu_3, \mu_4)$ i $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \sigma_3, \sigma_4)$ to średnie i wariancje poszczególnych cech. Przypominam, że zbiór danych Iris zawiera 150 próbek i 4 cechy na próbkę.

Innymi słowy, średnia opisuje przeciętną wartość próbek, a odchylenie standardowe reprezentuje średnią odległość od średniej. Im większe odchylenie standardowe, tym bardziej rozproszone są próbki.

Przeskalowany zbiór danych (rysunek 1.2) można opisać w następujący sposób:

$$\frac{x_1 - \mu}{\sigma}, \frac{x_2 - \mu}{\sigma}, \ldots, \frac{x_{150} - \mu}{\sigma},$$

gdzie zarówno odejmowanie, jak i dzielenie wykonywane są po współrzędnych.

Jeśli znasz Pythona i bibliotekę NumPy, to właśnie tak to się w nich odbywa. Nie martw się, jeżeli nie znasz tych narzędzi. Wszystko, co musisz wiedzieć na ich temat, wyjaśniam w następnym rozdziale, gdzie przedstawiam też przykładowy kod.

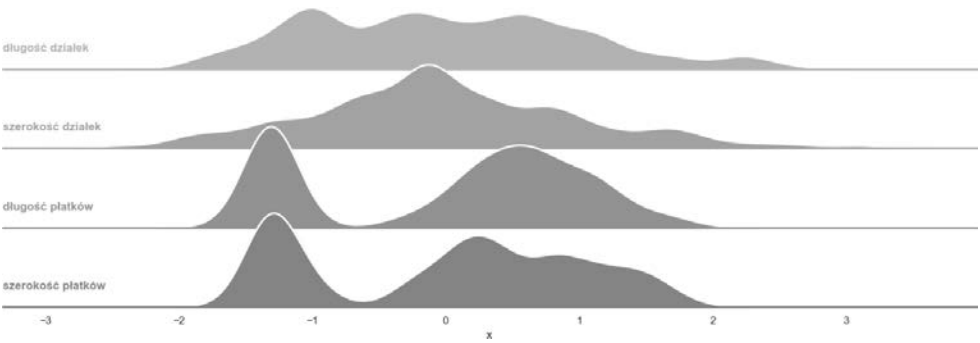
```
X_scaled = (X - X.mean(axis=0))/X.std(axis=0)
X_scaled[:10]
```

```
array([[-0.90068117,  1.01900435, -1.34022653, -1.3154443 ],
       [-1.14301691, -0.13197948, -1.34022653, -1.3154443 ],
       [-1.38535265,  0.32841405, -1.39706395, -1.3154443 ],
       [-1.50652052,  0.09821729, -1.2833891 , -1.3154443 ],
       [-1.02184904,  1.24920112, -1.34022653, -1.3154443 ],
       [-0.53717756,  1.93979142, -1.16971425, -1.05217993],
       [-1.50652052,  0.78880759, -1.34022653, -1.18381211],
       [-1.02184904,  0.78880759, -1.2833891 , -1.3154443 ],
       [-1.74885626, -0.36217625, -1.34022653, -1.3154443 ],
       [-1.14301691,  0.09821729, -1.-2833891 , -1.44707648]])
```

```
# Przygotowanie danych
x = X_scaled.ravel()
labels = ["Długość działek", "Szerokość działek", "Długość płatków",
          "Szerokość płatków"]
g = np.tile(labels, X_scaled.shape[0])
df = pd.DataFrame(dict(x=x, g=g))

# Inicjalizacja obiektu FacetGrid
pal = sns.cubehelix_palette(10, rot=-.25, light=.7)
grid = sns.FacetGrid(df, row="g", hue="g", aspect=10, height= 1.5, palette=pal)
```

```
# Generowanie wykresów gęstości
grid.map(sns.kdeplot, "x", bw_adjust=.5, clip_on=False, fill=True, alpha=1,
         linewidth=1.5)
grid.map(sns.kdeplot, "x", clip_on=False, color="w", lw=2, bw_adjust=.5)

# Dodawanie linii odniesienia
grid.refline(y=0, linewidth=2, linestyle="-", color=None, clip_on=False)

# Dodawanie etykiet do każdego wykresu
grid.map(lambda x, color, label: plt.gca().text(0, .2, label, fontweight="bold",
         color=color,ha="left", va="center", transform=plt.gca().transAxes), "x")

# Dostosowanie podwykresów i aspektów estetycznych
grid.figure.subplots_adjust(hspace=-.25)

grid.set_titles("")
grid.set(yticks=[], ylabel="")
grid.despine(bottom=True, left=True)

plt.show()
```

Jeśli porównasz zmodyfikowaną wersję z oryginalną, zauważysz, że wszystkie cechy mają teraz tę samą skalę (rysunek 1.2). Innymi słowy, zbiór danych został przekształcony na przydatniejszą postać. Na (bardzo) ogólnym poziomie efekt uczenia maszynowego to nic innego jak seria wyuczonych transformacji danych przekształcających surowe dane w formę umożliwiającą proste generowanie prognoz.



Rysunek 1.2. Przeskalowane cechy ze zbioru danych Iris

W ujęciu matematycznym operowanie danymi i modelowanie ich relacji z etykietami wywodzi się z koncepcji *przestrzeni wektorowych* i transformacji między nimi. Dlatego zacznę od podania precyzyjnej definicji przestrzeni wektorowych.

1.1. Czym jest przestrzeń wektorowa?

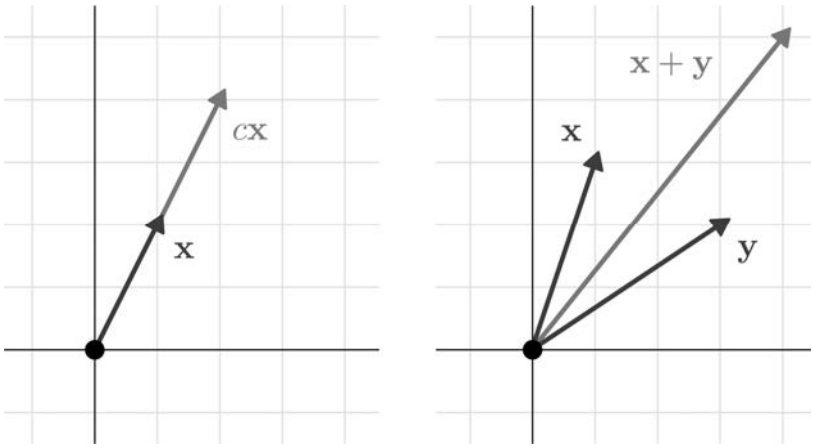
Reprezentowanie wielu pomiarów jako krotki $(x_1, x_2, \ldots, x_n)$ to naturalne rozwiązanie, które ma wiele zalet. Forma krotki sugeruje, że składowe powinny występować razem i mieć określoną kolejność. Zapewnia to jasny i zwięzły sposób przechowywania informacji.

Wiąże się to jednak z pewnym kosztem: wymaga to pracy z bardziej złożonymi obiektami. Jednak, mimo że trzeba wtedy operować na krotkach typu $(x_1, \ldots, x_n)$ zamiast na liczbach, istnieją pewne podobieństwa między wykonywanymi operacjami. Na przykład dowolne dwie krotki $\boldsymbol{x} = (x_1, \ldots, x_n)$ i $\boldsymbol{y} = (y_1, \ldots, y_n)$ można:

- dodawać: $\boldsymbol{x} + \boldsymbol{y} = (x_1 + y_1, \ldots, x_n + y_n)$;
- mnożyć przez skalar: jeśli $c \in \mathbb{R}$, to $c\boldsymbol{x} = (cx_1, \ldots, cx_n)$.

Jest to zbliżone do posługiwania się zwykłymi liczbami.

Te operacje mają również jednoznaczne interpretacje geometryczne. Dodawanie odpowiada *przesunięciu*, a mnożenie przez skalar to proste *rozciąganie* (lub kurczenie, jeśli $|c| < 1$). Ilustruje to rysunek 1.3.



Rysunek 1.3. Geometryczna interpretacja dodawania i mnożenia przez skalar

Jeśli chcesz korzystać z intuicyjnego ujęcia geometrycznego (co zdecydowanie warto robić), nie jest jasne, jak zdefiniować mnożenie wektorów. Mimo że definicja

$$\boldsymbol{xy} = (x_1y_1, \ldots, x_ny_n)$$

ma sens algebraiczny, to trudno zrozumieć jej znaczenie geometryczne.

Gdy myślisz o wektorach i przestrzeniach wektorowych, zapewne wyobrażasz sobie struktury matematyczne, które odpowiadają Twoim intuicyjnym założeniom i oczekiwaniom. Warto przekształcić te intuicje w formalną definicję.

Definicja 1.1.1. Przestrzenie wektorowe

Przestrzeń wektorowa to struktura matematyczna $(V, F, +, \cdot)$, gdzie:

a) V to zbiór wektorów;

b) F to ciało skalarów (najczęściej liczby rzeczywiste $\mathbb{R}$ lub liczby zespolone $\mathbb{C}$);

c) $+ : V \times V \to V$ to operacja dodawania spełniająca następujące właściwości:

- $x + y = y + x$ (przemienność),
- $x + (y + z) = (x + y) + z$ (łączność),
- istnieje element $0 \in V$ taki, że $x + 0 = x$ (istnienie wektora zerowego),
- dla każdego $x \in V$ istnieje $-x \in V$ taki, że $x + (-x) = 0$ (istnienie wektora przeciwnego)

dla wszystkich wektorów $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in V$;

d) $\cdot : F \times V \to V$ to operacja mnożenia przez skalar spełniająca następujące właściwości:

- $a(bx) = (ab)x$ (łączność),
- $a(x + y) = ax + ay$ (rozdzielność),
- $1x = x$

dla wszystkich skalarów $a, b \in F$ i wektorów $x, y \in V$.

Ta definicja zawiera wiele nowych pojęć, więc warto przyjrzeć się jej bliżej.

Traktowanie operacji takich jak dodawanie i mnożenie przez skalar jako *funkcji* może wydawać się nietypowe, lecz jest to całkowicie naturalna reprezentacja. Funkcje omawiam szczegółowo dalej, a na razie możesz myśleć o nich w intuicyjnym ujęciu. W zapisie używam notacji $x + y$, ale jeśli potraktować + jako funkcję dwóch zmiennych, można równie dobrze zastosować zapis $+(x, y)$. Forma $x + y$ nazywana jest notacją *infiksową*, natomiast $+(x, y)$ to notacja *prefiksowa*.

W przestrzeniach wektorowych argumentami dodawania są dwa wektory, a wynikiem jest pojedynczy wektor, więc + jest funkcją odwzorowującą iloczyn kartezjański $V \times V$ na V.

Podobnie mnożenie przez skalar to operacja na skalarze i wektorze dająca w wyniku wektor. Oznacza to, że jest to funkcja odwzorowująca $F \times V$ na V.

Iloczyn kartezjański $V \times V$ to zbiór uporządkowanych par:

$$V \times V = \{(\boldsymbol{u}, \boldsymbol{v}) : \boldsymbol{u}, \boldsymbol{v} \in V\}.$$

Więcej szczegółów na ten temat znajdziesz w dodatku dotyczącym teorii zbiorów (dodatek C), a na razie wystarczy ujęcie intuicyjne.

Warto w tym miejscu zauważyć, że definicje matematyczne są zawsze formalizowane po jakimś czasie, gdy obiekty, których dotyczą, są już w pewnym stopniu ugruntowane i znane użytkownikom. Matematyka często prezentowana jest w kolejności: najpierw definicje, potem twierdzenia. Jednak w praktyce wygląda to inaczej: to przykłady motywują do tworzenia definicji, a nie odwrotnie.

Ciałem skalarów mogą być nie tylko liczby rzeczywiste czy zespolone. Termin *ciało* oznacza dobrze zdefiniowaną strukturę matematyczną, która precyzuje intuicyjne pojęcie. Bez zagłębiania się w szczegóły techniczne wspomnę tylko, że ciała traktuję jak zbiory liczb, w których dodawanie i mnożenie działa tak samo jak dla liczb rzeczywistych.

Ponieważ nie zajmuję się tu najbardziej ogólnym przypadkiem, będę używać $\mathbb{R}$ lub $\mathbb{C}$, aby uniknąć niepotrzebnych komplikacji. Jeśli nie znasz dokładnej matematycznej definicji ciała, nie martw się. Wystarczy, że za każdym razem gdy natrafisz na słowo „ciało", zastąpisz je symbolem $\mathbb{R}$.

Kiedy wszystkie informacje wynikają z kontekstu, $(V, \mathbb{R}, +, \cdot)$ będę często oznaczał jako V, aby uprościć zapis. Zatem jeśli ciało F nie jest określone, domyślnie można przyjąć, że jest to $\mathbb{R}$. Gdy chcę podkreślić ten fakt, używam nazwy *rzeczywiste* przestrzenie wektorowe.

Definicja 1.1.1 jest z pewnością zbyt złożona, by ją od razu zrozumieć. Wydaje się tylko zbiorem operacji i właściwości zapisanych w jednym miejscu. Aby łatwiej Ci było zbudować model mentalny, możesz wyobrazić sobie wektor jako strzałkę, która wychodzi z wektora zerowego. Warto przypomnieć, że wektor zerowy 0 to szczególny wektor, dla którego $\boldsymbol{x} + 0 = x$ dla wszystkich $\boldsymbol{x}$. Można go więc uznać za strzałkę o zerowej długości, czyli początek układu współrzędnych.

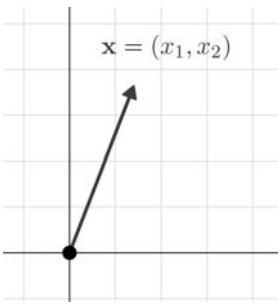
Aby lepiej zapoznać się z przestrzeniami wektorowymi, przyjrzyj się kilku przykładom.

1.1.1. Przykłady przestrzeni wektorowych

Przykłady są jednym z najlepszych sposobów na zrozumienie pozornie trudnych koncepcji, takich jak przestrzenie wektorowe. Ludzie zwykle myślą w kategoriach modeli, a nie abstrakcji. Dotyczy to również specjalistów od czystej matematyki, nawet jeśli temu zaprzeczają.

Przykład 1. Najbardziej powszechnym przykładem przestrzeni wektorowej jest $(\mathbb{R}^n, \mathbb{R}, +, \cdot)$. Jest to ta sama przestrzeń wektorowa, której użyłem w definicji. $\mathbb{R}^n$ oznacza n-krotny iloczyn kartezjański zbioru liczb rzeczywistych. Jeśli nie znasz tego zapisu, zapoznaj się z samouczkiem z teorii zbiorów w dodatku C.

$(\mathbb{R}^n, \mathbb{R}, +, \cdot)$ jest modelem kanonicznym używanym w trakcie studiów. Dla $n = 2$ powstaje znana płaszczyzna euklidesowa (rysunek 1.4).



Rysunek 1.4. Płaszczyzna euklidesowa jako przestrzeń wektorowa

Posługiwanie się $\mathbb{R}^2$ lub $\mathbb{R}^3$ do wizualizacji może być bardzo pomocne. Operacje, które działają w tych przestrzeniach, zwykle będą działać również w przypadku ogólnym, choć takie założenie nie zawsze jest bezpieczne. Matematyka opiera się zarówno na intuicji, jak i logice. Rozwijamy pomysły na podstawie intuicji, ale potwierdzamy je za pomocą logiki.

Przykład 2. Przestrzenie wektorowe to nie tylko zbiór skończonych krotek. Przykładem jest przestrzeń funkcji wielomianowych o współczynnikach rzeczywistych zdefiniowana jako

$$\mathbb{R}[x] = \left\{ \sum_{i=0}^{n} p_i x^i : p_i \in \mathbb{R}, n = 0,1, \ldots \right\}.$$

Dwa wielomiany $p(x)$ i $q(x)$ można dodać do siebie w następujący sposób:

$$p(x) + q(x) := \sum_{k=1}^{n} (p_i + q_i) x^i.$$

Z kolei operacja mnożenia przez skalar rzeczywisty wygląda tak:

$$cp(x) = \sum_{k=1}^{n} cp_i x^i.$$

Z tymi operacjami $(\mathbb{R}[x], \mathbb{R}, +, \cdot)$ jest przestrzenią wektorową. Chociaż zazwyczaj wielomiany są reprezentowane w formie funkcji, można je również przedstawić jako krotki współczynników:

$$\sum_{i=0}^{n} p_i x^i \leftrightarrow (p_0, \ldots, p_n).$$

Warto zauważyć, że n (stopień wielomianu) jest nieograniczony. W konsekwencji ta przestrzeń wektorowa ma znacznie bogatszą strukturę niż $\mathbb{R}^n$.

Przykład 3. Poprzedni przykład można dalej uogólnić. Niech $C([0, 1])$ oznacza zbiór wszystkich ciągłych funkcji rzeczywistych $f : [0, 1] \to \mathbb{R}$. Wtedy $(C(\mathbb{R}), \mathbb{R}, +, \cdot)$ jest przestrzenią wektorową, gdzie dodawanie i mnożenie przez skalar są zdefiniowane dla każdego elementu:

$$(f + g)(x) := f(x) + g(x), (cf)(x) = cf(x)$$

dla wszystkich $f, g \in C(\mathbb{R})$ i $c \in \mathbb{R}$. Chociaż nie zdefiniowałem jeszcze pojęcia ciągłości, możesz myśleć o funkcji ciągłej jako takiej, której wykres da się narysować bez odrywania długopisu od kartki.

Tak, to prawda: funkcje również można traktować jako wektory. Przestrzenie funkcji odgrywają istotną rolę w matematyce i występują w różnych formach. Często taka przestrzeń jest ograniczana do funkcji ciągłych, funkcji różniczkowalnych lub dowolnego innego podzbioru zamkniętego ze względu na dane działania.

(Również $\mathbb{R}^n$ można traktować jako przestrzeń funkcji. W ujęciu abstrakcyjnym każdy wektor $x = (x_1, \ldots, x_n)$ jest odwzorowaniem ze zbioru $\{1, 2, \ldots, n\}$ na $\mathbb{R}$).

Przestrzenie funkcji pojawiają się w kontekście zaawansowanych zagadnień, takich jak odwracanie architektur ResNet. W tej książce nie omawiam takich tematów, warto jednak zapoznać się z przykładami innymi niż $\mathbb{R}^n$ (i nie tak oczywistymi).

1.2. Podstawy

Chociaż przestrzenie wektorowe zawierają nieskończenie wiele wektorów, można zmniejszyć złożoność dzięki specjalnym podzbiorom, które pozwalają wyrazić *dowolny* inny wektor.

Aby lepiej zrozumieć tę ideę, rozważ znaną już przestrzeń $\mathbb{R}^n$. Występuje w niej specjalny zbiór wektorów:

$$\boldsymbol{e}_1 = (1, 0, \ldots, 0)$$
$$\boldsymbol{e}_2 = (0, 1, \ldots, 0)$$
$$\vdots$$
$$\boldsymbol{e}^n = (0, 0, \ldots, 1).$$

Można go użyć do wyrażenia każdego wektora $\boldsymbol{x} = (x_1, \ldots, x_n)$ w następującej formie:

$$x = \sum_{i=1}^{n} x_i \boldsymbol{e}_i, \qquad x_i \in \mathbb{R}, \qquad \boldsymbol{e}_i \in \mathbb{R}^n.$$

Na przykład w $\mathbb{R}^2$ występują wektory $\boldsymbol{e}_1 = (1, 0)$ i $\boldsymbol{e}_2 = (0, 1)$.

Ten przykład może wydawać się trywialny i pozornie tylko komplikuje pracę. Po co zapisywać wektory w postaci $\boldsymbol{x} = \sum_{i=1}^{n} x_i \boldsymbol{e}_i$, zamiast używać współrzędnych $(x_1, \ldots, x_n)$? Ponieważ notacja współrzędnych zależy od bazowego zestawu wektorów (w tym przykładzie $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$) używanego do wyrażenia innych wektorów.

Wektor *nie* jest tym samym co jego współrzędne! Dany wektor może mieć wiele różnych współrzędnych w różnych układach, a przechodzenie między układami jest użytecznym narzędziem.

Zatem zbiór $E = \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\} \subseteq \mathbb{R}^n$ jest szczególny, ponieważ znacznie upraszcza reprezentację wektorów. Wraz z operacjami dodawania wektorów i mnożenia przez skalar *generuje* on całą przestrzeń wektorową. E jest przykładem *bazy* przestrzeni wektorowej, czyli zbioru, który służy jako szkielet $\mathbb{R}^n$.

W tym podrozdziale wprowadzam i szczegółowo omawiam koncepcję bazy przestrzeni wektorowej.

1.2.1. Kombinacje liniowe i niezależność

Teraz odejdę od szczególnego przypadku $\mathbb{R}^n$ i przejdę do ogólnych przestrzeni wektorowych. Na podstawie przykładu dotyczącego baz pokazałem, że bardzo istotne są sumy w postaci

$$\sum_{i=1}^{n} x_i \boldsymbol{v}_i,$$

gdzie $\boldsymbol{v}_i$ to wektory, a x_i to współczynniki skalarne. Takie sumy nazywane są *kombinacjami liniowymi*. Są one *trywialne*, jeśli wszystkie współczynniki są równe zero.

Dla danego zbioru wektorów ten sam wektor może być wyrażony jako kombinacja liniowa na wiele sposobów. Na przykład jeśli $\boldsymbol{v}_1 = (1, 0)$, $\boldsymbol{v}_2 = (0, 1)$ i $\boldsymbol{v}_3 = (1, 1)$, to

$$\begin{aligned}(2, 1) &= 2\boldsymbol{v}_1 + \boldsymbol{v}_2 \\ &= \boldsymbol{v}_1 + \boldsymbol{v}_3.\end{aligned}$$

Sugeruje to, że zbiór $S = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ jest nadmiarowy, ponieważ zawiera powtarzające się informacje. Pojęcie *zależności i niezależności liniowej* precyzuje to spostrzeżenie.

Definicja 1.2.1. Zależność i niezależność liniowa

Niech V będzie przestrzenią wektorową, a $S = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ podzbiorem jej wektorów. Mówimy, że S jest *liniowo zależny*, jeśli zawiera tylko wektor zerowy lub jeśli istnieje niezerowy $\boldsymbol{v}_k$, który można wyrazić jako kombinację liniową pozostałych wektorów $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{k-1}, \boldsymbol{v}_{k+1}, \ldots, \boldsymbol{v}_n$.

S jest *liniowo niezależny*, jeśli nie jest liniowo zależny.

Zależność i niezależność liniową można rozpatrywać z innej perspektywy. Jeśli dla pewnego niezerowego wektora $\boldsymbol{v}_k$ zachodzi równość

$$\boldsymbol{v}_k = \sum_{i=1}^{k-1} x_i \boldsymbol{v}_i + \sum_{i=k+1}^{n} x_i \boldsymbol{v}_i,$$

to po odjęciu $\boldsymbol{v}_k$ od obu stron równania okazuje się, że wektor zerowy można przedstawić jako nietrywialną kombinację liniową

$$\mathbf{0} = \sum_{i=1}^{n} x_i \boldsymbol{v}_i$$

dla pewnych skalarów x_i, gdzie $x_k = -1$. Jest to równoważna definicja zależności liniowej. W ten sposób udowodniłem twierdzenie z ramki.

Twierdzenie pomocnicze 1.2.1. *Niech V będzie przestrzenią wektorową, a $S = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ podzbiorem V.*

a) *S jest liniowo zależny wtedy i tylko wtedy, jeśli wektor zerowy 0 można uzyskać jako nietrywialną kombinację liniową.*

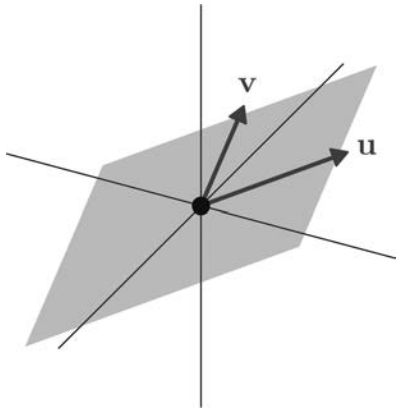
b) *S jest liniowo niezależny wtedy i tylko wtedy, gdy dla $\mathbf{0} = \sum_{i=1}^{n} x_i \boldsymbol{v}_i$ wszystkie współczynniki x_i zawsze są zerowe.*

1.2.2. Powłoki liniowe zbiorów wektorów

Kombinacje liniowe umożliwiają wygenerowanie dużej liczby wektorów na podstawie niewielkiego zbioru wektorów wyjściowych. Proces tworzenia wszystkich możliwych kombinacji liniowych dla zbioru wektorów S nazywamy *generowaniem* (lub *rozpinaniem*), a powstały w ten sposób zbiór nazywamy *powłoką liniową* i oznaczamy słowem *span*. Formalna definicja wygląda następująco:

$$\text{span}(S) = \left\{ \sum_{i=1}^{n} x_i \boldsymbol{v}_i : n \in \mathbb{N}, \boldsymbol{v}_i \in S, x_i \text{ jest skalarem} \right\}.$$

Warto zauważyć, że zbiór wektorów S nie musi być skończony. Aby lepiej zobrazować pojęcie powłoki liniowej, można przedstawić cały proces w trzech wymiarach. Powłoka liniowa dwóch liniowo niezależnych wektorów tworzy *płaszczyznę* (rysunek 1.5).



Rysunek 1.5. Powłoka liniowa dwóch liniowo niezależnych wektorów $u, v \in \mathbb{R}^3$

Powłokę liniową skończonego zbioru $\{v_1, \ldots, v_n\}$ oznacza się tak:

$$\text{span}(v_1, \ldots, v_n).$$

Pozwala to uniknąć nadmiernego komplikowania zapisu przez nazywanie osobno każdego zbioru.

Twierdzenie 1.2.1. *Niech V będzie przestrzenią wektorową, a $S, S_1, S_2 \subseteq V$ będą podzbiorami jej wektorów.*

a) *Jeśli* $S_1 \subseteq S_2$, *to* $\text{span}(S_1) \subseteq \text{span}(S_2)$.

b) $\text{span}(\text{span}(S)) = \text{span}(S)$.

To pierwsze w tej książce twierdzenie z dowodem. Przeczytaj je, a jeśli okaże się zbyt trudne, przejdź dalej i wróć do niego później. Upewnij się jednak, że rozumiesz treść twierdzenia.

Dowód. Własność *(a)* wynika bezpośrednio z definicji. Aby udowodnić *(b)*, trzeba wykazać, że $\text{span}(S) \subseteq \text{span}(\text{span}(S))$ oraz $\text{span}(\text{span}(S)) \subseteq \text{span}(S)$.

To jeden z tych momentów, gdy *krzywa uczenia się staje się stroma*. Zastanów się jednak przez chwilę: dwa zbiory A i B są równe wtedy i tylko wtedy, gdy $A \subseteq B$ i $B \subseteq A$.

Pierwszy punkt wynika z definicji. Dla drugiego przyjmijmy, że $\boldsymbol{x} \in \text{span}(\text{span}(\mathcal{S}))$. Wtedy

$$x = \sum_{i=1}^{n} x_i \boldsymbol{v}_i$$

dla pewnych $\boldsymbol{v}_i \in \text{span}(S)$. Ponieważ $\boldsymbol{v}_i$ należą do $\text{span}(S)$, mamy

$$\boldsymbol{v}_i = \sum_{j=1}^{m} v_{i,j} \boldsymbol{u}_j$$

dla pewnych $\boldsymbol{u}_j \in S$. Zatem

$$\boldsymbol{x} = \sum_{i=1}^{n} x_i \boldsymbol{v}_i = \sum_{i=1}^{n} x_i \sum_{j=1}^{m} v_{i,j} \boldsymbol{u}_j = \sum_{j=1}^{m} \left(\sum_{i=1}^{n} x_i v_{i,j} \right) \boldsymbol{u}_j,$$

co oznacza, że $\boldsymbol{x} \in \text{span}(S)$.

Ponieważ jeśli S jest liniowo zależny, to $\text{span}(\text{span}(S)) = \text{span}(S)$, można usunąć nadmiarowe wektory, a generowana przestrzeń pozostanie taka sama.

Pomyśl o następującej zależności: jeśli $S = \{v_1, \ldots, v_n\}$ i na przykład $\boldsymbol{v}_n = \sum_{i=1}^{n-1} x_i \boldsymbol{v}_i$, to $\boldsymbol{v}_n \in \text{span}(\mathcal{S} \backslash \{\boldsymbol{v}_n\})$. Zatem

$$\text{span}(\mathcal{S} \backslash \{\boldsymbol{v}_n\}) = \text{span}(\text{span}(\mathcal{S} \backslash \{\boldsymbol{v}_n\})) = \text{span}(S).$$

(Operacja $A \backslash B$ to różnica zbiorów. Wynik zawiera wszystkie elementy z A, które nie są elementami B. Więcej informacji znajdziesz w dodatku C).

W zbiorze wektorów te z nich, które generują całą przestrzeń wektorową, są szczególne. Po tym wprowadzeniu pora przedstawić formalną definicję. Każdy zbiór wektorów S, który ma właściwość $\text{span}(S) = V$, nazywamy *zbiorem generującym* dla V.

Zbiór S można traktować jako „bezstratną kompresję" przestrzeni V, ponieważ zawiera on wszystkie informacje potrzebne do odtworzenia dowolnego elementu z V, a jednocześnie jest mniejszy niż cała przestrzeń. Celem jest maksymalne zmniejszenie rozmiaru zbioru generującego. To prowadzi do jednego z najważniejszych pojęć w algebrze liniowej: minimalnych zbiorów generujących, które nazywane są *bazami*.

1.2.3. Bazy, czyli minimalne zbiory generujące

Po tych intuicyjnych rozważaniach przejdę od razu do definicji.

Definicja 1.2.2. Baza

Niech V będzie przestrzenią wektorową, a S podzbiorem jej wektorów. S jest bazą V, jeśli:

a) S jest liniowo niezależny

b) oraz $\text{span}(S) = V$.

Elementy zbioru bazowego nazywamy *wektorami bazowymi*.

Można wykazać, że te definiujące właściwości oznaczają, iż każdy wektor x może zostać *jednoznacznie* zapisany jako kombinacja liniowa S. Wykazanie tego pozostawiam jako ćwiczenie dla czytelnika.

Pora przedstawić kilka przykładów. W $\mathbb{R}^3$ zbiór {(1, 0, 0), (0, 1, 0), (0, 0, 1)} jest bazą, ale {(1, 1, 1), (1, 1, 0), (0, 1, 1)} również nią jest. Tak więc dla tej samej przestrzeni wektorowej może istnieć więcej niż jedna baza.

Dla $\mathbb{R}^n$ najczęściej używaną bazą jest $\{e_1, \ldots, e_n\}$, gdzie e_i jest wektorem, którego prawie wszystkie współrzędne są równe 0; wyjątkiem jest i-ta współrzędna, która wynosi 1. Taka baza nazywana jest *standardową*.

Jeśli chodzi o „informacje" zawarte w zbiorze wektorów, bazy są optymalne. Dodanie jakiegokolwiek nowego wektora do zbioru bazowego wprowadza redundancję, a usunięcie któregokolwiek z elementów powoduje, że zbiór staje się niekompletny.

Formalnym ujęciem tych spostrzeżeń są dwa twierdzenia z ramki.

Twierdzenie 1.2.2. *Niech V będzie przestrzenią wektorową, a $S = \{v_1, \ldots, v_n\}$ podzbiorem wektorów. Następujące stwierdzenia są równoważne:*

a) *S jest bazą.*

b) *S jest liniowo niezależny i dla każdego $x \in V \backslash S$ zbiór $S \cup \{x\}$ jest liniowo zależny. Innymi słowy, S jest maksymalnym zbiorem liniowo niezależnym.*

Dowód. Aby wykazać równoważność tych dwóch stwierdzeń, należy udowodnić dwa punkty: że *(a)* implikuje *(b)*, a także że *(b)* implikuje *(a)*. Zacznę od pierwszego z nich.

(a) $\Rightarrow$ *(b)* Jeśli S jest bazą, to dowolny $x \in V$ można zapisać jako

$$x = \sum_{i=1}^{n} x_i \boldsymbol{v}_i$$

dla pewnych $x_i \in \mathbb{R}$. Zatem z definicji $S \cup \{x\}$ jest liniowo zależny.

(b) $\Rightarrow$ *(a)* Celem jest wykazanie, że dowolny x można zapisać jako kombinację liniową wektorów z S. Zgodnie z założeniem $S \cup \{x\}$ jest liniowo zależny, więc $\mathbf{0}$ można zapisać jako nietrywialną kombinację liniową

$$\mathbf{0} = \alpha \boldsymbol{x} + \sum_{i=1}^{n} x_i \boldsymbol{v}_i,$$

gdzie nie wszystkie współczynniki są zerowe. Ponieważ S jest liniowo niezależny, α nie może być zerem (gdyż implikowałoby to liniową zależność S, co byłoby sprzeczne z założeniami). Zatem

$$\boldsymbol{x} = -\sum_{i=1}^{n} \frac{x_i}{\alpha} \boldsymbol{v}_i,$$

co dowodzi, że S jest bazą.

Teraz wykażę, że każdy wektor bazy jest niezbędny.

Twierdzenie 1.2.3. *Niech V będzie przestrzenią wektorową, a $S = \{v_1, ..., v_n\}$ jej bazą. Wtedy dla dowolnego $v_i \in S$*

$$\operatorname{span}(S \backslash \{v_i\}) \subset V,$$

czyli powłoka liniowa $S\backslash\{v_i\}$ jest podzbiorem właściwym V.

Dowód. Przeprowadzę dowód nie wprost. Bez utraty ogólności można założyć, że $i = 1$. Jeśli

$$\operatorname{span}(S \setminus \{v_1\}) = V,$$

to

$$\boldsymbol{v}_1 = \sum_{i=2}^{n} x_i \boldsymbol{v}_i.$$

Oznacza to, że $S = \{v_1, ..., v_n\}$ *nie* jest liniowo niezależny, co jest sprzeczne z założeniami.

Innymi słowy, powyższe wyniki oznaczają, że baza jest jednocześnie *maksymalnym zbiorem liniowo niezależnym* i *minimalnym zbiorem generującym*.

Gdy dana jest baza $S = \{v_1, ..., v_n\}$, wektor $\boldsymbol{x} = \sum_{i=1}^{n} x_i \boldsymbol{v}_i$ można zapisać w skróconej formie jako $x = (x_1, ..., x_n)$. Ponieważ taki rozkład jest jednoznaczny, można to zrobić bez obaw. Współczynniki x_i są nazywane również *współrzędnymi* wektora. Warto zauważyć, że współrzędne zależą od wybranej bazy. Dla dwóch różnych baz współrzędne tego samego wektora mogą być inne.

1.2.4. Przestrzenie wektorowe o skończonej liczbie wymiarów

Wcześniej wyjaśniłem, że pojedyncza przestrzeń wektorowa może mieć wiele różnych baz. Bazy nie są więc unikatowe. W tym kontekście pojawia się następujące pytanie: jeśli S_1 i S_2 są dwiema bazami przestrzeni V, to czy zachodzi równość $|S_1| = |S_2|$? W tym zapisie $|S|$ oznacza *kardynalność* zbioru S, czyli jego „wielkość”.

Innymi słowy, czy można uzyskać lepszy efekt dzięki bardziej przemyślanemu doborowi bazy? Okazuje się, że nie jest to możliwe, ponieważ liczności *dowolnych* dwóch zbiorów bazowych są takie same. Nie będę tego udowadniać, ale przedstawiam pełną postać twierdzenia.

Twierdzenie 1.2.4. *Niech V będzie przestrzenią wektorową, a S_1 i S_2 dwiema bazami V. Wtedy $|S_1| = |S_2|$.*

Na tej podstawie można zdefiniować *wymiar* przestrzeni wektorowej, czyli kardynalność jej bazy. Wymiar V jest oznaczany jako $dim(V)$. Na przykład $\mathbb{R}^n$ jest n-wymiarowa, na co wskazuje baza standardowa $\{(1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$.

W poprzednich twierdzeniach przyjmowałem założenie, że baza jest skończona. Można zadać pytanie: czy zawsze tak jest? Odpowiedź brzmi: nie. Obrazują to przykłady 2. i 3. Na przykład przeliczalnie nieskończony zbiór $\{1, x, x^2, x^3, \ldots\}$ jest bazą dla $\mathbb{R}[x]$. Zgodnie z powyższym twierdzeniem dla tej przestrzeni nie może istnieć skończona baza.

Prowadzi to do ważnego rozróżnienia między przestrzeniami wektorowymi: te z bazami skończonymi nazywane są *skończenie wymiarowymi*. Mam dobrą wiadomość: *wszystkie* skończenie wymiarowe rzeczywiste przestrzenie wektorowe są w istocie przestrzeniami $\mathbb{R}^n$ (warto przypomnieć w tym miejscu, że przestrzeń wektorowa jest nazywana *rzeczywistą*, jeśli jej skalary są liczbami rzeczywistymi).

Aby zrozumieć, dlaczego tak się dzieje, przyjmij, że V jest n-wymiarową rzeczywistą przestrzenią wektorową z bazą $\{v_1, \ldots, v_n\}$. Ponadto zdefiniuj odwzorowanie $\varphi : V \to \mathbb{R}^n$ w następujący sposób:

$$\varphi: \sum_{i=1}^{n} x_i \boldsymbol{v}_i \to (x_1, \ldots, x_n).$$

φ jest odwracalne i zachowuje strukturę V, czyli operacje dodawania i mnożenia przez skalar. Rzeczywiście, jeśli $u, v \in V$ i $\alpha, \beta \in \mathbb{R}$, to $\varphi(\alpha u + \beta v) = \alpha\varphi(x) + \beta\varphi(y)$. Takie odwzorowania nazywane są *izomorfizmami*. Słowo to pochodzi z greki, gdzie *isos* oznacza *ten sam*, a *morphe* — *kształt*. Choć brzmi to abstrakcyjnie, występowanie izomorfizmu między dwiema przestrzeniami wektorowymi oznacza, że mają one tę samą strukturę. Zatem $\mathbb{R}^n$ nie jest tylko przykładem skończenie wymiarowych rzeczywistych przestrzeni wektorowych, lecz stanowi ich uniwersalny model. Warto zauważyć, że jeśli skalary nie są liczbami rzeczywistymi, izomorfizm z $\mathbb{R}^n$ nie zachodzi. Więcej o przekształceniach piszę w dalszych rozdziałach.

Ponieważ w tej książce zajmuję się prawie wyłącznie skończenie wymiarowymi rzeczywistymi przestrzeniami wektorowymi, jest to dobra wiadomość. Używanie $\mathbb{R}^n$ jest nie tylko heurystyką, ale dobrym modelem mentalnym.

1.2.5. Dlaczego bazy są tak istotne?

Jeśli każda skończenie wymiarowa rzeczywista przestrzeń wektorowa jest w istocie taka sama jak $\mathbb{R}^n$, co możesz zyskać dzięki abstrakcji? Owszem, możesz pracować z $\mathbb{R}^n$ bez uwzględniania baz, ale aby dogłębnie zrozumieć podstawowe koncepcje matematyczne w uczeniu maszynowym, potrzebna będzie abstrakcja w postaci baz.

Wybiegnę na chwilę w przyszłość i przedstawię przykład. Jeśli masz doświadczenie z sieciami neuronowymi, wiesz, że macierze odgrywają w nich bardzo istotną rolę. Bez kontekstu macierze są tylko tabelami liczb z pozornie arbitralnymi regułami wykonywania obliczeń. Czy kiedykolwiek zdarzyło Ci się zastanawiać, dlaczego mnożenie macierzy jest zdefiniowane w taki, a nie inny sposób?

Chociaż jeszcze dokładnie nie zdefiniowałem macierzy, prawdopodobnie już je znasz. Omawiam je szczegółowo w rozdziałach 3. i 4., a na razie wspomnę tylko, że dla dwóch macierzy:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix}, B = \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,n} \\ b_{2,1} & b_{2,2} & \dots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \dots & b_{n,n} \end{bmatrix}$$

ich iloczyn AB jest zdefiniowany w następujący sposób:

$$AB = \begin{bmatrix} \sum_{k=1}^{n} a_{1,k}b_{k,1} & \sum_{k=1}^{n} a_{1,k}b_{k,2} & \dots & \sum_{k=1}^{n} a_{1,k}b_{k,n} \\ \sum_{k=1}^{n} a_{2,k}b_{k,1} & \sum_{k=1}^{n} a_{2,k}b_{k,2} & \dots & \sum_{k=1}^{n} a_{2,k}b_{k,n} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{n,k}b_{k,1} & \sum_{k=1}^{n} a_{n,k}b_{k,2} & \dots & \sum_{k=1}^{n} a_{n,k}b_{k,n} \end{bmatrix}.$$

Tak więc element (i, j) macierzy AB jest zdefiniowany jako

$$\sum_{k=1}^{n} a_{i,k}b_{k,j}.$$

Ta definicja może wydawać się arbitralna. Dlaczego nie pomnożyć odpowiadających sobie elementów: $\left(a_{i,j}b_{i,j}\right)_{i,j=1}^{n}$? Przedstawiona definicja staje się jednak w pełni zrozumiała, gdy potraktujesz macierz jako narzędzie do opisywania przekształceń liniowych między przestrzeniami wektorowymi, gdzie elementy macierzy opisują obrazy wektorów bazowych. W tym kontekście mnożenie macierzy jest składaniem przekształceń liniowych.

Zamiast tylko podać definicję i pokazać, jak jej używać, chcę pomóc Ci *zrozumieć*, dlaczego wygląda ona w taki, a nie inny sposób. W kolejnych rozdziałach wyjaśniam wszystkie szczegóły mnożenia macierzy.

1.2.6. Istnienie baz

W tym momencie możesz zadać sobie pytanie: czy dla każdej przestrzeni wektorowej istnieje baza? Bez takiej gwarancji cała wcześniejsza teoria byłaby bezużyteczna, ponieważ może nie występować baza, z którą można pracować.

Na szczęście baza zawsze istnieje. Ponieważ dowód na to jest niezwykle skomplikowany, nie będę go tu przedstawiać, ale to twierdzenie jest tak ważne, że warto je przynajmniej sformułować. Jeśli interesuje Cię dowód, przedstawiam tu jego zarys. Możesz go pominąć, gdyż w tej książce nie jest on niezbędny.

Twierdzenie 1.2.5. *Każda przestrzeń wektorowa ma bazę.*

Dowód (zarys). W dowodzie tego twierdzenia wykorzystywana jest zaawansowana technika zwana *indukcją pozaskończoną*, która znacznie wykracza poza zakres rozważań z tej książki (szczegóły można znaleźć w pracy *Naive Set Theory* Paula Halmosa). Dlatego zamiast przedstawiać precyzyjny dowód, skupię się na intuicyjnym wyjaśnieniu, jak skonstruować bazę dla dowolnej przestrzeni wektorowej.

Pokażę, jak krok po kroku ustalić bazę dla przestrzeni wektorowej V. Należy wybrać dowolny niezerowy wektor v_1. Jeśli $\text{span}(S_1) \neq V$, to zbiór $S_1 = \{v_1\}$ nie jest jeszcze bazą. Można więc znaleźć wektor $v_2 \in V \setminus \text{span}(S_1)$ taki, że $S_2 := S_1 \cup \{v_2\}$ nadal jest liniowo niezależny.

Czy S_2 jest bazą? Jeśli nie, należy kontynuować proces. Gdyby zakończył się on po skończonej liczbie kroków, uzyskana zostałaby baza. Jednak nie ma gwarancji, że tak się stanie. Pomyśl o $\mathbb{R}[x]$, przestrzeni wektorowej wielomianów, która nie jest skończenie wymiarowa, co wyjaśniłem w punkcie 1.2.4.

Na tym etapie trzeba zastosować zaawansowane narzędzia z teorii mnogości (których nie omawiam).

Jeśli opisany proces się nie zatrzyma, należy znaleźć zbiór $S_{\aleph_0}$, który zawiera wszystkie S_i jako podzbiory (znalezienie zbioru $S_{\aleph_0}$ jest najtrudniejszą częścią dowodu). Czy $S_{\aleph_0}$ jest bazą? Jeśli nie, proces jest kontynuowany.

Trudno to wykazać, ale proces w końcu się zakończy i nie będzie można dodać więcej wektorów do ich liniowo niezależnego zbioru bez naruszenia własności niezależności. Gdy to nastąpi, oznacza to znalezienie *maksymalnego liniowo niezależnego zbioru*, a więc bazy.

Dla skończenie wymiarowych przestrzeni wektorowych powyższy proces jest łatwy do opisania. Co więcej, jednym z filarów algebry liniowej jest tak zwany proces Grama-Schmidta, używany do bezpośredniego konstruowania specjalnych baz dla przestrzeni wektorowych. Ponieważ wiele istotnych wyników opiera się na tym procesie, szczegółowo przeanalizuję go w dalszych rozdziałach.

1.2.7. Podprzestrzenie

Zanim przejdę do praktycznego zastosowania wektorów w Pythonie, warto uwzględnić jeszcze jeden temat, który przyda się przy omawianiu przekształceń liniowych (takie przekształcenia są niezbędne w uczeniu maszynowym; wszystko, co opisuję, służy lepszemu ich zrozumieniu). Dla danej przestrzeni wektorowej V często istotny jest jeden z jej podzbiorów, który sam w sobie jest przestrzenią wektorową. Takie podzbiory to *podprzestrzenie*.

Definicja 1.2.3. Podprzestrzenie

Niech V będzie przestrzenią wektorową. Zbiór $U \subseteq V$ jest podprzestrzenią V, jeśli jest zamknięty ze względu na dodawanie i mnożenie przez skalar.

U jest *podprzestrzenią właściwą*, jeśli jest podprzestrzenią i $U \subset V$.

Podprzestrzenie z definicji są przestrzeniami wektorowymi, więc można określić ich wymiar. Dla każdej przestrzeni wektorowej istnieją co najmniej dwie podprzestrzenie: ona sama i {0}. Nazywamy je podprzestrzeniami *trywialnymi*. Ponadto powłoka liniowa zbioru wektorów zawsze tworzy podprzestrzeń. Przykład tego został zilustrowany na rysunku 1.5.

Jednym z najważniejszych aspektów podprzestrzeni jest to, że można ich użyć do tworzenia kolejnych podprzestrzeni. Precyzyjnie opisuje to definicja.

Definicja 1.2.4. Suma prosta podprzestrzeni

Niech V będzie przestrzenią wektorową, a U_1 i U_2 jej podprzestrzeniami. Sumą prostą U_1 i U_2 nazywamy zbiór

$$U_1 + U_2 = \{\boldsymbol{u}_1 + \boldsymbol{u}_2 : \boldsymbol{u}_1 \in U_1, \boldsymbol{u}_2 \in U_2\}.$$

Można łatwo stwierdzić, że $U_1 + U_2$ rzeczywiście jest podprzestrzenią, a $U_1 + U_2 = \text{span}(U_1 \cup U_2)$. Podprzestrzenie i ich suma prosta odgrywają istotną rolę w wielu zagadnieniach, na przykład w rozkładach macierzy. Dalej wyjaśniam, że rozkład macierzy często można sprowadzić do rozkładu przestrzeni liniowej na sumę przestrzeni wektorowych.

Często przydatna okazuje się umiejętność wyboru bazy, której podzbiory generują określone podprzestrzenie. Formalnie ujmuje to kolejne twierdzenie.

Twierdzenie 1.2.6. *Niech V będzie przestrzenią wektorową, a U_1 i U_2 jej podprzestrzeniami spełniającymi warunek $U_1 + U_2 = V$. Ponadto niech $\{p_1, ..., p_k\} \subseteq U_1$ będzie bazą U_1, a $\{q_1, ..., q_1\} \subseteq U_2$ bazą U_2. Wówczas suma*

$$\{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k\} \cup \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_l\}$$

stanowi bazę w przestrzeni V.

Dowód. Dowód wynika bezpośrednio z definicji sumy prostej. Jeśli $V = U_1 + U_2$, to dowolny wektor $\boldsymbol{x} \in V$ można zapisać w postaci $\boldsymbol{x} = \boldsymbol{a} + \boldsymbol{b}$, gdzie $\boldsymbol{a} \in U_1$ i $\boldsymbol{b} \in U_2$.

Ponieważ $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k$ tworzą bazę w U_1, a $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_l$ tworzą bazę w U_2, wektory $\boldsymbol{a}$ i $\boldsymbol{b}$ można zapisać jako

$$\boldsymbol{a} = \sum_{i=1}^{k} a_i \boldsymbol{p}_i, \boldsymbol{b} = \sum_{i=1}^{l} b_i \boldsymbol{q}_i.$$

Zatem dowolny wektor $\boldsymbol{x}$ przyjmuje postać

$$\boldsymbol{x} = \sum_{i=1}^{k} a_i \boldsymbol{p}_i + \sum_{i=1}^{l} b_i \boldsymbol{q}_i,$$

co stanowi definicję bazy.

To jednak dopiero początek. Bazy są niezwykle ważne, ale stanowią jedynie szkielet dla przestrzeni wektorowych spotykanych w praktyce. Aby właściwie reprezentować i przetwarzać dane, trzeba zbudować wokół tego szkieletu strukturę geometryczną. Jak możemy zmierzyć „odległość" między dwoma pomiarami? A co z ich podobieństwem?

Poza tym pojawia się jeszcze ważniejsze pytanie: jak reprezentować wektory w komputerze? W następnym podrozdziale omawiam struktury danych z Pythona, aby zapewnić podstawy dla operacji i transformacji danych, które będą wykonywane później.

1.3. Wektory w praktyce

Do tej pory pisałem głównie o teorii wektorów i przestrzeni wektorowych. Jednak ostatecznym celem jest zbudowanie modeli obliczeniowych do wykrywania i analizowania wzorców w danych. Aby przełożyć teorię na praktykę, należy wyjaśnić, jak wektory są reprezentowane w obliczeniach.

W informatyce istnieje wyraźna różnica między tym, jak myślimy o strukturach matematycznych, a tym, jak są one reprezentowane w komputerze. Do tej pory celem było opracowanie ram matematycznych, które pozwalają rozumować o strukturze danych i ich przekształceniach. Odpowiedni język powinien być:

- ekspresyjny,
- łatwy w użyciu,
- możliwie zwięzły.

Jednak cele wyglądają inaczej, gdy zamiast czystego rozumowania logicznego potrzebne są obliczenia. Wtedy odpowiednia implementacja powinna być:

- łatwa w obsłudze,
- wydajna pamięciowo,
- szybka w dostępie, operacjach i przekształcaniu.

Te wymogi są często sprzeczne ze sobą, a w konkretnych sytuacjach możesz preferować jedną z tych cech kosztem pozostałych. Na przykład jeśli masz dużo pamięci, ale

chcesz wykonać wiele obliczeń, możesz poświęcić wydajność pamięciową na rzecz szybkości. Ze względu na różnorodność potencjalnych zastosowań istnieje wiele formatów reprezentacji tych samych pojęć matematycznych. Te formaty są nazywane *strukturami danych*.

W poszczególnych językach programowania wektory są zaimplementowane w odmienny sposób. Ponieważ Python jest wszechobecny w danologii i uczeniu maszynowym, to właśnie z niego korzystam w tej książce. W tym rozdziale omawiam wszystkie struktury danych dostępne w Pythonie, aby ustalić, która z nich najlepiej nadaje się do reprezentowania wektorów w obliczeniach wymagających wysokiej wydajności.

1.3.1. Krotki

W standardowym Pythonie istnieją (co najmniej) dwie wbudowane struktury danych, które można wykorzystać do reprezentowania wektorów: *krotki* i *listy*. Zacznę od krotek. Można je łatwo zdefiniować przez podanie ich rozdzielonych przecinkami elementów między dwoma nawiasami okrągłymi:

```
v_tuple = (1, 3.5, -2.71, "napis", 42)
v_tuple
(1, 3.5, -2.71, "napis", 42)
type(v_tuple)
tuple
```

Pojedyncza krotka może przechowywać elementy różnych typów. Mimo że w obliczeniowej algebrze liniowej będziesz pracować wyłącznie z liczbami zmiennoprzecinkowymi, ta właściwość jest niezwykle przydatna w programowaniu ogólnym.

Dostęp do elementów krotki uzyskuje się za pomocą indeksów. W Pythonie, podobnie jak w wielu innych językach programowania, indeksowanie zaczyna się od zera. Inaczej jest w matematyce, gdzie indeksowanie często zaczyna się od jedynki. W związku z tym w większości języków przeznaczonych do obliczeń naukowych, takich jak Fortran, Matlab czy Julia, indeksowanie zaczyna się od jedynki.

(Nie mów tego nikomu, ale indeksowanie od zera doprowadzało mnie kiedyś do szału. Z wykształcenia jestem matematykiem):

```
v_tuple[0]
1
```

Rozmiar krotki można sprawdzić za pomocą wbudowanej funkcji `len`:

```
len(v_tuple)
5
```

Inną obok indeksowania metodą dostępu do elementów jest *pobieranie wycinka*:

```
v_tuple[1:4]
(3.5, -2.71, "napis")
```

Pobieranie wycinka wymaga określenia pierwszego i ostatniego elementu z opcjonalnym rozmiarem kroku. Służy do tego składnia *obiekt*[*pierwszy*:*ostatni*:*krok*].

Krotki są mało elastyczne, ponieważ nie można modyfikować ich elementów. Próba wykonania takiej operacji skutkuje błędem `TypeError`. W Pythonie jest to standardowy sposób na informowanie, że obiekt nie obsługuje metody, którą próbujesz wywołać. W tym przykładzie chodzi o brak możliwości przypisania wartości do elementu krotki:

```
v_tuple[0] = 2
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[22], line 1
----> 1 v_tuple[0] = 2

TypeError: 'tuple' object does not support item assignment
```

Ponadto niemożliwe jest również rozszerzanie krotki o dodatkowe elementy. Ponieważ nie można w żaden sposób zmienić stanu obiektu `tuple` po jego utworzeniu, jest on *niemodyfikowalny*. W zależności od sytuacji niemodyfikowalność może być albo wadą, albo zaletą. Obiekty niemodyfikowalne chronią przed przypadkowymi zmianami wartości, ale każda operacja wymaga utworzenia nowego obiektu, co wiąże się z dodatkowym obciążeniem obliczeniowym. Dlatego krotki nie są optymalne do reprezentowania dużych ilości danych w złożonych obliczeniach.

Ten problem rozwiązują *listy*. Omówię teraz tę strukturę i nowe wyzwania, które ze sobą niesie.

1.3.2. Listy

Listy są podstawowym narzędziem w Pythonie. W odróżnieniu od krotek są niezwykle elastyczne i łatwe w użyciu, choć wiąże się to z niższą wydajnością. Obiekt `list` jest tworzony podobnie jak obiekt `tuple`, a mianowicie należy podać oddzielone przecinkami elementy, ale ujęte w nawiasy kwadratowe:

```
v_list = [1, 3.5, -2.71, "qwerty"]
type(v_list)
list
```

Dostęp do elementów listy również uzyskuje się podobnie jak w krotkach, czyli za pomocą indeksowania lub pobierania wycinka. Na listach można wykonywać różnego rodzaju operacje: zastępować elementy, dodawać nowe lub usuwać istniejące.

```
v_list[0] = "to jest napis"
v_list
['to jest napis', 3.5, -2.71, 'qwerty']
```

Ten przykład pokazuje, że także listy mogą zawierać elementy różnych typów. Do dodawania i usuwania elementów służą metody takie jak `append`, `push`, `pop` i `remove`.

Zanim je wypróbujesz, zwróć uwagę na adres przykładowej listy w pamięci. Możesz go uzyskać za pomocą funkcji `id`:

```
v_list_addr = id(v_list)
v_list_addr
126433407319488
```

Ta liczba reprezentuje adres w pamięci mojego komputera określający, gdzie znajduje się obiekt `v_list`. Dosłownie tak jest, ponieważ ta książka powstaje na moim osobistym komputerze.

Teraz wykonam kilka prostych operacji na liście i pokażę, że jej adres w pamięci nie ulega zmianie. Oznacza to, że nie jest tworzony żaden nowy obiekt.

```
v_list.append([42]) # Dodawanie listy [42] na koniec pierwotnej listy
v_list
['to jest napis', 3.5, -2.71, 'qwerty', [42]]
(v_list) == v_list_addr # Dodawanie elementów nie powoduje tworzenia nowych obiektów
True
v_list.pop(1) # Usuwanie elementu o indeksie "1"
v_list
['to jest napis', -2.71, 'qwerty', [42]]
id(v_list) == v_list_addr # Usuwanie elementów także nie powoduje tworzenia nowych
                          # obiektów
True
```

Niestety dodawanie list do siebie daje wynik zupełnie inny od oczekiwanego:

```
[1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

Zamiast dodawać do siebie odpowiadające sobie elementy, co byłoby pożądane dla wektorów, listy są złączane (operacja konkatenacji). Ta cecha jest przydatna przy tworzeniu aplikacji ogólnego przeznaczenia, ale nie sprawdza się dobrze w obliczeniach naukowych. Operacja mnożenia przez skalar również daje nieoczekiwane rezultaty:

```
3*[1, 2, 3]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Mnożenie listy przez liczbę całkowitą powoduje powtórzenie zawartości struktury określoną liczbę razy. Jeśli wziąć pod uwagę działanie operatora + dla list, wydaje się to logiczne, ponieważ mnożenie przez liczbę całkowitą to w istocie wielokrotne dodawanie:

$$a \cdot b = \underbrace{b + \cdots + b}_{a \text{ razy}}$$

Listy oferują znacznie więcej możliwości niż potrzeba do reprezentacji wektorów. Chociaż potrzebna jest możliwość zmiany elementów wektorów, nie jest konieczne dodawanie ani usuwanie elementów, ani przechowywanie w nich obiektów innych niż liczby zmiennoprzecinkowe. Czy można zrezygnować z tych dodatkowych funkcji i uzyskać implementację, która będzie odpowiednia do reprezentowania wektorów, a jednocześnie zapewni wysoką wydajność obliczeniową? Tak. Umożliwiają to tablice NumPy.

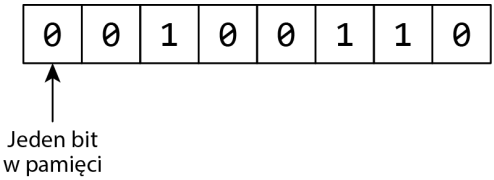
1.3.3. Tablice NumPy

Choć wbudowane struktury danych Pythona są niezwykle przydatne, zostały zoptymalizowane pod kątem łatwości użycia, a nie obliczeń naukowych. Problem ten dostrzeżono już na wczesnym etapie rozwoju języka i rozwiązano przez udostępnienie biblioteki NumPy.

Jedną z głównych zalet Pythona jest szybkość i prostota pisania kodu nawet dla złożonych zadań. Odbywa się to jednak kosztem wydajności, a w uczeniu maszynowym szybkość jest niezwykle istotna. Podczas uczenia sieci neuronowej niewielki zestaw operacji jest powtarzany miliony razy. Nawet drobna poprawa wydajności może zaoszczędzić godziny, dni, a wręcz tygodnie, gdy model jest wyjątkowo duży.

Język C znajduje się na przeciwnym końcu spektrum. Kod w C jest trudny w użyciu, ale jeśli jest poprawnie opracowany, działa błyskawicznie. Ponieważ Python jest napisany w C, sprawdzoną metodą osiągania wysokiej wydajności jest wywoływanie funkcji z języka C z poziomu Pythona. Właśnie to zapewnia biblioteka NumPy: udostępnia w Pythonie tablice i operacje napisane w C.

Aby lepiej zrozumieć problemy związane z wbudowanymi strukturami danych Pythona, przyjrzyj się bliżej liczbom i tablicom. W pamięci komputera obiekty są reprezentowane jako ciągi zer i jedynek o stałej długości. Każdy element takiego ciągu nazywany jest bitem. Bity są zwykle grupowane w 8-, 16-, 32-, 64- lub nawet 128-bitowe fragmenty. W zależności od tego, co jest reprezentowane, identyczne sekwencje mogą oznaczać różne rzeczy. Na przykład 8-bitowa sekwencja *00100110* może reprezentować liczbę całkowitą 38 lub znak ASCII „&" (rysunek 1.6).

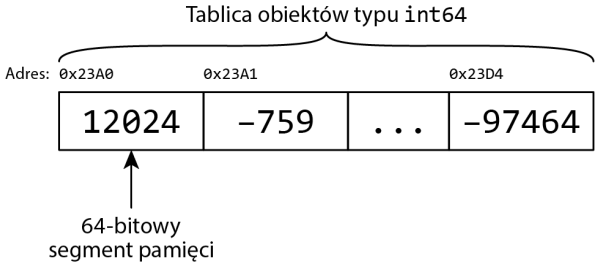


Rysunek 1.6. Ośmiobitowy obiekt w pamięci

Dzięki podaniu *typu danych* można dekodować obiekty binarne. 32-bitowe liczby całkowite są typu `int32`, 64-bitowe liczby zmiennoprzecinkowe mają typ `float64` i tak dalej.

Ponieważ pojedynczy bit zawiera bardzo mało informacji, pamięć jest adresowana przez podział jej na kolejno numerowane fragmenty o rozmiarze 32 lub 64 bitów. Adres jest liczbą *szesnastkową* zaczynającą się od 0. Dla uproszczenia przyjmuję tu, że pamięć jest adresowana co 64 bity. Jest to typowe rozwiązanie w nowoczesnych komputerach.

Naturalnym sposobem przechowywania sekwencji powiązanych obiektów (o tym samym typie danych) jest umieszczenie ich obok siebie w pamięci. Taka struktura danych nazywana jest *tablicą* (rysunek 1.7).



Rysunek 1.7. Tablica obiektów typu int64

Dzięki zapisaniu adresu pierwszego obiektu w pamięci, na przykład `0x23A0`, można natychmiast uzyskać dostęp do elementu *k*-tego. W tym celu wystarczy wskazać pamięć o adresie `0x23A0 + k`.

Taka struktura nazywana jest tablicą statyczną lub tablicą języka C, ponieważ tak właśnie jest ona zaimplementowana w tym świetnym języku. Chociaż ta implementacja tablic działa błyskawicznie, jest stosunkowo mało elastyczna. Po pierwsze, można przechowywać w niej obiekty tylko jednego typu. Po drugie, trzeba z góry znać rozmiar tablicy, gdyż nie można używać adresów pamięci wykraczających poza wstępnie przydzielony obszar. Dlatego przed rozpoczęciem pracy z tablicą należy zarezerwować dla niej pamięć (czyli zająć miejsce, aby inne programy nie nadpisały umieszczonych tam danych).

Natomiast w Pythonie można przechowywać na jednej liście dowolnie duże i różne obiekty. Możliwe jest również usuwanie z niej elementów i dodawanie nowych.

```
l = [2**142 + 1, "napis"]
l.append(lambda x: x)
l
```

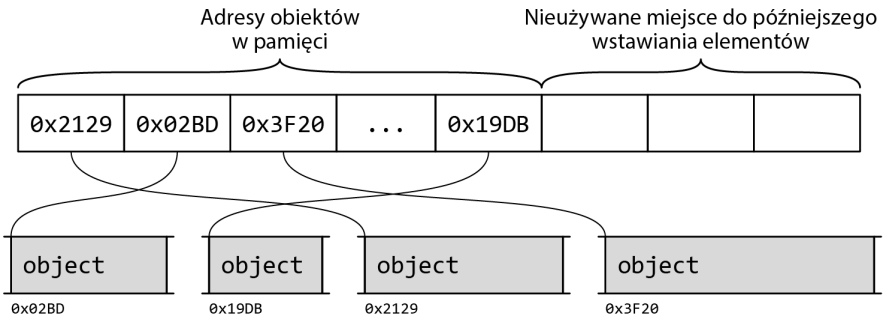
```
[5575186299632655785383929568162090376495105,
 'napis',
 <function __main__.<lambda>(x)>]
```

W tym przykładzie `l[0]` to liczba całkowita tak duża, że nie mieści się w 128 bitach. Co więcej, na liście znajdują się obiekty różnego rodzaju, w tym funkcja. Jak to możliwe?

Typ `list` w Pythonie jest elastyczną strukturą danych dzięki:

1. nadmiarowemu przydzielaniu pamięci,
2. przechowywaniu adresów obiektów w pamięci zamiast samych obiektów.

Takie rozwiązanie stosowane jest w najpopularniejszej implementacji CPython (rysunek 1.8): *https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython*.



Rysunek 1.8: Implementacja list w CPythonie

Jeśli sprawdzisz adresy w pamięci każdego obiektu z listy `l`, zauważysz, że poszczególne obiekty są zapisane w różnych miejscach:

```
[id(x) for x in l]
```

```
[126433412959232, 126433407528240, 126433410174944]
```

Dzięki nadmiarowej alokacji pamięci usuwanie lub wstawianie elementów zawsze można wykonać przez zwykłe przesunięcie pozostałych elementów. Ponieważ lista przechowuje adresy elementów w pamięci, w jednej strukturze można zapisać obiekty różnych typów.

Jednak ma to swoją cenę. Jako że obiekty nie są zapisane w pamięci jeden za drugim, następuje utrata lokalności odwołań (*https://en.wikipedia.org/wiki/Locality_of_reference*). To oznacza, że częste używanie odległych obszarów pamięci spowalnia odczyt danych. W rezultacie iteracyjne pobieranie elementów listy w Pythonie nie jest efektywne.

Z kolei tablice NumPy przypominają tradycyjne tablice znane z języka C, ale dostępne w Pythonie z przyjaznym dla użytkownika interfejsem podobnym do list Pythona. Jeśli kiedykolwiek pracowałeś z językiem C, wiesz, jakim błogosławieństwem jest to rozwiązanie. Zobacz teraz, jak pracować z tablicami NumPy.

Na początek zaimportuj bibliotekę `numpy`. Aby skrócić kod, zgodnie z konwencją jest ona importowana jako `np`:

```
import numpy as np
```

Główną strukturą danych jest `np.ndarray` (nazwa to skrót od ang. *n-dimensional array*, czyli tablica *n*-wymiarowa). Możesz używać funkcji `np.array` do tworzenia tablic NumPy na podstawie standardowych kontenerów z Pythona lub inicjalizować tablice od podstaw. (Tak, wiem, że ta składnia może być myląca, ale przyzwyczaisz się do niej. Po prostu zapamiętaj, że `np.ndarray` to klasa, a `np.array` to funkcja służąca do tworzenia tablic NumPy na podstawie obiektów Pythona).

```
X = np.array([87.7, 4.5, -4.1, 42.1414, -3.14, 2.001]) # Tworzenie tablicy NumPy
                                                       # na podstawie listy Pythona
X
array([87.7   ,  4.5   , -4.1   , 42.1414, -3.14  ,  2.001 ])
np.ones(shape=7) # Inicjalizacja tablicy NumPy za pomocą jedynek
array([1., 1., 1., 1., 1., 1., 1.])
np.zeros(shape=5) # Inicjalizacja tablicy NumPy za pomocą samych zer
array([0., 0., 0., 0., 0.])
```

Można również inicjalizować tablice NumPy przy użyciu liczb losowych:

```
np.random.rand(10)
array([0.92428404, 0.37719596, 0.92071695, 0.56905245, 0.12024811,
       0.02868856, 0.53215047, 0.51749348, 0.21022765, 0.96749756])
```

Co najważniejsze, gdy masz już tablicę, możesz zainicjalizować inną o takich samych wymiarach za pomocą funkcji `np.zeros_like`, `np.ones_like` oraz `np.empty_like`:

```
np.zeros_like(X)
array([0., 0., 0., 0., 0., 0.])
```

Tablice NumPy, podobnie jak listy Pythona, obsługują przypisywanie wartości do elementów oraz pobieranie wycinków:

```
X[0] = 1545.215
X
array([1545.215 , 4.5 , -4.1 , 42.1414, -3.14 , 2.001 ])
X[1:4]
array([ 4.5   , -4.1   , 42.1414])
```

Jak można się spodziewać, w jednej tablicy `ndarray` możesz przechowywać dane tylko jednego typu. Próba przypisania łańcucha znaków jako pierwszego elementu spowoduje wyświetlenie komunikatu o błędzie:

```
X[0] = "str"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[48], line 1
----> 1 X[0] = "str"

ValueError: could not convert string to float: 'str
```

Jak można się domyślić, każda tablica `ndarray` ma atrybut określający typ danych. Wartość tego atrybutu można sprawdzić za pomocą wywołania `ndarray.dtype`. Jeśli możliwe jest dokonanie konwersji między przypisywaną wartością a typem danych tablicy, zostanie ona wykonana automatycznie, a element zostanie poprawnie przypisany. Dzięki temu operacje na tablicach stają się bardziej elastyczne i wygodne w użyciu.

```
X.dtype
```

```
dtype('float64')
```

```
val = 23
type(val)
```

```
int
```

```
X[0] = val
X
```

```
array([23.   , 4.5  , -4.1  , 42.1414 , -3.14  , 2.001 ])
```

Tablice NumPy, podobnie jak inne typy kontenerów w Pythonie, umożliwiają iterowanie:

```
for x in X:
    print(x)
```

```
23.0
4.5
-4.1
42.1414
-3.14
2.001
```

Czy te tablice nadają się do reprezentowania wektorów? Tak. Zaraz wyjaśnię, dlaczego tak jest.

1.3.4. Tablice NumPy jako wektory

Pora wrócić do wektorów. Od tego miejsca do modelowania wektorów będę używać tablic `ndarray` z biblioteki NumPy.

```
v_1 = np.array([-4.0, 1.0, 2.3])
v_2 = np.array([-8.3, -9.6, -7.7])
```

Operacje dodawania i mnożenia przez skalar są domyślnie obsługiwane i działają zgodnie z oczekiwaniami:

```
v_1 + v_2 # Dodawanie wektorów v_1 i v_2
```

```
array([-12.3, -8.6, -5.4])
```

```
10.0*v_1 # Mnożenie wektora v_1 przez skalar
```

```
array([-40., 10., 23.])
v_1 * v_2 # Iloczyn po współrzędnych wektorów v_1 i v_2
array([ 33.2 , -9.6 , -17.71])
np.zeros(shape=3) + 1
array([1., 1., 1.])
```

Dzięki stosowanemu w Pythonie dynamicznemu określaniu typów często można używać tablic NumPy w funkcjach przeznaczonych dla wartości skalarnych:

```
def f(x):
    return 3*x**2 - x**4
f(v_1)
array([-208.  , 2.    , -12.1141])
```

Na razie tablice NumPy spełniają prawie wszystkie wymagania związane z reprezentowaniem wektorów. Pozostaje jeszcze tylko jedna kwestia do sprawdzenia: wydajność. Aby ją zbadać, zmierzę czas wykonywania kodu za pomocą wbudowanego w Pythona narzędzia `timeit`.

Funkcja `timeit` (*https://docs.python.org/3/library/timeit.html*) jako pierwszy argument przyjmuje funkcję, która ma zostać wykonana z pomiarem czasu. Obok przekazywania obiektu funkcji można również podać instrukcje do wykonania w postaci ciągu znaków. Ponieważ wywołania funkcji w Pythonie wiążą się ze znaczącym narzutem obliczeniowym, tu przekazuję kod zamiast obiektu funkcji, aby uzyskać dokładniejsze pomiary czasu.

Poniżej porównuję dodawanie dwóch tablic NumPy z dodawaniem list Pythona. Wszystkie te struktury zawierają po tysiąc zer.

```
from timeit import timeit

n_runs = 100000
size = 1000


t_add_builtin = timeit(
    "[x + y for x, y in zip(v_1, v_2)]",
    setup=f"size={size}; v_1 = [0 for _ in range(size)]; v_2 = [0 for _ in
    ↪range(size)]",
    number=n_runs
)

t_add_numpy = timeit(
    "v_1 + v_2",
    setup=f"import numpy as np; size={size}; v_1 = np.zeros(shape=size);
    v_2 = np.zeros(shape=size)",
    number=n_runs
)

print(f"Dodawanie wbudowane: \t{t_add_builtin} s")
print(f"Dodawanie NumPy:     \t{t_add_numpy} s")
print(f"Poprawa wydajności:  \t{t_add_builtin/t_add_numpy:.3f} raza szybciej")
Dodawanie wbudowane: 3.3522969299992837 s
Dodawanie NumPy:     0.09616518099937821 s
Poprawa wydajności:  34.860 raza szybciej
```

Tablice NumPy są znacznie szybsze. Wynika to z następujących powodów:

- są ułożone w pamięci w sposób ciągły;
- są jednorodne pod względem typu danych;
- mają operacje zaimplementowane w języku C.

To tylko początek. Pokazałem zaledwie niewielką część możliwości biblioteki NumPy, ale oferuje ona znacznie więcej niż tylko szybkie struktury danych. W miarę postępu lektury będę stopniowo omawiać ją coraz bardziej szczegółowo, aby przedstawić szeroki zestaw funkcji, jakie zapewnia ta biblioteka.

1.3.5. Czy NumPy naprawdę jest szybsza niż Python?

NumPy jest zaprojektowana w taki sposób, aby działać szybciej niż standardowy Python. Ale czy rzeczywiście tak jest? Nie zawsze. Jeśli użyjesz jej niewłaściwie, może nawet spowodować spadek wydajności. Aby zrozumieć, kiedy użycie NumPy jest korzystne, wytłumaczę teraz dokładniej, dlaczego w praktyce jest ona szybsza.

Aby uprościć analizy, w przykładowym problemie skupię się na generowaniu liczb losowych. Załóżmy, że potrzebna jest tylko jedna liczba losowa. Czy należy wtedy użyć NumPy? Sprawdźmy to. Porównam ją z wbudowanym generatorem liczb losowych. Oba rozwiązania uruchomię dziesięć milionów razy i zmierzę czas wykonania:

```
from numpy.random import random as random_np
from random import random as random_py

n_runs = 10000000
t_builtin = timeit(random_py, number=n_runs)
t_numpy = timeit(random_np, number=n_runs)

print(f"Wbudowana funkcja random:\t{t_builtin} s")
print(f"Funkcja random z NumPy:  \t{t_numpy} s")
```

```
Wbudowana funkcja random:    0.47474874800172984 s
Funkcja random z NumPy:      5.1664929229991685 s
```

Generowanie pojedynczej liczby losowej za pomocą NumPy jest znacznie wolniejsze. Dlaczego tak się dzieje? A co się stanie, jeśli potrzebna jest tablica zamiast pojedynczej liczby? Czy wtedy też NumPy będzie wolniejsza?

Tym razem kod będzie generował listę i tablicę składającą się z tysiąca elementów:

```
size = 1000
n_runs = 10000

t_builtin_list = timeit(
    "[random_py() for _ in range(size)]",
    setup=f"from random import random as random_py; size={size}",
    number=n_runs
)
```

```
t_numpy_array = timeit(
    "random_np(size)",
    setup=f"from numpy.random import random as random_np; size={size}",
    number=n_runs
)

print(f"Wbudowana funkcja random dla list: \t{t_builtin_list}s")
print(f"Funkcja random z NumPy dla tablic: \t{t_numpy_array}s")
```

```
Wbudowana funkcja random dla list: 0.5773125300001993s
Funkcja random z NumPy dla tablic: 0.08449692800058983s
```

(Także w tym przykładzie nie umieszczam mierzonych wyrażeń w lambdach, ponieważ w Pythonie wywołania funkcji powodują koszty. Chcę uzyskać możliwie precyzyjne pomiary, dlatego wyrażenia przekazuję do funkcji `timeit` jako łańcuchy znaków).

Sytuacja wygląda teraz zupełnie inaczej. Jeśli chodzi o generowanie tablicy losowych liczb, NumPy zdecydowanie wygrywa.

Wyniki wskazują na kilka ciekawych kwestii. Najpierw wygenerowałem pojedynczą losową liczbę 10 000 000 razy. Następnie 10 000 razy wygenerowałem tablicę 1000 losowych liczb. W obu przykładach ostatecznie powstaje 10 000 000 losowych liczb. Gdy używana jest wbudowana metoda, zapisanie tych liczb na liście zajęło około 2 razy więcej czasu niż samo ich wygenerowanie. Natomiast gdy użyłem NumPy, uzyskałem mniej więcej 30-krotne przyspieszenie w porównaniu do pracy z pojedynczymi liczbami. Na Twoim komputerze te wartości mogą oczywiście wyglądać inaczej.

Aby zobaczyć, co dzieje się za kulisami, przeprowadzę profilowanie kodu za pomocą narzędzia cProfiler (*https://docs.python.org/3/library/profile.html*). Dzięki temu dokładnie określę, ile razy dana funkcja została wywołana i ile czasu zajęło jej wykonywanie.

Zacznę od funkcji wbudowanej. W tym przykładzie kod, podobnie jak wcześniej, tworzy 10 000 000 losowych liczb:

```
def builtin_random_single(n_runs):
    for _ in range(n_runs):
        random_py()
```

W notatnikach Jupytera, gdzie powstaje ta książka, narzędzie cProfiler można wywołać za pomocą specjalnego polecenia `%prun`:

```
n_runs = 10000000

%prun builtin_random_single(n_runs)
```

```
10000558 function calls (10000539 primitive calls) in 2.082 seconds
  Ordered by: internal time
  ncalls  tottime  percall cumtime percall filename:lineno(function)
       1    0.937    0.937   1.671   1.671 2471337341.py:1(builtin_random_single)
10000000    0.911    0.000   0.911   0.000 {method 'random' of '_random.Random'
↪objects}
     4/0    0.213    0.053   0.000         {method 'poll' of 'select.epoll'
     ↪objects}
      10    0.009    0.001   0.016   0.002 socket.py:626(send)
       2    0.009    0.004   0.015   0.008 {method '__exit__' of 'sqlite3.
       ↪Connection' objects}
```

W tym kontekście istotne są dwie kolumny. Kolumna `ncalls` informuje, ile razy dana funkcja została wywołana, natomiast `tottime` określa całkowity czas spędzony w funkcji (z wyłączeniem czasu wykonywania podfunkcji).

Wbudowana funkcja `random.random()` została wywołana 10 000 000 razy, co jest zgodne z oczekiwaniami. Zwróć uwagę na całkowity czas spędzony w tej funkcji (nie podaję dokładnej wartości, ponieważ zależy ona od maszyny, na której uruchamiany jest kod).

A co z wersją wykorzystującą NumPy? Wyniki są zaskakujące:

```
def numpy_random_single(n_runs):
    for _ in range(n_runs):
        random_np()

%prun numpy_random_single(n_runs)
448 function calls (444 primitive calls) in 7.203 seconds

Ordered by: internal time

ncalls tottime percall cumtime percall filename:lineno(function)
     1   7.029   7.029   7.029   7.029 2015715881.py:1(numpy_random_single)
     2   0.136   0.068   0.136   0.068 {method 'poll' of 'select.epoll' objects}
     2   0.015   0.007   0.015   0.007 {method '__exit__' of 'sqlite3.Connection'
    ↪objects}
     1   0.011   0.011   0.011   0.011 {method 'execute' of 'sqlite3.Connection'
    ↪objects}
     3   0.010   0.003   7.339   2.446 base_events.py:1910(_run_once)
     7   0.000   0.000   0.000   0.000 socket.py:626(send)
     1   0.000   0.000   0.000   0.000 {method 'disable' of '_lsprof.Profiler'
    ↪objects}
     1   0.000   0.000   0.026   0.026 history.py:833(_writeout_input_cache)
     1   0.000   0.000   0.000   0.000 inspect.py:3102(_bind)
 88/84   0.000   0.000   0.000   0.000 {built-in method builtins.isinstance}
```

Podobnie jak wcześniej funkcja `numpy.random.random()` została wywołana dokładnie 10 000 000 razy, co jest zgodne z oczekiwaniami. Jednak tym razem skrypt spędził znacznie więcej czasu na wykonywaniu tej funkcji niż w przypadku wbudowanej funkcji `random` Pythona. Oznacza to, że koszt pojedynczego wywołania funkcji `numpy.random.↪random()` jest wyższy.

Gdy zaczynasz pracować z dużymi tablicami i listami, sytuacja zmienia się diametralnie. W następnym przykładzie generuję listy i tablice zawierające po 1000 losowych liczb oraz mierzę czas wykonywania tych operacji:

```
def numpy_random_single(n_runs):
    for _ in range(n_runs):
        random_np()
%prun numpy_random_single(n_runs)
448 function calls (444 primitive calls) in 7.203 seconds

Ordered by: internal time

ncalls tottime percall cumtime percall filename:lineno(function)
     1   7.029   7.029   7.029   7.029 2015715881.py:1(numpy_random_single)
     2   0.136   0.068   0.136   0.068 {method 'poll' of 'select.epoll' objects}
```

```
     2   0.015   0.007   0.015   0.007 {method '__exit__' of 'sqlite3.Connection'
    ↪objects}
     1   0.011   0.011   0.011   0.011 {method 'execute' of 'sqlite3.Connection'
    ↪objects}
     3   0.010   0.003   7.339   2.446 base_events.py:1910(_run_once)
     7   0.000   0.000   0.000   0.000 socket.py:626(send)
     1   0.000   0.000   0.000   0.000 {method 'disable' of '_lsprof.Profiler'
    ↪objects}
     1   0.000   0.000   0.026   0.026 history.py:833(_writeout_input_cache)
     1   0.000   0.000   0.000   0.000 inspect.py:3102(_bind)
 88/84   0.000   0.000   0.000   0.000 {built-in method builtins.isinstance}
```

Jak widać, około 60% czasu zajęło wykonywanie wyrażeń listowych. Warto zauważyć, że `tottime` nie uwzględnia wywołań podfunkcji, takich jak wywołania `random.random()` w tym przykładzie.

Teraz mogę wytłumaczyć, dlaczego NumPy jest szybsza, gdy używa się jej prawidłowo.

```
def numpy_random_array(size, n_runs):
    for _ in range(n_runs):
        random_np(size)
%prun numpy_random_array(size, n_runs)
149 function calls (148 primitive calls) in 0.132 seconds

Ordered by: internal time

ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.122   0.122   0.122   0.122 1681905588.py:1(numpy_random_array)
     2   0.009   0.004   0.009   0.004 {method '__exit__' of 'sqlite3.Connection'
objects}
   2/1   0.000   0.000   0.122   0.122 {built-in method builtins.exec}
```

Każde z 10 000 wywołań funkcji zwraca tablicę `numpy.ndarray` zawierającą 1000 losowych liczb. NumPy jest szybka, gdy używa się jej prawidłowo, ponieważ tablice NumPy są niezwykle wydajne. Przypominają one tablice z języka C, a nie listy Pythona.

Między listami Pythona a tablicami NumPy występują więc dwie istotne różnice:

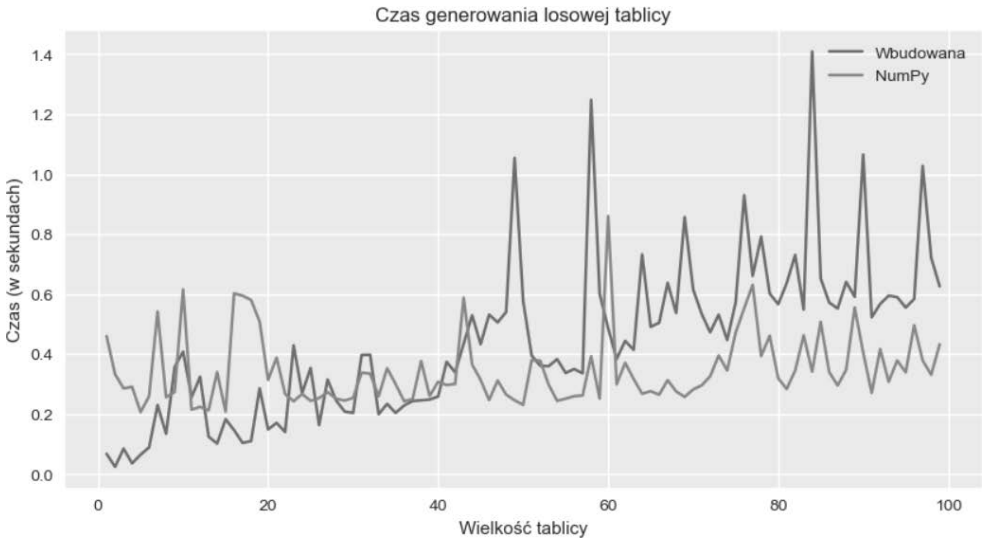
- Listy Pythona są dynamiczne, więc umożliwiają na przykład dodawanie i usuwanie elementów. Tablice NumPy mają stałą długość, więc nie można dodawać ani usuwać elementów bez tworzenia nowej tablicy.
- Listy Pythona mogą przechowywać jednocześnie elementy różnych typów danych, natomiast tablica NumPy może zawierać dane tylko jednego typu.

Tablice NumPy są mniej elastyczne, ale znacznie wydajniejsze. Gdy większa elastyczność nie jest potrzebna, użycie biblioteki NumPy daje lepsze wyniki niż korzystanie ze standardowego Pythona.

Aby dokładnie określić, przy jakim rozmiarze danych NumPy zaczyna przewyższać Pythona w generowaniu losowych liczb (rysunek 1.9), można zmierzyć czasy wykonania dla różnych rozmiarów próbek i porównać oba podejścia:

```
sizes = list(range(1, 100))

runtime_builtin = [
    timeit(
```

```
            "[random_py() for _ in range(size)]",
            setup=f"from random import random as random_py; size={size}",
            number=100000
        )
        for size in sizes
    ]

    runtime_numpy = [
        timeit(
            "random_np(size)",
            setup=f"from numpy.random import random as random_np; size={size}",
            number=100000
        )
        for size in sizes

    import matplotlib.pyplot as plt

    with plt.style.context("seaborn-v0_8"):
        plt.figure(figsize=(10, 5))
        plt.plot(sizes, runtime_builtin, label="Wbudowana")
        plt.plot(sizes, runtime_numpy, label="NumPy")
        plt.xlabel("wielkość tablicy")
        plt.ylabel("czas (w sekundach)")
        plt.title("Czas generowania losowej tablicy")
        plt.legend() plt.show()
```



Rysunek 1.9. Czas generowania losowej tablicy

Przy mniej więcej 40 elementach NumPy zaczyna przewyższać wydajnością standardowego Pythona. Oczywiście ta granica może być odmienna dla innych operacji, takich jak obliczanie sinusa czy dodawanie liczb, ale zależność pozostaje taka sama. Python jest nieco szybszy dla małych zbiorów danych, lecz biblioteka NumPy okazuje się zdecydowanie wydajniejsza, gdy rozmiar danych rośnie.

1.4. Podsumowanie

W tym rozdziale wyjaśniłem, czym są wektory i dlaczego są niezbędne w danologii i uczeniu maszynowym. Wektory to nie tylko zbiór połączonych ze sobą liczb, lecz struktura matematyczna, która pozwala efektywniej analizować dane (zarówno w teorii, jak i w praktyce). Wbrew powszechnemu przekonaniu wektory są wektorami nie dlatego, że mają kierunek i wartość, ale dlatego, że można je dodawać.

Ująłem to formalnie w postaci koncepcji *przestrzeni wektorowych*, które stanowią matematyczne ramy dla dalszych analiz. Przestrzenie wektorowe najlepiej opisywać za pomocą *baz*, czyli minimalnych i liniowo niezależnych zbiorów generujących. Zrozumienie przestrzeni wektorowych i ich baz okaże się bardzo pomocne, gdy będziesz studiować przekształcenia liniowe, które są najważniejszym elementem modeli predykcyjnych.

Wektory nie tylko pozwalają zastosować abstrakcyjne podejście, ale też dają znaczące korzyści praktyczne dzięki przekształceniu kodu na postać wektorową. Pomaga to uprościć złożoną logikę do jednowierszowych operacji. Dotyczy to na przykład skalowania danych:

```
X_scaled = (X - X.mean(axis=0)) / X.std(axis=0)
```

W tym rozdziale oprócz koncepcyjnego przejścia od skalarów do wektorów i macierzy opisałem efektywne przetwarzanie danych dzięki NumPy (jest to skrót od ang. *numerical Python*), która jest najważniejszą biblioteką w zestawie narzędzi do uczenia maszynowego. Jeśli jakaś biblioteka do obsługi tensorów nie korzysta bezpośrednio z NumPy, to z pewnością autorzy przynajmniej się nią inspirowali. Znasz już podstawy korzystania z NumPy i wiesz, dlaczego i kiedy warto jej używać.

W następnym rozdziale kontynuuję omawianie przestrzeni wektorowych. Bazy są oczywiście ważne, ale oprócz nich przestrzenie wektorowe mają piękną i bogatą strukturę geometryczną. Warto przyjrzeć się jej bliżej.

1.5. Zadania

Zadanie 1. Nie wszystkie przestrzenie wektorowe są nieskończone. Istnieją takie, które zawierają tylko skończoną liczbę wektorów. Tego właśnie dotyczy ten problem. Zdefiniujmy zbiór

$$\mathbb{Z}_2 := \{0, 1\},$$

gdzie operacje + i · są zdefiniowane w następujący sposób:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0$$

oraz:

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1.$$

Jest to tzw. arytmetyka binarna (lub modulo 2).

a) Wykaż, że $(\mathbb{Z}_2, \mathbb{Z}_2, +, \cdot)$ jest przestrzenią wektorową.

b) Wykaż, że $(\mathbb{Z}_2^n, \mathbb{Z}_2, +, \cdot)$ również jest przestrzenią wektorową, gdzie $\mathbb{Z}_2^n$ jest n-krotnym iloczynem kartezjańskim

$$\mathbb{Z}_2^n = \underbrace{\mathbb{Z}_2 \times \cdots \times \mathbb{Z}_2}_{n \text{ razy}},$$

a dodawanie i mnożenie przez skalar są zdefiniowane element po elemencie:

$$x + y = (x_1 + y_1, \ldots, x_n + y_n), x, y \in \mathbb{Z}_2^n$$

$$cx = (cx_1, \ldots, cx_n), c \in \mathbb{Z}_2.$$

Zadanie 2. Czy poniższe zbiory wektorów są liniowo niezależne?

a) $S_1 = \{(1, 0, 0), (1, 1, 0), (1, 1, 1)\} \subseteq \mathbb{R}^3$

b) $S_2 = \{(1, 1, 1), (1, 2, 4), (1, 3, 9)\} \subseteq \mathbb{R}^3$

c) $S_3 = \{(1, 1, 1), (1, 1, -1), (1, -1, -1)\} \subseteq \mathbb{R}^3$

d) $S_4 = \{(\pi, e), (-42, 13/6), (\pi^3, -2)\} \subseteq \mathbb{R}^2$

Zadanie 3. Niech V będzie skończoną n-wymiarową przestrzenią wektorową, a $S = \{v_1, \ldots, v_m\}$ liniowo niezależnym zbiorem wektorów, gdzie $m < n$. Udowodnij, że istnieje taka baza B, że $S \subset B$.

Zadanie 4. Niech V będzie przestrzenią wektorową, a $S = \{v_1, \ldots, v_n\}$ jej bazą. Udowodnij, że każdy wektor $x \in V$ można jednoznacznie zapisać jako kombinację liniową wektorów z S (tzn. jeśli $\boldsymbol{x} = \sum_{i=1}^{n} \alpha_i \boldsymbol{v}_i = \sum_{i=1}^{n} \beta_i \boldsymbol{v}_i$, to $\alpha_i = \beta_i$ dla wszystkich $i = 1, \ldots, n$).

Zadanie 5. Niech V będzie dowolną przestrzenią wektorową, a $U_1, U_2 \subseteq V$ jej dwiema podprzestrzeniami. Udowodnij, że $U_1 + U_2 = \text{span}(U_1 \cup U_2)$.

Wskazówka: aby udowodnić równość tych dwóch zbiorów, należy wykazać dwie rzeczy:

1) jeśli $x \in U_1 + U_2$, to $x \in \text{span}(U_1 \cup U_2)$,

2) jeśli $x \in \text{span}(U_1 \cup U_2)$, to $x \in U_1 + U_2$.

Zadanie 6. Rozważ przestrzeń wektorową wielomianów o współczynnikach rzeczywistych zdefiniowaną jako

$$\mathbb{R}[x] = \left\{ p(x) = \sum_{i=0}^{n} p_i x^i : p_i \in \mathbb{R}, n = 0,1, \ldots \right\}.$$

a) Udowodnij, że

$$x\mathbb{R}[x] := \left\{ p(x) = \sum_{i=1}^{n} p_i x^i : p_i \in \mathbb{R}, n = 1,2,\ldots \right\}$$

jest podprzestrzenią właściwą $\mathbb{R}[x]$.

b) Wykaż, że

$$f : \mathbb{R}[x] \to x\mathbb{R}[x],\ p(x) \mapsto xp(x)$$

jest bijekcją i odwzorowaniem liniowym. Funkcja $f : X \to Y$ jest bijekcją, jeśli dla każdego $y \in Y$ istnieje dokładnie jeden $x \in X$, dla którego $f(x) = y$. Jeśli nie znasz jeszcze tego pojęcia, możesz wrócić do zadania po przeczytaniu rozdziału 9.

Liniowa i bijektywna funkcja $f : U \to V$ między przestrzeniami wektorowymi nazywana jest *izomorfizmem*. Gdy istnieje taka funkcja, można powiedzieć, że przestrzenie wektorowe U i V są *izomorficzne*, co oznacza, iż mają identyczną strukturę algebraiczną.

Po połączeniu *(a)* i *(b)* otrzymasz, że $\mathbb{R}[X]$ jest izomorficzne ze swoją podprzestrzenią właściwą $x\mathbb{R}[X]$. Jest to interesujące zjawisko: przestrzeń wektorowa, która jest algebraicznie identyczna ze swoją podprzestrzenią właściwą. Warto zauważyć, że nie jest to możliwe dla przestrzeni skończenie wymiarowych, takich jak $\mathbb{R}^n$.

W każdej wiedzy jest tyle prawdy, ile jest w niej matematyki!

— Immanuel Kant

Uczenie maszynowe jest powszechnie stosowane w aplikacjach, jednak szczegóły związane z aspektami teoretycznymi bywają zaniedbywane. Często wynika to z braku swobody w posługiwaniu się matematyką. Tymczasem bez solidnych podstaw w tym zakresie nie można mówić o profesjonalnym podejściu do uczenia maszynowego.

Dzięki tej książce poznasz najważniejsze dziedziny matematyki — algebrę liniową, rachunek różniczkowy i całkowy, a także teorię prawdopodobieństwa — niezbędne do opanowania zaawansowanych koncepcji w uczeniu maszynowym. Poszczególne zagadnienia przedstawiono z wyjątkową przejrzystością i w uporządkowany sposób. W książce powiązano teorię z praktyką: koncepcje matematyczne zostały bezpośrednio zastosowane w przykładach z zakresu uczenia maszynowego, zaimplementowanych w Pythonie. Wiedza uzyskana w toku lektury będzie przydatna na przykład w trenowaniu modeli uczenia maszynowego metodą spadku gradientu czy w pracy z wektorami, macierzami i tablicami wielowymiarowymi.

W książce znajdziesz najważniejsze koncepcje i zasady z dziedziny:

- algebry liniowej, w tym macierze, wartości własne i rozkłady
- rachunku różniczkowego i całkowego, w tym różniczkowanie i całkowanie
- złożonych technik analizy wielu zmiennych
- teorii prawdopodobieństwa, w tym rozkłady, twierdzenie Bayesa i entropię

Dr Tivadar Danka jest z wykształcenia matematykiem, z zawodu inżynierem, z powołania nauczycielem i niezależnym myślicielem. Od kilku lat zgłębia meandry uczenia maszynowego. Jego prace wyróżnia jasny, intuicyjny styl tłumaczenia skomplikowanych zagadnień — dzięki czemu nawet trudne pojęcia stają się logiczne i zrozumiałe.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-3325-5
9 788328 933255
Cena: 139,00 zł
‹packt›