O'REILLY®

Praktyczna algebra liniowa dla analityków danych

Od podstawowych koncepcji do użytecznych aplikacji w Pythonie

Helion

Mike X Cohen

Tytuł oryginału: Practical Linear Algebra for Data Science:
From Core Concepts to Applications Using Python

Tłumaczenie: Filip Kamiński

ISBN: 978-83-289-0261-9

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

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

Printed in Poland.

Spis treści

Wstęp **11**

1. Wprowadzenie **13**
Co to jest algebra liniowa i dlaczego warto ją poznać? 13
O książce 14
Wymagania wstępne 14
Matematyka 14
Postawa 15
Programowanie 15
Dowody matematyczne kontra kod 16
Kod pokazany w książce i do pobrania z sieci 17
Ćwiczenia z programowania 17
Jak korzystać z tej książki (dla nauczycieli i osób uczących się samodzielnie)? 18

2. Wektory — część I **19**
Tworzenie i wizualizacja wektorów w NumPy 19
Geometryczna interpretacja wektorów 21
Operacje na wektorach 22
Dodawanie dwóch wektorów 22
Geometryczne dodawanie i odejmowanie wektorów 24
Mnożenie wektorów przez skalar 24
Dodawanie wektorów i skalarów 25
Transpozycja 26
Broadcasting w Pythonie 27
Moduł wektora i wektory jednostkowe 28
Iloczyn skalarny wektorów 29
Iloczyn skalarny jest rozdzielny względem dodawania 31
Geometryczna interpretacja iloczynu skalarnego 31

Inne sposoby mnożenia wektorów 32
Iloczyn Hadamarda 33
Iloczyn zewnętrzny 33
Iloczyn wektorowy i mieszany 34
Ortogonalny rozkład wektora 34
Podsumowanie 38
Ćwiczenia z programowania 38

3. Wektory — część II ... 41
Zbiory wektorów 41
Ważona kombinacja liniowa 42
Niezależność liniowa 43
Matematyka związana z niezależnością liniową 44
Niezależność a wektor zerowy 45
Podprzestrzeń i rozpinanie 46
Baza 48
Definicja bazy 51
Podsumowanie 52
Ćwiczenia z programowania 52

4. Zastosowania wektorów ... 54
Korelacja i podobieństwo cosinusowe 54
Filtrowanie szeregów czasowych i wykrywanie cech 56
Klasteryzacja za pomocą algorytmu k-średnich 57
Ćwiczenia z programowania 60
Ćwiczenia z korelacji 60
Ćwiczenia z filtrowania i wykrywania cech 62
Ćwiczenia z algorytmu k-średnich 63

5. Macierze — część I ... 64
Tworzenie i wizualizowanie macierzy w NumPy 64
Wizualizowanie, indeksowanie i slicing 64
Specjalne macierze 66
Matematyka macierzy: dodawanie, mnożenie przez skalar i iloczyn Hadamarda 68
Dodawanie i odejmowanie 68
„Przesuwanie" macierzy 68
Mnożenie przez skalar i iloczyn Hadamarda 69
Standardowe mnożenie macierzy 70
Kiedy można pomnożyć przez siebie dwie macierze? 70
Mnożenie macierzy 71
Mnożenie macierz – wektor 72

Operacje na macierzach: transpozycja 74
Iloczyn skalarny i iloczyn zewnętrzny — notacja 75
Operacje na macierzach: LIVE EVIL (kolejność operacji) 75
Macierze symetryczne 75
Tworzenie macierzy symetrycznych z macierzy niesymetrycznych 76
Podsumowanie 77
Ćwiczenia z programowania 78

6. Macierze — część II 82
Normy macierzowe 82
Ślad macierzy i norma Frobeniusa 84
Przestrzenie macierzowe (kolumnowa, wierszowa, jądro) 84
Przestrzeń kolumnowa 84
Przestrzeń wierszowa 88
Jądro 88
Rząd 91
Rzędy specjalnych macierzy 93
Rząd a dodawanie i mnożenie macierzy 95
Rząd a przesuwanie macierzy 95
Teoria a praktyka 96
Zastosowania rzędu 97
Czy wektor znajduje się w przestrzeni kolumnowej macierzy? 97
Niezależność liniowa zbioru wektorów 98
Wyznacznik 99
Obliczanie wyznacznika 100
Wyznacznik a zależność liniowa 101
Wielomian charakterystyczny 101
Podsumowanie 103
Ćwiczenia z programowania 104

7. Zastosowania macierzy 109
Wielowymiarowe macierze kowariancji danych 109
Transformacje geometryczne za pomocą mnożenia macierz – wektor 112
Wykrywanie cech na obrazie 115
Podsumowanie 118
Ćwiczenia z programowania 119
Ćwiczenia z macierzy kowariancji i korelacji 119
Ćwiczenia z transformacji geometrycznych 120
Ćwiczenia z wykrywania cech w obrazach 122

8. Odwracanie macierzy 124
Odwrotność macierzy 124
Rodzaje odwrotności i warunki odwracalności 125
Obliczanie odwrotności 125
Odwrotność macierzy 2 × 2 126
Odwrotność macierzy diagonalnej 128
Odwracanie dowolnej macierzy kwadratowej o pełnym rzędzie 128
Odwrotności jednostronne 129
Unikalność odwrotności 132
Pseudoodwrotność Moore'a-Penrose'a 132
Stabilność numeryczna obliczania odwrotności 133
Geometryczna interpretacja odwrotności 135
Podsumowanie 136
Ćwiczenia z programowania 137

9. Macierze ortogonalne i rozkład QR 140
Macierze ortogonalne 140
Ortogonalizacja Grama-Schmidta 142
Rozkład QR 142
Wymiary Q i R 144
Rozkład QR i obliczanie odwrotności 145
Podsumowanie 146
Ćwiczenia z programowania 146

10. Przekształcenie do macierzy schodkowej i rozkład LU 151
Układy równań 151
Przekształcanie równań w macierze 152
Praca z równaniami macierzowymi 153
Sprowadzanie macierzy do postaci schodkowej 154
Eliminacja Gaussa 156
Eliminacja Gaussa-Jordana 157
Odwracanie macierzy za pomocą eliminacji Gaussa-Jordana 158
Rozkład LU 160
Zamiana wierszy za pomocą macierzy permutacji 160
Podsumowanie 162
Ćwiczenia z programowania 162

11. Ogólne modele liniowe i metoda najmniejszych kwadratów 165
Ogólne modele liniowe 166
Terminologia 166
Tworzenie ogólnego modelu liniowego 166

Dopasowywanie ogólnego modelu liniowego 167
Czy to rozwiązanie jest dokładne? 168
Metoda najmniejszych kwadratów — perspektywa geometryczna 169
Dlaczego metoda najmniejszych kwadratów działa? 170
Prosty przykład ogólnego modelu liniowego 172
Rozwiązywanie problemu najmniejszych kwadratów za pomocą rozkładu QR 176
Podsumowanie 177
Ćwiczenia z programowania 177

12. Zastosowania metody najmniejszych kwadratów 181
Przewidywanie liczby wypożyczonych rowerów na podstawie pogody 181
Tabela regresji z pakietu statsmodels 186
Współliniowość 187
Regularyzacja 187
Regresja wielomianowa 188
Znajdowanie parametrów modelu za pomocą przeszukiwania siatki 191
Podsumowanie 193
Ćwiczenia z programowania 194
Zbiór danych z informacjami o wypożyczaniu rowerów 194
Ćwiczenie ze współliniowości 195
Ćwiczenie z regularyzacji 196
Ćwiczenie z regresji wielomianowej 197
Ćwiczenia z przeszukiwania siatki 198

13. Rozkład według wartości własnych 199
Interpretacje wartości i wektorów własnych 200
Interpretacja geometryczna 200
Statystyka (analiza głównych składowych) 201
Redukcja szumów 201
Redukcja wymiarowości (kompresja danych) 202
Znajdowanie wartości własnych 203
Znajdowanie wektorów własnych 205
Znak i skala nieokreśloności wektorów własnych 206
Diagonalizacja macierzy kwadratowej 207
Wyjątkowość macierzy symetrycznych 208
Ortogonalne wektory własne 208
Rzeczywiste wartości własne 210
Rozkład według wartości własnych macierzy osobliwych 211
Forma kwadratowa, określoność i wartości własne 212
Forma kwadratowa macierzy 212
Określoność 214
A^TA jest dodatnio (pół)określona 215

Uogólniony rozkład według wartości własnych 216
Podsumowanie 217
Ćwiczenia z programowania 218

14. Rozkład według wartości osobliwych 223
Spojrzenie na rozkład według wartości osobliwych z szerszej perspektywy 223
Wartości osobliwe i rzędy macierzy 225
Rozkład według wartości osobliwych w Pythonie 225
Rozkład według wartości osobliwych i „warstwy" macierzy rzędu 1. 226
Rozkład według wartości osobliwych z rozkładu według wartości własnych 228
Rozkład według wartości osobliwych A^TA 228
Zamiana wartości osobliwych na wariancję — wyjaśnienie 229
Współczynnik uwarunkowania 230
Rozkład według wartości osobliwych i pseudoodwrotność Moore'a-Penrose'a 231
Podsumowanie 232
Ćwiczenia z programowania 232

15. Zastosowania rozkładu według wartości własnych i rozkładu według wartości osobliwych 236
Analiza głównych składowych za pomocą rozkładów według wartości własnych i osobliwych 236
Matematyka analizy głównych składowych 237
Kroki algorytmu analizy głównych składowych 239
Analiza głównych składowych za pomocą rozkładu według wartości osobliwych 239
Liniowa analiza dyskryminacyjna 240
Aproksymacja macierzami niskiego rzędu za pomocą rozkładu według wartości osobliwych 242
Wykorzystanie rozkładu według wartości osobliwych do usuwania szumów 243
Podsumowanie 243
Ćwiczenia z programowania 244
Analiza głównych składowych 244
Liniowa analiza dyskryminacyjna 248
Aproksymacja macierzami niskiego rzędu za pomocą rozkładu według wartości osobliwych 252
Wykorzystanie rozkładu według wartości osobliwych do usuwania szumów z obrazu 255

16. Wprowadzenie do programowania w Pythonie 258
Dlaczego Python i jakie są alternatywy? 258
IDE (zintegrowane środowiska programistyczne) 259
Lokalny Python i Python dostępny w sieci 259
Praca z plikami kodu w Google Colab 260

Zmienne 261
Typy danych 262
Indeksowanie 263
Funkcje 264
Metody a funkcje 265
Tworzenie własnych funkcji 265
Biblioteki 266
NumPy 267
Indeksowanie i slicing w NumPy 267
Wizualizacje 268
Zamiana równań na kod 271
Formatowanie wyjścia i f-stringi 273
Przepływ sterowania 274
Operatory porównania 275
Klauzule if 275
Pętle for 277
Zagnieżdżone instrukcje kontrolne 277
Pomiar czasu obliczeń 278
Uzyskiwanie pomocy i więcej informacji 278
Co robić, gdy coś idzie nie tak 278
Podsumowanie 279

Skorowidz 281

ROZDZIAŁ 2.

Wektory — część I

Wektory stanowią fundament, na którym zbudowana jest cała algebra liniowa (a tym samym reszta tej książki).

Z tego rozdziału dowiesz się wszystkiego o wektorach: czym są, do czego służą, jak je interpretować oraz jak je tworzyć i pracować z nimi w Pythonie. Poznasz najważniejsze działania na wektorach, w tym algebrę wektorów i iloczyn skalarny. Na zakończenie poznasz rozkłady wektorów, które są jednym z głównych problemów algebry liniowej.

Tworzenie i wizualizacja wektorów w NumPy

W algebrze liniowej **wektor** to uporządkowana lista liczb. (W abstrakcyjnej algebrze liniowej wektory mogą zawierać również inne obiekty matematyczne, w tym funkcje, ale ponieważ ta książka koncentruje się na zastosowaniach, rozważamy w niej tylko wektory zawierające liczby).

Wektory mają kilka ważnych cech. Pierwsze dwie, od których zacznę, to:

Wymiarowość
: Określa liczbę liczb w wektorze.

Orientacja
: Określa, czy wektor jest **kolumnowy** (wąski i wysoki), czy **wierszowy** (płaski i szeroki).

Wymiarowość wektora często określa się za pomocą fantazyjnie wyglądającego zapisu $\mathbb{R}^N$, gdzie $\mathbb{R}$ oznacza liczby rzeczywiste (np. $\mathbb{C}$ to liczby zespolone), a N oznacza liczbę wymiarów. Na przykład wektor zawierający dwa elementy należy do $\mathbb{R}^2$. Symbol $\mathbb{R}$ można utworzyć za pomocą kodu, korzystając z LaTeX-a, ale równie dobrze możesz napisać R^2, *R2* lub *R^2*.

Równanie 2.1 zawiera kilka przykładów wektorów. Przed przeczytaniem kolejnego akapitu spróbuj określić ich wymiary i orientację.

Równanie 2.1. Przykłady wektorów kolumnowych i wierszowych

$$\mathbf{x} = \begin{bmatrix} 1 \\ 4 \\ 5 \\ 6 \end{bmatrix}, \mathbf{y} = \begin{bmatrix} 0{,}3 \\ -7 \end{bmatrix}, \mathbf{z} = [1\ 4\ 5\ 6]$$

Oto odpowiedzi: **x** to czterowymiarowy wektor kolumnowy, **y** to dwuwymiarowy wektor kolumnowy, a **z** to czterowymiarowy wektor wierszowy. Możesz też zapisać na przykład, że $\mathbf{x} \in \mathbb{R}^4$, gdzie symbol ∈ oznacza „należy do zbioru".

Czy **x** i **z** to ten sam wektor? Technicznie są to różne wektory, pomimo tego, że zawierają te same elementy występujące w tej samej kolejności. Więcej informacji na ten temat znajdziesz nieco dalej w tym rozdziale, w uwadze „Czy orientacja wektora ma znaczenie?".

W trakcie lektury tej książki oraz podczas łączenia matematyki i programowania dowiesz się, że istnieją różnice między matematyką uprawianą „na tablicy" i matematyką implementowaną w kodzie. Niektóre rozbieżności są niewielkie i nieistotne, inne zaś powodują zamieszanie i błędy. Pozwól, że przedstawię Ci różnice w terminologii stosowanej w matematyce i programowaniu.

Wspomniałem wcześniej, że *wymiarowość* wektora to liczba elementów, które się w nim znajdują. Natomiast w Pythonie wymiarowość wektora lub macierzy to liczba wymiarów geometrycznych używanych do wyświetlenia wektora lub macierzy na ekranie. Na przykład wszystkie wektory pokazane powyżej są w Pythonie „tablicami dwuwymiarowymi", niezależnie od liczby znajdujących się w nich elementów (czyli ich matematycznej wymiarowości). W Pythonie tablicą jednowymiarową nazywamy listę liczb bez określonej orientacji, niezależnie od liczby znajdującej się w niej elementów (tablica ta zostanie wyświetlona jako wiersz, ale jak zobaczysz później, Python traktuje ją inaczej niż wektory wierszowe). Matematyczna wymiarowość — tzn. liczba elementów w wektorze — nazywana jest w Pythonie długością (ang. *length*) lub kształtem (ang. *shape*) wektora.

Ta niespójna i czasami sprzeczna terminologia może być myląca, ale tak zazwyczaj bywa na styku różnych dyscyplin (w tym przypadku matematyki i informatyki). Nie martw się, przy odrobinie doświadczenia da się to ogarnąć.

Wektory często oznacza się za pomocą małych, pogrubionych liter alfabetu łacińskiego, na przykład **v** oznacza „wektor *v*". W niektórych tekstach stosuje się też kursywę (*v*) lub umieszcza strzałkę nad literą ($\vec{v}$).

O ile nie określono inaczej, w algebrze liniowej przyjmuje się, że wektory są wektorami kolumnowymi. Wektory wierszowe zapisuje się jako $\mathbf{w}^T$. Wykładnik ^{T} reprezentuje **operację transpozycji**, którą omówię później. Na razie zapamiętaj tylko, że operacja transpozycji przekształca wektor kolumnowy w wierszowy.

Czy orientacja wektora ma znaczenie?

Naprawdę musimy się martwić o to, czy wektory są wektorami kolumnowymi, wierszowymi czy tablicami jednowymiarowymi bez orientacji? Czasami tak, a czasami nie. W przypadku wektorów zawierających dane orientacja zazwyczaj nie ma znaczenia. Z drugiej strony, jeśli orientacja jest niewłaściwa, to niektóre operacje w Pythonie mogą powodować błędy lub dawać nieoczekiwane wyniki. A zatem warto zrozumieć orientację wektorów, aby nie stracić 30 minut na debugowanie kodu tylko po to, by uświadomić sobie, że wektor wierszowy powinien być wektorem kolumnowym (z pewnością taka sytuacja przyprawiłaby Cię o ból głowy).

W Pythonie wektory mogą być reprezentowane przy użyciu kilku typów danych. Najprostszym sposobem reprezentacji wektora wydaje się użycie typu `list`. Choć typ ten sprawdza się w niektórych zastosowaniach, wielu operacji algebry liniowej nie da się wykonać na listach Pythona. Dlatego zazwyczaj najlepiej jest zapisywać wektory w postaci tablic NumPy. Poniżej pokazałem cztery sposoby tworzenia wektora:

```
asList  = [1,2,3]
asArray = np.array([1,2,3]) # tablica jednowymiarowa
rowVec  = np.array([ [1,2,3] ]) # wiersz
colVec  = np.array([ [1],[2],[3] ]) # kolumna
```

Zmienna `asArray` jest tablicą *bez orientacji*, co oznacza, że nie jest ani wektorem wierszowym, ani kolumnowym, lecz po prostu jednowymiarową listą liczb w NumPy. Do określania orientacji w NumPy wykorzystuje się nawiasy — najbardziej zewnętrzne nawiasy łączą wszystkie liczby w jeden obiekt. Każdy dodatkowy zestaw nawiasów reprezentuje wiersz: wektor wierszowy (zmienna `rowVec`) zawiera wszystkie liczby zapisane w jednym wierszu, podczas gdy wektor kolumnowy (zmienna `colVec`) składa się z wielu wierszy, z których każdy zawiera tylko jedną liczbę.

Do zbadania orientacji można wykorzystać atrybut `shape` (możliwość sprawdzania kształtu obiektów często przydaje się podczas programowania):

```
print(f'asList:  {np.shape(asList)}')
print(f'asArray: {asArray.shape}')
print(f'rowVec:  {rowVec.shape}')
print(f'colVec:  {colVec.shape}')
```

Oto wyniki:

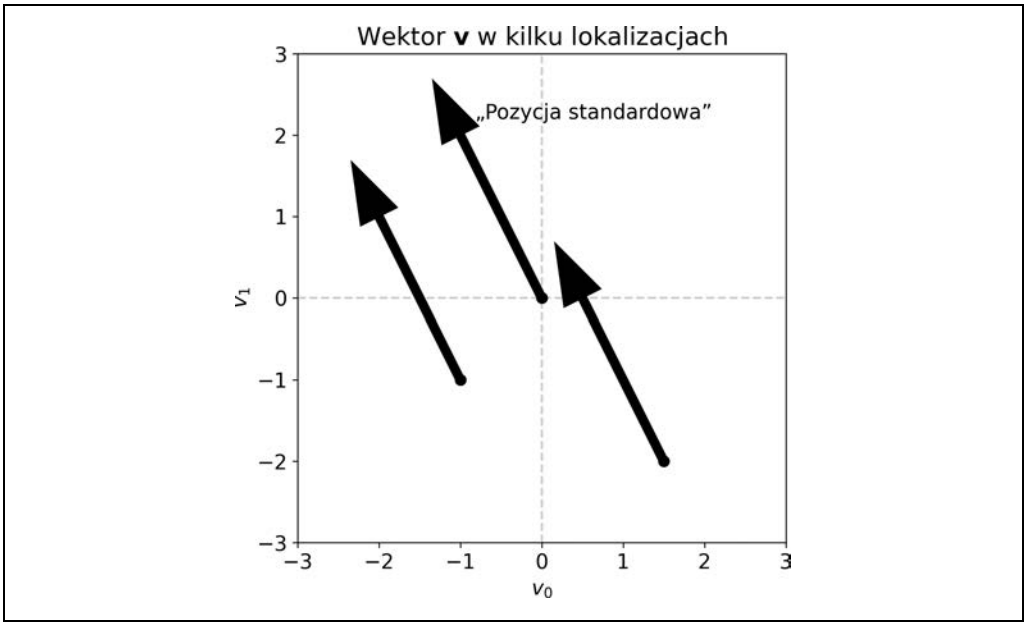
```
asList:  (3,)
asArray: (3,)
rowVec:  (1, 3)
colVec:  (3, 1)
```

Na powyższym listingu widać, że jednowymiarowa tablica `asArray` ma rozmiar `(3)`. Zorientowane wektory są zawsze tablicami dwuwymiarowymi. W zależności od orientacji ich rozmiar to `(1,3)` lub `(3,1)`. Wymiary są zawsze podawane jako `(liczba wierszy, liczba kolumn)`.

Geometryczna interpretacja wektorów

Z algebraicznego punktu widzenia wektor to *uporządkowana lista liczb*. Geometrycznie wektor to odcinek linii prostej o określonej długości (nazywanej również **modułem**), kierunku i zwrocie (**kącie pochylenia**; kąt określa się względem dodatniej półosi osi x). Dwa punkty na krańcach wektora nazywane są początkiem lub punktem zaczepienia (miejsce, w którym zaczyna się wektor) i końcem (miejsce, w którym wektor się kończy). Koniec wektora rysuje się często w formie grotu strzałki, aby można go było odróżnić od początku.

Może Ci się wydawać, że wektor to zbiór współrzędnych geometrycznych, ale tak naprawdę wektory i współrzędne to dwie różne rzeczy. Taka interpretacja daje poprawne wyniki tylko wtedy, gdy wektor jest zaczepiony w początku układu współrzędnych („pozycja standardowa”), co pokazałem na rysunku 2.1.



Rysunek 2.1. Wszystkie strzałki reprezentują ten sam wektor. W położeniu standardowym początek wektora znajduje się w początku układu współrzędnych, a koniec — w punkcie określonym przez jego elementy

Geometryczne i algebraiczne interpretacje wektorów pozwalają rozwinąć intuicję w różnych zastosowaniach, ale *de facto* są to dwie strony tej samej monety. Na przykład interpretacja geometryczna przydaje się w fizyce i inżynierii (np. przedstawianie sił fizycznych), a algebraiczna — w analizie danych (np. przechowywanie danych dotyczących sprzedaży w czasie). Często koncepcje algebry liniowej prezentuje się w sposób geometryczny na dwuwymiarowych wykresach, a następnie rozszerza się je na przestrzenie o większej liczbie wymiarów za pomocą algebry.

Operacje na wektorach

Wektory są jak rzeczowniki. Są to bohaterowie naszej opowieści o algebrze liniowej. Zabawa w algebrę liniową jest możliwa dzięki czasownikom, czyli działaniom, które tchną życie w nasze postaci. Działania te nazywane są **operacjami**.

Niektóre operacje algebry liniowej są proste i intuicyjne i działają dokładnie tak, jak można by się tego spodziewać (np. dodawanie). Inne są bardziej skomplikowane i do ich wyjaśnienia potrzeba całych rozdziałów (np. rozkład według wartości osobliwych). Zacznijmy od prostych operacji.

Dodawanie dwóch wektorów

Aby dodać do siebie dwa wektory, wystarczy dodać do siebie odpowiadające sobie elementy ich obu. Przykład dodawania wektorów pokazałem w równaniu 2.2.

Równanie 2.2. Dodawanie dwóch wektorów

$$\begin{bmatrix}4\\5\\6\end{bmatrix} + \begin{bmatrix}10\\20\\30\end{bmatrix} = \begin{bmatrix}14\\25\\36\end{bmatrix}$$

Jak zapewne się domyślasz, dodawanie wektorów jest zdefiniowane tylko dla wektorów, które mają tę samą liczbę elementów. Nie można dodać do siebie na przykład wektora z $\mathbb{R}^3$ i wektora z $\mathbb{R}^5$.

Odejmowanie wektorów również przebiega zgodnie z naszymi przewidywaniami i polega na odejmowaniu od siebie odpowiadających sobie elementów. Przykład odejmowania pokazałem w równaniu 2.3.

Równanie 2.3. Odejmowanie dwóch wektorów

$$\begin{bmatrix}4\\5\\6\end{bmatrix} - \begin{bmatrix}10\\20\\30\end{bmatrix} = \begin{bmatrix}-6\\-15\\-24\end{bmatrix}$$

Dodawanie wektorów w Pythonie jest bardzo proste:

```
v = np.array([4,5,6])
w = np.array([10,20,30])
u = np.array([0,3,6,9])
vPlusW = v+w
uPlusW = u+w # Błąd! Niezgodność wymiarów!
```

Czy orientacja wektora ma znaczenie podczas dodawania? Spójrz na równanie 2.4.

Równanie 2.4. Czy można dodać wektor wierszowy do wektora kolumnowego?

$$\begin{bmatrix}4\\5\\6\end{bmatrix} + \begin{bmatrix}10 & 20 & 30\end{bmatrix} = ?$$

Można by pomyśleć, że nie ma żadnej różnicy pomiędzy tym a pokazanym wcześniej przykładem — w końcu oba wektory mają po trzy elementy. Zobaczmy, co zrobi Python:

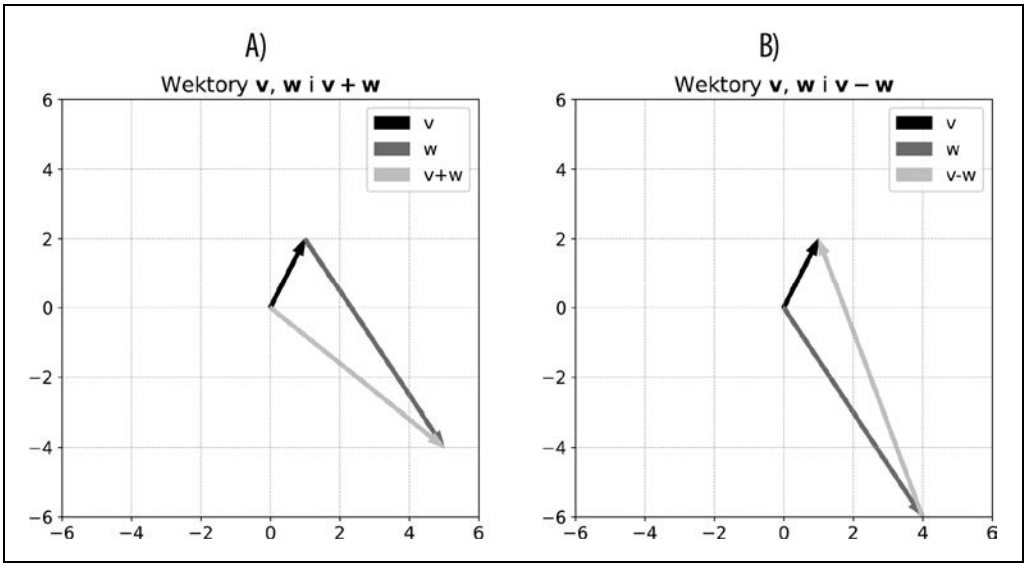
```
v = np.array([[4,5,6]]) # wektor wierszowy
w = np.array([[10,20,30]]).T # wektor kolumnowy
v+w

>> array([[14, 15, 16],
          [24, 25, 26],
          [34, 35, 36]])
```

Wynik może wydawać się mylący i niespójny z podaną wcześniej definicją dodawania wektorów. W Pythonie zaimplementowano operację nazywaną broadcastingiem. Więcej na jego temat przeczytasz w dalszej części tego rozdziału, ale zachęcam Cię do poświęcenia chwili na przyjrzenie się temu wynikowi i zastanowienie się, w jaki sposób powstał on z dodania do siebie wektora wierszowego i kolumnowego. Niezależnie od tego przykład ten pokazuje, że orientacja wektora ma znaczenie: *dwa wektory można dodać do siebie tylko wtedy, gdy mają tę samą liczbę elementów i tę samą orientację.*

Geometryczne dodawanie i odejmowanie wektorów

Aby dodać do siebie dwa wektory w sposób geometryczny, umieść je tak, aby koniec jednego wektora znajdował się w początku drugiego. Wektor będący wynikiem dodawania zaczyna się w początku pierwszego wektora i kończy w końcu drugiego (część A rysunku 2.2). Możesz rozszerzyć tę procedurę na sumowanie dowolnej liczby wektorów: po prostu ułóż wszystkie wektory w taki sposób, żeby koniec poprzedniego był początkiem następnego. Suma tych wektorów to odcinek biegnący od początku pierwszego wektora do końca ostatniego.



Rysunek 2.2. Suma i różnica dwóch wektorów

Geometryczne odejmowanie wektorów przebiega nieco inaczej, ale również w bardzo prosty sposób. Ustaw dwa wektory tak, aby ich początki znajdowały się w tym samym punkcie (można to łatwo zrobić, umieszczając oba wektory w pozycji standardowej); wektor będący wynikiem odejmowania to odcinek biegnący od końca wektora odejmowanego do końca wektora, od którego odejmujemy (część B rysunku 2.2).

Nie lekceważ znaczenia geometrii odejmowania wektorów, jest ona bowiem podstawą ortogonalnego rozkładu wektora, który z kolei jest podstawą liniowej metody najmniejszych kwadratów, czyli jednego z najważniejszych zastosowań algebry liniowej w nauce i inżynierii.

Mnożenie wektorów przez skalar

W algebrze liniowej **skalar** to samodzielna liczba, która nie znajduje się wewnątrz wektora lub macierzy. Zazwyczaj skalary oznacza się małymi greckimi literami, takimi jak α lub λ. Z tego powodu mnożenie wektora przez skalar zapisuje się na przykład jako $\beta \boldsymbol{u}$.

Mnożenie wektora przez skalar jest bardzo proste i polega na pomnożeniu każdego elementu wektora przez skalar. Myślę, że do zrozumienia tej operacji wystarczy jeden przykład (równanie 2.5).

Równanie 2.5. Mnożenie wektora przez skalar (lub skalara przez wektor)

$$\lambda = 4,\ \mathbf{w} = \begin{bmatrix} 9 \\ 4 \\ 1 \end{bmatrix},\ \lambda\mathbf{w} = \begin{bmatrix} 36 \\ 16 \\ 4 \end{bmatrix}$$

Wektor zerowy

Wektor składający się z samych zer nazywamy **wektorem zerowym** i oznaczamy pogrubionym zerem (**0**). W algebrze liniowej wektor ten ma szczególne znaczenie. Rozwiązanie problemu wykorzystujące wektor zerowy jest często nazywane **rozwiązaniem trywialnym**. Rozwiązanie to często się odrzuca. Algebra liniowa jest pełna takich stwierdzeń jak „znajdź niezerowy wektor będący rozwiązaniem…" lub „znajdź nietrywialne rozwiązanie…".

Wspomniałem wcześniej, że typ danych wykorzystywany do przechowywania wektora czasami ma znaczenie, a czasami nie. Przykładem, w którym jest on istotny, jest mnożenie wektorów przez skalar:

```
s= 2
a = [3,4,5] # lista
b = np.array(a) # tablica NumPy
print(a*s)
print(b*s)

>> [ 3, 4, 5, 3, 4, 5 ]
>> [ 6 8 10 ]
```

W powyższym kodzie zdefiniowałem skalar (zmienna `s`) oraz wektor w postaci listy (zmienna `a`). Następnie przekonwertowałem listę na tablicę NumPy (zmienna `b`). W Pythonie operator gwiazdki jest przeciążony, co oznacza, że jego zachowanie zależy od typu zmiennej. Mnożenie listy przez skalar zwielokrotnia zawartość listy `s` razy (w tym przypadku dwa razy), co zdecydowanie *nie* odpowiada mnożeniu wektorów przez skalar z algebry liniowej. Jednak gdy wektor jest przechowywany w postaci tablicy NumPy, gwiazdka jest interpretowana jako mnożenie odpowiadających sobie elementów. (Oto małe ćwiczenie: co się stanie, gdy zdefiniuję `s` jako `2.0` i dlaczego?[1]). Obie te operacje (powtarzanie zawartości listy i mnożenie wektorów przez skalar) są używane w praktyce. Pamiętaj więc o różnicach pomiędzy nimi.

Dodawanie wektorów i skalarów

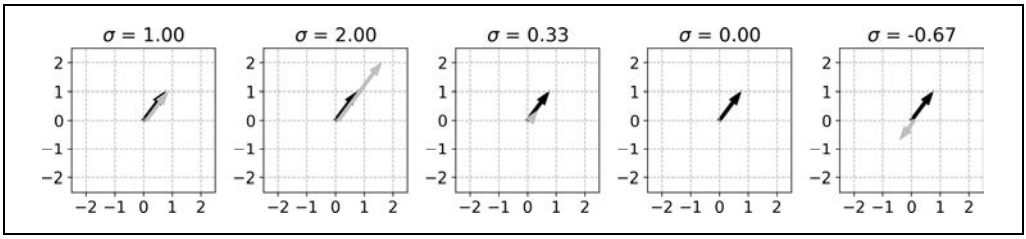
W algebrze liniowej dodawanie do siebie wektorów i skalarów nie jest formalnie zdefiniowane. Są to dwa różne rodzaje obiektów matematycznych i nie można ich ze sobą łączyć. Natomiast programy do obliczeń numerycznych, takie jak Python, pozwalają dodawać do siebie skalary i wektory w sposób podobny do tego, w jaki przeprowadzane jest mnożenie wektora przez skalar. W operacji tej skalar jest po prostu dodawany do każdego elementu wektora. Koncepcję tę pokazałem na poniższym listingu:

[1] Wyrażenie `a*s` zwróci błąd, ponieważ zwielokrotnianie listy jest możliwe tylko przy użyciu liczb całkowitych. Nie da się powtórzyć zawartości listy 2,72 raza!

```
s= 2
v = np.array([3,6])
s+v
>> [5 8]
```

Geometria mnożenia wektor – skalar

Dlaczego skalary nazywane są „skalarami"? Nazwa ta pochodzi od ich interpretacji geometrycznej. Skalary skalują wektory bez zmiany ich kierunku. Mnożenie wektora przez skalar skutkuje jednym z czterech efektów. w zależności od tego. czy skalar jest większy niż 1, ma wartość z przedziału od 0 do 1, ma wartość 0 czy jest ujemny. Efekty te pokazałem na rysunku 2.3.



Rysunek 2.3. Ten sam wektor (czarna strzałka) pomnożony przez różne skalary σ (wynikowy wektor został zaznaczony na szaro i odrobinę przesunięty, aby był lepiej widoczny)

Wspomniałem wcześniej, że skalary nie zmieniają kierunku wektora. Natomiast na rysunku widać, że po pomnożeniu wektora przez ujemny skalar jego zwrot zmienił się na przeciwny (to znaczy wektor obrócił się o 180°). Kierunek wektora jest określony przez prostą przechodzącą przez początek i koniec wektora (w następnym rozdziale nazwę tę prostą podprzestrzenią jednowymiarową). W tym sensie „obrócony" wektor nadal wskazuje tę samą prostą, a zatem ujemny skalar nie zmienił jego kierunku. Ta interpretacja ma znaczenie w kontekście przestrzeni macierzowych oraz wektorów własnych i osobliwych. Wszystkie je omówię w kolejnych rozdziałach.

Mnożenie wektorów przez skalar w połączeniu z dodawaniem wektorów prowadzi do **uśredniania wektorów**. Uśrednianie wektorów przebiega tak samo jak liczenie średniej i polega na dodaniu ich do siebie oraz podzieleniu wyniku przez liczbę składników sumy. A zatem aby uśrednić dwa wektory, należy je do siebie dodać, a następnie wynik pomnożyć przez skalar 0,5. Ogólnie rzecz biorąc, aby uśrednić N wektorów, należy je do siebie dodać i pomnożyć wynik przez $^1/_N$.

Transpozycja

Poznałeś już operację transpozycji: pozwala ona zmienić wektory kolumnowe w wektory wierszowe i odwrotnie. Podam teraz nieco bardziej formalną definicję, która pozwala transponować również macierze (rozdział 5.).

Macierz składa się z wierszy i kolumn. Każdy element macierzy ma indeks (*wiersz, kolumna*). Operacja transpozycji po prostu zamienia te indeksy miejscami. Formalną definicję tej zamiany znajdziesz w równaniu 2.6.

Równanie 2.6. Operacja transpozycji

$$\mathbf{m}_{i,j}^{\mathrm{T}} = \mathbf{m}_{j,i}$$

W zależności od orientacji wektory mają jeden wiersz lub jedną kolumnę. Na przykład wektor wierszowy z przestrzeni sześciowymiarowej zawiera indeks $i = 1$, natomiast indeks j zmienia się w nim od 1 do 6. Sześciowymiarowy wektor kolumnowy ma indeks $j = 1$, natomiast indeks i zmienia się w nim od 1 do 6. A zatem zamiana indeksów i, j miejscami spowoduje zamianę wierszy i kolumn.

Uwaga: dwukrotna transpozycja przywraca wektorowi jego pierwotną orientację. Innymi słowy $\mathbf{v}^{\mathrm{TT}} = \mathbf{v}$. Choć może się to wydawać oczywiste i trywialne, tożsamość ta jest kluczowym elementem kilku ważnych dowodów w nauce o danych i uczeniu maszynowym. Pozwala ona między innymi tworzyć symetryczne macierze kowariancji poprzez mnożenie macierzy danych przez jej transpozycję (co z kolei jest powodem, dla którego analiza głównych składowych jest ortogonalnym obrotem w przestrzeni danych… nie martw się, to zdanie zyska sens w dalszej części książki!).

Broadcasting w Pythonie

Broadcasting to operacja, która istnieje tylko w nowoczesnej algebrze liniowej wykonywanej na komputerach. Nie jest to procedura, którą można znaleźć w tradycyjnym podręczniku do algebry liniowej.

Broadcasting polega na wielokrotnym powtórzeniu operacji między jednym wektorem a każdym elementem innego wektora. Spójrz na następujący ciąg równań:

$$\begin{bmatrix}1 & 1\end{bmatrix} + \begin{bmatrix}10 & 20\end{bmatrix}$$

$$\begin{bmatrix}2 & 2\end{bmatrix} + \begin{bmatrix}10 & 20\end{bmatrix}$$

$$\begin{bmatrix}3 & 3\end{bmatrix} + \begin{bmatrix}10 & 20\end{bmatrix}$$

Zwróć uwagę na wzór w wektorach. Możemy zwięźle zaimplementować ten ciąg przekształceń, kondensując te elementy w wektorach [1 2 3] i [10 20], które następnie dodamy do siebie z wykorzystaniem broadcastingu. Oto jak obliczenia te wyglądałyby w Pythonie:

```
v = np.array([[1,2,3]]).T # wektor kolumnowy
w = np.array([[10,20]])   # wektor wierszowy
v + w # dodawanie z użyciem broadcastingu

>> array([[11, 21],
          [12, 22],
          [13, 23]])
```

Jest to kolejny przykład, w którym orientacja ma znaczenie: spróbuj uruchomić powyższy kod, zmieniając `v` w wektor wierszowy i `w` w wektor kolumnowy[2].

Ponieważ broadcasting umożliwia wydajne i zwarte obliczenia, często wykorzystuje się go w obliczeniach numerycznych. W tej książce zobaczysz kilka przykładów broadcastingu. Wykorzystam go na przykład w podrozdziale dotyczącym klasteryzacji za pomocą algorytmu *k*-średnich (rozdział 4.).

[2] W tym przypadku Python również przeprowadzi broadcasting, ale wynik będzie macierzą 3×2, a nie 2×3.

Moduł wektora i wektory jednostkowe

Moduł wektora (nazywany także **długością geometryczną** lub **normą**) to odległość od początku do końca wektora. Oblicza się go za pomocą standardowego wzoru na odległość w przestrzeni euklidesowej, czyli jako pierwiastek kwadratowy z sumy kwadratów elementów wektora (równanie 2.7). Moduł wektora oznacza się za pomocą symbolu składającego się z podwójnych pionowych kresek umieszczonych po obu stronach nazwy wektora: $\| \mathbf{v} \|$.

Równanie 2.7. Norma wektora

$$\| \mathbf{v} \| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

W niektórych zastosowaniach wykorzystuje się kwadrat modułu (oznaczany jako $\| \mathbf{v} \|^2$). W tym przypadku po prawej stronie powyższej formuły nie występuje pierwiastek kwadratowy.

Zanim pokażę Ci kod w Pythonie, pozwól, że wyjaśnię jeszcze kilka rozbieżności terminologicznych pomiędzy algebrą liniową uprawianą „na tablicy" a algebrą liniową w Pythonie. W matematyce wymiarowość wektora to liczba elementów, które się w nim znajdują, a długość wektora to odległość w sensie geometrycznym. Natomiast w Pythonie funkcja `len()` (gdzie `len` jest skrótem od ang. *length* — „długość") zwraca *wymiarowość* tablicy, podczas gdy funkcja `np.norm()` oblicza długość geometryczną (moduł). Aby uniknąć nieporozumień, w tej książce będę używał terminu „moduł" (lub „długość geometryczna") zamiast „długość":

```
v = np.array([1,2,3,7,8,9])
v_dim = len(v)   # wymiarowość w sensie matematycznym
v_mag = np.linalg.norm(v) # moduł, długość geometryczna lub norma
```

W niektórych zastosowaniach potrzebujemy wektora o długości geometrycznej równej 1. Wektor ten nazywamy **wektorem jednostkowym**. Wektory jednostkowe wykorzystuje się na przykład w macierzach ortogonalnych, macierzach rotacji, wektorach własnych i wektorach osobliwych.

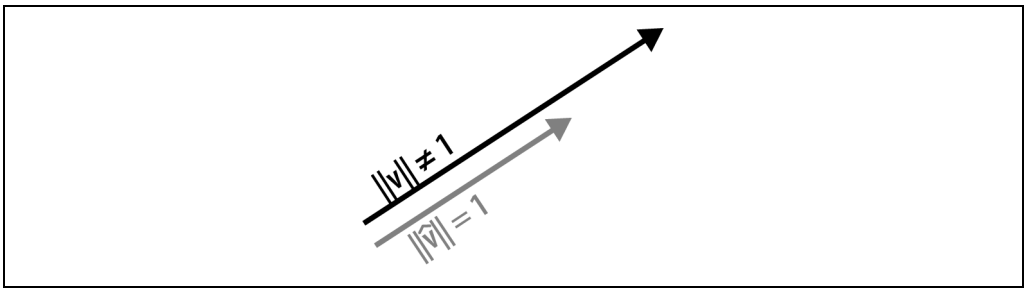
Wektor jednostkowy definiuje się jako $\| \mathbf{v} \| = 1$.

Nie muszę chyba dodawać, że wiele wektorów nie jest wektorami jednostkowymi. (Kusi mnie, aby napisać, że „większość wektorów nie jest wektorami jednostkowymi", ale istnieje nieskończenie wiele wektorów jednostkowych i nieskończenie wiele wektorów niejednostkowych. Co gorsza zbiór tych pierwszych jest równoliczny ze zbiorem tych drugich). Na szczęście z każdym wektorem niejednostkowym związany jest wektor jednostkowy. Oznacza to, że możemy utworzyć wektor jednostkowy o kierunku zgodnym z odpowiadającym mu wektorem niejednostkowym. Utworzenie takiego wektora jednostkowego jest proste. Wystarczy pomnożyć wektor niejednostkowy przez skalar będący odwrotnością jego długości geometrycznej (równanie 2.8).

Równanie 2.8. Tworzenie wektora jednostkowego

$$\hat{\mathbf{v}} = \frac{1}{\| \mathbf{v} \|} \mathbf{v}$$

Przyjęło się, że wektory jednostkowe ($\hat{\mathbf{v}}$) mają ten sam zwrot co powiązane z nimi wektory niejednostkowe ($\mathbf{v}$). Sytuacje tę pokazałem na rysunku 2.4.



Rysunek 2.4. Wektor jednostkowy (szara strzałka) można utworzyć na podstawie wektora niejednostkowego (czarna strzałka); oba mają ten sam kierunek, ale różnią się długością geometryczną

W rzeczywistości twierdzenie, że „z *każdym* wektorem niejednostkowym związany jest wektor jednostkowy", nie jest do końca prawdziwe. Istnieje wektor, którego moduł jest różny od 1 i nie jest z nim związany żaden wektor jednostkowy. Czy potrafisz zgadnąć, jaki to wektor?[3]

W tym podrozdziale nie zamieściłem kodu tworzącego wektory jednostkowe, ponieważ jego napisanie jest jednym z ćwiczeń zamieszczonych na końcu tego rozdziału.

Iloczyn skalarny wektorów

Iloczyn skalarny (czasami nazywany **iloczynem wewnętrznym**) jest jedną z najważniejszych operacji w całej algebrze liniowej. Jest to podstawowy blok obliczeniowy, który wykorzystuje się w wielu operacjach i algorytmach algebry liniowej, m.in. w obliczaniu splotu/konwolucji i korelacji, transformacie Fouriera, mnożeniu macierzy, liniowej ekstrakcji cech oraz filtrowaniu sygnału.

Istnieje kilka sposobów oznaczania iloczynu skalarnego dwóch wektorów. Najczęściej będę stosował dość powszechną notację $\mathbf{a}^T\mathbf{b}$ z powodów, które staną się jasne po zapoznaniu się z mnożeniem macierzy. Czasami iloczyn skalarny wektorów oznacza się również jako $\mathbf{a} \cdot \mathbf{b}$ lub $<\mathbf{a}, \mathbf{b}>$.

Iloczyn skalarny to pojedyncza liczba, która dostarcza nam informacje o relacji między dwoma wektorami. Najpierw skupimy się na algorytmie obliczania iloczynu skalarnego. Następnie wyjaśnię, jak należy interpretować jego wynik.

Aby obliczyć iloczyn skalarny, musisz wymnożyć odpowiadające sobie elementy z dwóch wektorów, a następnie dodać do siebie otrzymane wyniki. W równaniu 2.9 $\mathbf{a}$ i $\mathbf{b}$ są wektorami, $\mathbf{a}_i$ zaś oznacza i-tą składową wektora $\mathbf{a}$.

[3] Wektor zerowy ma długość geometryczną równą 0. Z wektorem tym nie jest związany żaden wektor jednostkowy, ponieważ wektor zerowy nie ma kierunku i nie da się go przeskalować tak, aby jego długość była różna od zera.

Równanie 2.9. Definicja iloczynu skalarnego wektorów

$$\delta = \sum_{i=1}^{n} a_i b_i$$

Z powyższego wzoru możemy wywnioskować, że iloczyn skalarny da się obliczyć tylko między dwoma wektorami o tej samej wymiarowości. Przykład liczbowy pokazałem w równaniu 2.10.

Równanie 2.10. Przykład obliczenia iloczynu skalarnego

$$\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix} \cdot \begin{bmatrix}5 & 6 & 7 & 8\end{bmatrix} = 1 \cdot 5 + 2 \cdot 6 + 3 \cdot 7 + 4 \cdot 8 = 5 + 12 + 21 + 32 = 70$$

Irytujące indeksowanie

Standardowa notacja matematyczna i niektóre zorientowane na matematykę programy do wykonywania obliczeń numerycznych, takie jak MATLAB i Julia, rozpoczynają indeksowanie od 1 i kończą na *N*. Natomiast w niektórych językach programowania, takich jak Python i Java, indeksowanie rozpoczyna się od 0 i kończy na *N* – 1. Nie chcę debatować nad zaletami i ograniczeniami każdego z tych rozwiązań (choć czasami zastanawiam się, ile błędów spowodowała ta niekonsekwencja). Ważne jest, aby pamiętać o tej różnicy podczas tłumaczenia wzorów matematycznych na kod w Pythonie.

Istnieje wiele sposobów implementacji iloczynu skalarnego w Pythonie, a najprostszym sposobem jego obliczenia jest użycie funkcji `np.dot()`:

```
v = np.array([1,2,3,4])
w = np.array([5,6,7,8])
np.dot(v,w)
```

Uwaga dotycząca `np.dot()`

Funkcja `np.dot()` w rzeczywistości nie jest implementacją wektorowego iloczynu skalarnego, lecz implementacją mnożenia macierzy, które jest zbiorem iloczynów skalarnych. Wyjaśni się to, gdy zapoznasz się z zasadami i mechanizmami mnożenia macierzy (rozdział 5.). Jeśli chcesz teraz zbadać działanie tej funkcji, możesz zmodyfikować powyższy kod i nadać obu wektorom orientację (iloczyn wektora wierszowego i kolumnowego). Po kilku eksperymentach odkryjesz, że funkcja zwraca wynik iloczynu skalarnego tylko wtedy, gdy jej pierwszy argument jest wektorem wierszowym, a drugi wektorem kolumnowym.

Oto ciekawa własność iloczynu skalarnego: pomnożenie przez skalar jednego z występujących w nim wektorów spowoduje przeskalowanie iloczynu skalarnego o tę samą wartość. Możesz to sprawdzić, modyfikując powyższy kod:

```
s = 10
np.dot(s*v,w)
```

Iloczyn skalarny `v` i `w` ma wartość 70, a iloczyn skalarny z użyciem `s*v` (co w notacji matematycznej można by zapisać jako $\sigma\mathbf{v}^T\mathbf{w}$) to 700. Spróbuj powtórzyć te obliczenia z ujemnym skalarem, np. `s = -1`. Zobaczysz, że wartość iloczynu skalarnego zostanie zachowana, ale jego znak zmieni się na przeciwny. Oczywiście gdy `s = 0`, to iloczyn skalarny będzie równy zero.

Wiesz już, jak obliczyć iloczyn skalarny. Zastanówmy się, co oznacza ta wartość i jak się ją interpretuje.

Iloczyn skalarny można interpretować jako miarę *podobieństwa* lub *odwzorowanie* między dwoma wektorami. Wyobraź sobie, że masz dane dotyczące wzrostu i wagi 20 osób, które przechowujesz w dwóch wektorach. Z pewnością możemy oczekiwać, że te zmienne są ze sobą w jakiś sposób powiązane (wyższe osoby zwykle ważą więcej), a zatem można by oczekiwać, że iloczyn skalarny tych dwóch wektorów będzie duży. Z drugiej strony wartość iloczynu skalarnego zależy od skali danych, co oznacza, że iloczyn skalarny danych mierzonych w gramach i centymetrach będzie większy niż iloczyn skalarny tych samych danych zapisanych w funtach i stopach. To arbitralne skalowanie można jednak wyeliminować za pomocą współczynnika normalizacji. Znormalizowany iloczyn skalarny dwóch zmiennych nazywamy **współczynnikiem korelacji Pearsona**. Jest to jedna z najważniejszych miar stosowanych w nauce o danych. Więcej na ten temat w rozdziale 4.!

Iloczyn skalarny jest rozdzielny względem dodawania

W matematyce rozdzielność względem dodawania polega na tym, że $a(b + c) = ab + ac$. W kategoriach wektorów i ich iloczynu skalarnego oznacza to, że:

$$\mathbf{a}^T(\mathbf{b} + \mathbf{c}) = \mathbf{a}^T\mathbf{b} + \mathbf{a}^T\mathbf{c}$$

Można powiedzieć, że iloczyn skalarny sumy wektorów jest równy sumie iloczynów skalarnych.

W poniższym kodzie w Pythonie pokazałem rozdzielność iloczynu skalarnego względem dodawania:

```
v = np.array([ 0,1,2 ])
b = np.array([ 3,5,8 ])
c = np.array([ 13,21,34 ])

# iloczyn skalarny jest rozdzielny względem dodawania
res1 = np.dot( a, b+c )
res2 = np.dot( a,b ) + np.dot( a,c )
```

Zmienne `res1` i `res2` zawierają tę samą wartość (dla tych wektorów jest to 110), co ilustruje rozdzielność iloczynu skalarnego względem dodawania. Zwróć uwagę na to, w jaki sposób wzór matematyczny został przetłumaczony na kod w Pythonie. Tłumaczenie wzorów na kod jest ważną umiejętnością w programowaniu zorientowanym na matematykę.

Geometryczna interpretacja iloczynu skalarnego

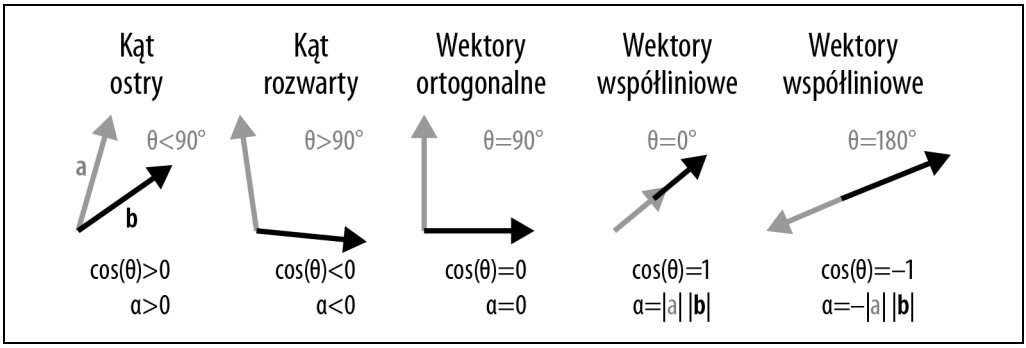
Istnieje również geometryczna definicja iloczynu skalarnego. Zgodnie z nią iloczyn skalarny to iloczyn modułów dwóch wektorów przeskalowany przez cosinus kąta pomiędzy nimi (równanie 2.11).

Równanie 2.11. Geometryczna definicja iloczynu skalarnego wektora

$$\alpha = \cos\left(\theta_{\mathbf{v},\mathbf{w}}\right) \| \mathbf{v} \| \| \mathbf{w} \|$$

Równania 2.9 i 2.11 są sobie równoważne, ale zostały zapisane w dwóch różnych formach. Dowód ich równoważności jest interesującym zadaniem z obszaru analizy matematycznej, ale jego zaprezentowanie wymagałoby około strony tekstu. Dowód ten opiera się na uprzednim udowodnieniu innych zasad, w tym twierdzenia cosinusów. Z punktu widzenia treści tej książki dowód ten nie jest istotny i zdecydowałem się go pominąć.

Zauważ, że moduły wektorów są wielkościami ściśle dodatnimi (z wyjątkiem wektora zerowego, dla którego $\| \mathbf{v} \| = 0$). Natomiast cosinus kąta pomiędzy wektorami to wartość z przedziału od –1 do +1. Oznacza to, że znak iloczynu skalarnego zależy jedynie od geometrycznej relacji pomiędzy wektorami. Rysunek 2.5 przedstawia pięć przypadków wartości znaku iloczynu skalarnego w zależności od kąta między dwoma wektorami (na rysunku zostały pokazane wykresy dwuwymiarowe, ale zasada ta obowiązuje także w przestrzeniach o większej liczbie wymiarów).



Rysunek 2.5. Znak iloczynu skalarnego dwóch wektorów określa zależność geometryczną między wektorami

Zapamiętaj: iloczyn skalarny wektorów ortogonalnych ma wartość zero

Niektórzy nauczyciele matematyki uważają, że nie należy zapamiętywać wzorów i terminów, lecz trzeba rozumieć procedury i dowody. Bądźmy szczerzy: zapamiętywanie jest ważną i nieuniknioną częścią nauki matematyki. Na szczęście algebra liniowa nie wymaga zapamiętania zbyt wielu informacji. Jest jednak kilka rzeczy, które po prostu należy zapamiętać.

Oto jedna z nich: iloczyn skalarny wektorów ortogonalnych ma wartość zero (twierdzenie to działa w obie strony — gdy iloczyn skalarny jakichś wektorów wynosi zero, to te dwa wektory są ortogonalne względem siebie). A zatem następujące stwierdzenia są sobie równoważne: dwa wektory są ortogonalne; iloczyn skalarny dwóch wektorów jest równy zero; dwa wektory przecinają się pod kątem 90°. Powtarzaj tę równoważność, aż trwale zapisze się ona w Twojej pamięci.

Inne sposoby mnożenia wektorów

Iloczyn skalarny jest prawdopodobnie najważniejszym i najczęściej używanym sposobem mnożenia wektorów. Istnieje jednak kilka innych sposobów mnożenia wektorów.

Iloczyn Hadamarda

Iloczyn ten to tylko wymyślne określenie na mnożenie odpowiadających sobie elementów. Implementacja iloczynu Hadamarda polega na pomnożeniu odpowiadających sobie elementów z obu wektorów. Wynik tego iloczynu jest wektorem o tej samej wymiarowości co jego składniki. Na przykład:

$$\begin{bmatrix}5\\4\\8\\2\end{bmatrix} \odot \begin{bmatrix}1\\0\\0{,}5\\-1\end{bmatrix} = \begin{bmatrix}5\\0\\4\\-2\end{bmatrix}$$

W Pythonie do mnożenia odpowiadających sobie elementów wektorów lub macierzy służy operator gwiazdki:

```
a = np.array([5,4,8,2])
b = np.array([1,0,.5])
a*b
```

Spróbuj uruchomić ten kod w Pythonie... Och! Python zwrócił błąd. Znajdź go i spróbuj naprawić. Czego można się nauczyć o iloczynie Hadamarda na podstawie tego błędu? Odpowiedź znajdziesz w przypisie[4].

Iloczyn Hadamarda jest wygodnym sposobem implementacji wielokrotnych mnożeń przez skalar. Wyobraź sobie, że masz dane dotyczące liczby przedmiotów sprzedanych w różnych sklepach oraz ich ceny w każdym z nich. Każdą zmienną możesz przedstawić jako wektor, a następnie obliczyć ich iloczyn Hadamarda. W wyniku otrzymasz przychody ze sprzedaży poszczególnych towarów w *konkretnych sklepach* (jest to coś innego niż łączny przychód ze sprzedaży wszystkich produktów we *wszystkich sklepach*, który można obliczyć jako iloczyn skalarny tych wektorów).

Iloczyn zewnętrzny

Iloczyn zewnętrzny to sposób na utworzenie macierzy z wektora kolumnowego i wierszowego. Każdy wiersz w macierzy będącej wynikiem tego iloczynu jest wynikiem mnożenia wektora wierszowego przez skalar pochodzący z odpowiadającej mu pozycji w wektorze kolumnowym. Możemy również powiedzieć, że każda kolumna wynikowej macierzy jest wynikiem mnożenia wektora kolumnowego przez skalar będący odpowiadającym mu elementem wektora wierszowego. W rozdziale 6. wynik tego mnożenia będę nazywał „macierzą rzędu pierwszego", ale na razie nie martw się znaczeniem tego określenia i skup się na wzorcu przedstawionym w poniższym przykładzie:

$$\begin{bmatrix}a\\b\\c\end{bmatrix}\begin{bmatrix}d & e\end{bmatrix} = \begin{bmatrix}ad & ae\\bd & be\\cd & ce\end{bmatrix}$$

[4] Błąd jest związany z tym, że wektory te mają różną wymiarowość. Pokazuje on, że iloczyn Hadamarda jest zdefiniowany jedynie dla wektorów o tej samej liczbie składowych. Możesz naprawić ten błąd, usuwając jedną wartość z wektora `a` lub dodając jedną składową do wektora `b`.

Litery w algebrze liniowej

Ze szkoły średniej wiesz, że stosowanie liter w roli abstrakcyjnych symboli zastępczych dla liczb pozwala poznać matematykę głębiej niż sama arytmetyka. Ta sama koncepcja ma zastosowanie w algebrze liniowej. Czasami, gdy ułatwia to zrozumienie, wewnątrz macierzy nauczyciele umieszczają litery zamiast liczb. Możesz o nich myśleć tak jak o zmiennych.

Iloczyn zewnętrzny różni się znacznie od iloczynu skalarnego. Jego wynikiem jest macierz, a nie skalar. Dwa wektory będące elementami iloczynu zewnętrznego mogą należeć do przestrzeni o różnej liczbie wymiarów, co jest niedopuszczalne w iloczynie skalarnym, w którym wymiarowość obu wektorów musi być taka sama.

Iloczyn zewnętrzny zapisujemy jako $\mathbf{v}\mathbf{w}^T$ (pamiętaj, że zakładamy, iż wektory są wektorami kolumnowymi, a zatem iloczyn zewnętrzny polega na pomnożeniu kolumny przez wiersz). Zwróć uwagę na subtelną, ale ważną różnicę między zapisem iloczynu skalarnego ($\mathbf{v}^T\mathbf{w}$) i iloczynu zewnętrznego ($\mathbf{v}\mathbf{w}^T$). Zapis ten może się wydawać dziwny i nieco zagmatwany, ale obiecuję, że zyska on sens, gdy zapoznasz się z mnożeniem macierzy opisanym w rozdziale 5.

Iloczyn zewnętrzny przypomina nieco operację broadcastingu, ale nie jest to samo: *broadcasting* to funkcjonalność języka programowania, która pozwala rozszerzać wektory podczas wykonywania operacji arytmetycznych, takich jak dodawanie, mnożenie i dzielenie, natomiast *iloczyn zewnętrzny* jest procedurą matematyczną służącą do mnożenia dwóch wektorów.

W NumPy możesz obliczyć iloczyn zewnętrzny za pomocą funkcji `np.outer()`. Wyznaczysz go również za pomocą funkcji `np.dot()`, jeśli przekażesz jej w pierwszym argumencie wektor kolumnowy, a w drugim wektor wierszowy.

Iloczyn wektorowy i mieszany

Istnieje kilka innych sposobów mnożenia wektorów, w tym iloczyn wektorowy lub mieszany. Metody te są stosowane w geometrii i fizyce, ale w zastosowaniach związanych z informatyką nie pojawiają się na tyle często, aby poświęcać im więcej uwagi w tej książce. Wspominam o nich tylko po to, abyś był zaznajomiony z ich nazwami.

Ortogonalny rozkład wektora

„Rozkład” wektora lub macierzy oznacza rozbicie tych struktur na wiele prostszych elementów. Celem rozkładów jest ujawnienie informacji „ukrytych” w macierzy. Robi się to, aby ułatwić sobie pracę z macierzą lub skompresować przechowywane w niej dane. Stwierdzenie, że znaczna część algebry liniowej (zarówno abstrakcyjnej, jak i praktycznej) to rozkłady macierzy, nie jest przesadą. Rozkłady macierzy to wielka sprawa.

Pozwól, że wyjaśnię Ci pojęcie rozkładu na dwóch prostych przykładach liczbowych:

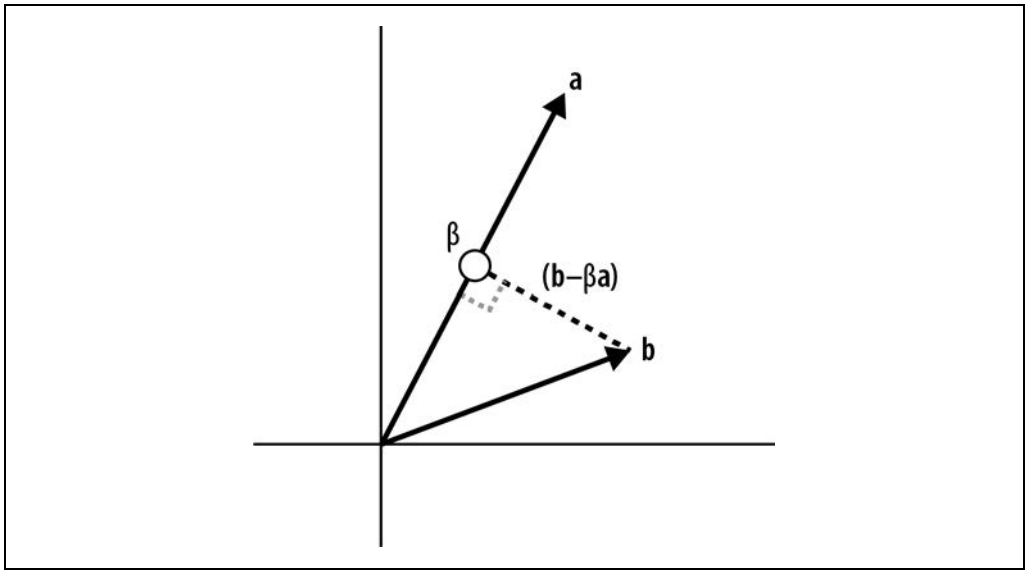
- Liczbę 42,01 możemy rozłożyć na dwie części: 42 i 0,01. Wartość 0,01 może być szumem, który chcę odfiltrować, a może zależy mi na kompresji danych (do zapamiętania liczby 42 w typie

całkowitoliczbowym potrzeba mniej pamięci niż do zapisania zmiennoprzecinkowej wartości 42,01). Niezależnie od motywacji rozkład polega na przedstawieniu jednego obiektu matematycznego w postaci sumy prostszych składników (42 = 42 + 0,01).

- Liczbę 42 możemy też rozłożyć na iloczyn liczb pierwszych 2, 3 i 7. Rozkład ten nazywamy **rozkładem na czynniki pierwsze**. Ma on wiele zastosowań w obliczeniach numerycznych oraz w kryptografii. W tym przykładzie zamiast sumy otrzymujemy iloczyn, ale cel jest nadal ten sam: chcemy rozłożyć jeden obiekt matematyczny na mniejsze, prostsze części.

W tym podrozdziale przyjrzymy się prostemu, ale ważnemu rozkładowi, który polega na rozbiciu wektora na dwa wektory, z których jeden jest prostopadły, a drugi równoległy do wektora referencyjnego. Rozkład ortogonalny wektorów prowadzi bezpośrednio do procedury Grama-Schmidta i rozkładu QR, który często wykorzystuje się do rozwiązywania problemów odwrotnych w statystyce.

Zacznijmy od rysunku, który pozwoli Ci zwizualizować sobie cel rozkładu. Na rysunku 2.6 pokazałem następującą sytuację: mamy dwa wektory, **a** i **b**, w pozycji standardowej i chcemy znaleźć punkt leżący na wektorze **a**, który znajduje się jak najbliżej końca wektora **b**. Problem ten można również wyrazić w kategoriach optymalizacji: szukamy rzutu wektora **b** na wektor **a**, dla którego odległość jest minimalna. Oczywiście punkt znajdujący się na wektorze **a** będzie przeskalowaną wersją **a**. Innymi słowy będzie to $\beta\mathbf{a}$. A zatem naszym celem jest znalezienie skalara β. (Związek tego opisu z ortogonalnym rozkładem wektorów wkrótce stanie się jasny).



Rysunek 2.6. Do znalezienia rzutu punktu znajdującego się w końcu wektora b na wektor a minimalizującego odległość potrzebujemy wzoru na β, dla którego długość wektora b – βa jest możliwie najmniejsza

Co ważne, do zdefiniowania odcinka od **b** do $\beta\mathbf{a}$ możemy wykorzystać odejmowanie wektorów. Moglibyśmy nadać temu odcinkowi własną literę, np. **c**, ale nie zmieni to faktu, że do znalezienia rozwiązania niezbędne będzie odejmowanie.

Kluczowym spostrzeżeniem, które doprowadzi nas do rozwiązania tego problemu, jest to, że punkt leżący na **a**, który jest najbliżej końca **b**, można znaleźć, rysując linię z **b**, która łączy się z **a** pod kątem prostym. Wyobraź sobie trójkąt utworzony przez początek układu współrzędnych oraz końce wektorów **b** i β**a**. Długość odcinka od **b** do β**a** wzrasta, gdy $\measuredangle\beta$**a** staje się mniejszy lub większy niż 90°.

Łącząc ze sobą te informacje, możemy wywnioskować, że **b** – β**a** musi być ortogonalne do β**a**. Stwierdzenie to jest równoważne temu, że wektory te powinny być do siebie prostopadłe. A to oznacza, że ich iloczyn skalarny musi wynosić zero. Przekształćmy te słowa w równanie:

$$\mathbf{a}^{\mathrm{T}}(\mathbf{b} - \beta\mathbf{a}) = 0$$

Teraz możemy zastosować przekształcenia algebraiczne do wyznaczenia z tego równania β (zwróć uwagę, że skorzystałem z rozdzielności iloczynu skalarnego względem dodawania). Przekształcenia pokazałem w równaniu 2.12.

Równanie 2.12. Rozwiązanie problemu znalezienia rzutu ortogonalnego

$$\begin{aligned} \mathbf{a}^{\mathrm{T}}\mathbf{b} - \beta\mathbf{a}^{\mathrm{T}}\mathbf{a} &= 0 \\ \beta\mathbf{a}^{\mathrm{T}}\mathbf{a} &= \mathbf{a}^{\mathrm{T}}\mathbf{b} \\ \beta &= \frac{\mathbf{a}^{\mathrm{T}}\mathbf{b}}{\mathbf{a}^{\mathrm{T}}\mathbf{a}} \end{aligned}$$

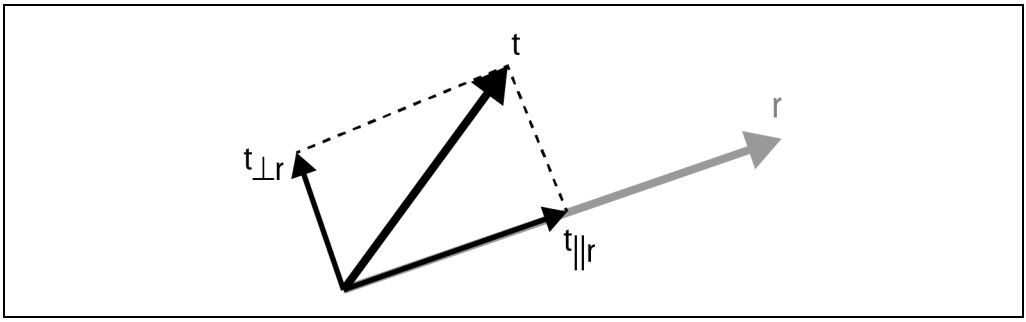
Pięknie: zaczęliśmy od prostego rysunku geometrycznego, zbadaliśmy implikacje praw geometrii, wyraziliśmy je w postaci wzoru, a następnie zastosowaliśmy trochę algebry. W rezultacie otrzymaliśmy wzór na rzutowanie punktu na prostą z zachowaniem minimalnej odległości. Rzut ten nazywa się **rzutem ortogonalnym**. Ma on wiele zastosowań w statystyce i uczeniu maszynowym. Występuje m.in. w słynnej formule najmniejszych kwadratów wykorzystywanej do dopasowywania modeli liniowych (rzuty ortogonalne zobaczysz w rozdziałach 9., 10. i 11.).

Mogę sobie wyobrazić, że bardzo Cię ciekawi, jak wyglądałaby implementacja tego rzutu w Pythonie. Nie pokażę Ci jej, ponieważ jest ona tematem zadania 2.8 z końca tego rozdziału. Jeśli nie możesz się już doczekać końca tego rozdziału, rozwiąż to ćwiczenie teraz, a potem wróć do dalszego opisu rozkładu ortogonalnego.

Być może się zastanawiasz, jak powyższy opis ma się do ortogonalnego rozkładu wektora, czyli tytułu tego podrozdziału. Rzut z minimalną odległością jest niezbędny do jego zrozumienia. Skoro go już znasz, możemy przejść do samego rozkładu.

Jak zwykle zaczynam od sytuacji początkowej i celu. Zaczynamy od dwóch wektorów, które będę nazywał „wektorem docelowym" i „wektorem referencyjnym". Naszym celem jest rozłożenie wektora docelowego na dwa inne wektory w taki sposób, że (1) suma tych dwóch wektorów daje wektor docelowy oraz że (2) jeden z tych dwóch wektorów jest prostopadły do wektora referencyjnego, a drugi jest do niego równoległy. Sytuację tę pokazałem na rysunku 2.7.

Zanim zaczniemy zabawę w matematykę, ustalmy oznaczenia. Wektor docelowy będę nazywał wektorem **t**, a wektor referencyjny wektorem **r**. Dwa wektory utworzone z wektora docelowego będą nazywane *składową prostopadłą* ($\mathbf{t}_{\perp\mathbf{r}}$) oraz *składową równoległą* ($\mathbf{t}_{\parallel\mathbf{r}}$).



Rysunek 2.7. Ilustracja ortogonalnego rozkładu wektora: rozkładam wektor t na sumę dwóch innych wektorów, z których jeden jest ortogonalny, a drugi równoległy do wektora r

Zaczynam od zdefiniowania składowej równoległej. Czym jest wektor równoległy do **r**? Cóż, każda przeskalowana wersja **r** jest oczywiście równoległa do **r**. A zatem aby znaleźć $\mathbf{t}_{\|\mathbf{r}}$, wystarczy po prostu zastosować wzór na rzut ortogonalny, który właśnie odkryliśmy (równanie 2.13).

*Równanie 2.13. Znajdowanie składowej **t** równoległej do **r***

$$\mathbf{t}_{\|\mathbf{r}} = \mathbf{r}\frac{\mathbf{t}^{\mathrm{T}}\mathbf{r}}{\mathbf{r}^{\mathrm{T}}\mathbf{r}}$$

Zwróć uwagę na subtelną różnicę w stosunku do równania 2.12. W poprzednim równaniu szukaliśmy skalara β, tym razem chcemy znaleźć przeskalowany wektor $\beta\mathbf{r}$.

Mamy już składową równoległą. Jak znaleźć składową prostopadłą? Można to zrobić w jeszcze prostszy sposób, ponieważ wiemy już, że te dwie składowe muszą sumować się do pierwotnego wektora docelowego. A zatem:

$$\begin{aligned}\mathbf{t} &= \mathbf{t}_{\perp\mathbf{r}} + \mathbf{t}_{\|\mathbf{r}} \\ \mathbf{t}_{\perp\mathbf{r}} &= \mathbf{t} - \mathbf{t}_{\|\mathbf{r}}\end{aligned}$$

Innymi słowy, od pierwotnego wektora odejmujemy składową równoległą, a otrzymana reszta to nasza składowa prostopadła.

Czy wyznaczona składowa prostopadła *naprawdę* jest prostopadła do wektora referencyjnego? Tak! Aby to udowodnić, wystarczy pokazać, że iloczyn skalarny składowej prostopadłej i wektora referencyjnego ma wartość zero:

$$\begin{aligned}(\mathbf{t}_{\perp\mathbf{r}})^{\mathrm{T}}\mathbf{r} &= 0 \\ \left(\mathbf{t} - \mathbf{r}\frac{\mathbf{t}^{\mathrm{T}}\mathbf{r}}{\mathbf{r}^{\mathrm{T}}\mathbf{r}}\right)^{\mathrm{T}}\mathbf{r} &= 0\end{aligned}$$

Przekształcenia zastosowane w tym dowodzie są proste, ale bardzo żmudne, dlatego je pominąłem. Zamiast tego proponuję popracować nad ćwiczeniami, bo to pozwoli Ci wyrobić sobie wyczucie za pomocą kodu Pythona.

Mam nadzieję, że polubiłeś ortogonalny rozkład wektorów. Raz jeszcze chciałbym zwrócić Twoją uwagę na ogólną zasadę towarzyszącą rozkładom: rozkład to rozbicie jednego obiektu matematycznego na kombinację innych. Szczegóły dotyczące tego, jak to zrobić, zależą od ograniczeń (w tym

przypadku była to ortogonalność i równoległość względem wektora referencyjnego), co oznacza, że różne ograniczenia (czyli różne cele) mogą prowadzić do różnych rozkładów tego samego wektora.

Podsumowanie

Piękno algebry liniowej polega na tym, że nawet najbardziej wyrafinowane i wymagające dużej mocy obliczeniowej operacje na macierzach składają się z szeregu prostych operacji, z których większość można zrozumieć za pomocą geometrycznej intuicji. Nie lekceważ znaczenia prostych operacji na wektorach, ponieważ to, czego nauczyłeś się w tym rozdziale, będzie stanowić podstawę dla pozostałej części tej książki oraz reszty Twojej kariery *specjalisty w dziedzinie stosowanej algebry liniowej* (którym jesteś, jeśli robisz cokolwiek, co jest związane z nauką o danych, uczeniem maszynowym, sztuczną inteligencją, głębokim uczeniem, przetwarzaniem obrazu, wizją komputerową, statystyką i tak dalej).

Oto najważniejsze przesłania tego rozdziału:

- Wektor to uporządkowana lista liczb umieszczonych w kolumnie lub w rzędzie. Liczbę elementów wektora nazywamy jego wymiarowością. Wektor można przedstawić w postaci odcinka w przestrzeni geometrycznej o liczbie osi równej wymiarowości wektora.
- Kilka operacji arytmetycznych na wektorach (dodawanie, odejmowanie i iloczyn Hadamarda) wykonuje się na odpowiadających sobie elementach.
- Iloczyn skalarny to pojedyncza liczba, która reprezentuje związek między dwoma wektorami o tej samej wymiarowości. Wartość iloczynu skalarnego to suma iloczynów odpowiadających sobie elementów z dwóch wektorów.
- Iloczyn skalarny ortogonalnych wektorów to zero. Z geometrycznego punktu widzenia oznacza, że wektory te są do siebie prostopadłe.
- Ortogonalny rozkład wektora polega na przedstawieniu wektora w postaci sumy dwóch innych wektorów, które są ortogonalne i równoległe do wektora referencyjnego. Wzór tego rozkładu można wyprowadzić z geometrii, ale należy pamiętać, że wyraża on koncepcję rzutu prostokątnego (ortogonalnego).

Ćwiczenia z programowania

Mam nadzieję, że nie będziesz traktować poniższych ćwiczeń jako żmudnej pracy, którą musisz wykonać. Ćwiczenia te są bowiem okazją do szlifowania umiejętności matematycznych i programistycznych oraz sprawdzenia, czy naprawdę rozumiesz materiał z tego rozdziału.

Potraktuj te ćwiczenia również jako trampolinę do dalszego poznawania świata algebry liniowej przy użyciu Pythona. Zmodyfikuj kod tak, aby używał innych liczb, innej liczy wymiarów, różnych orientacji itp. Stwórz własny kod, aby przetestować inne koncepcje przedstawione w tym rozdziale. Co najważniejsze: baw się dobrze i ucz się, bawiąc.

Dla przypomnienia: rozwiązania do wszystkich ćwiczeń znajdziesz pod adresem *https://ftp.helion.pl/przyklady/pralli.zip*.

Ćwiczenie 2.1

W zbiorze kodów „brakuje" kodu odpowiedzialnego za utworzenie rysunku 2.2. (Tak naprawdę go nie brakuje — przeniosłem go do rozwiązania w tym ćwiczeniu). Twoim celem jest więc napisanie kodu, który wygeneruje rysunek 2.2.

Ćwiczenie 2.2

Napisz algorytm, który oblicza normę wektora, tzn. zaimplementuj równanie 2.7 w kodzie. Sprawdź poprawność rozwiązania, używając losowych wektorów o różnych wymiarach i orientacjach. Upewnij się, że wartość zwracana przez Twój kod jest taka sama jak wartość zwracana przez `np.linalg.norm()`. Celem tego ćwiczenia jest doskonalenie umiejętności związanych z indeksowaniem tablic NumPy i tłumaczeniem wzorów na kod. W praktyce łatwiej jest użyć `np.linalg.norm()`.

Ćwiczenie 2.3

Utwórz funkcję, która przyjmuje wektor i zwraca wektor jednostkowy o tym samym kierunku. Co się stanie, gdy przekażesz jej wektor zerowy?

Ćwiczenie 2.4

Wiesz już, jak tworzy się wektory jednostkowe. Co zrobić, aby utworzyć wektor o dowolnej zadanej długości geometrycznej? Utwórz w Pythonie funkcję, która przyjmuje wektor i żądaną długość geometryczną oraz zwraca wektor o tym samym kierunku, którego długość odpowiada drugiemu parametrowi funkcji.

Ćwiczenie 2.5

Utwórz pętlę `for`, która zamienia wektor wierszowy w wektor kolumnowy bez użycia wbudowanych w Pythona funkcji/metod, takich jak `np.transpose()` lub `v.T`. To ćwiczenie pomoże Ci tworzyć i indeksować zorientowane wektory.

Ćwiczenie 2.6

Oto interesujący fakt: kwadrat normy wektora to iloczyn skalarny tego wektora i jego samego. Aby się o tym przekonać, przeanalizuj raz jeszcze równanie 2.8. Następnie potwierdź tę prawidłowość za pomocą Pythona.

Ćwiczenie 2.7

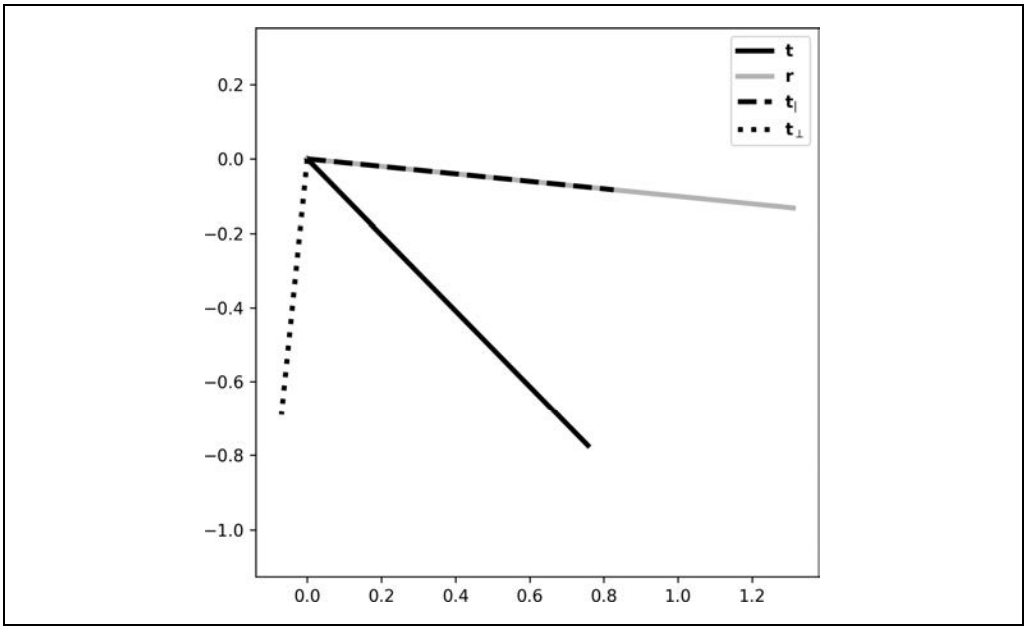
Utwórz kod, który zademonstruje *przemienność* iloczynu skalarnego. Przemienność oznacza, że $a \cdot b = b \cdot a$, co w przypadku iloczynu skalarnego oznacza, że $\mathbf{a}^T\mathbf{b} = \mathbf{b}^T\mathbf{a}$. Po sprawdzeniu prawdziwości tej własności w kodzie przeanalizuj równanie 2.9, aby zrozumieć, dlaczego iloczyn skalarny jest przemienny.

Ćwiczenie 2.8

Utwórz kod, który wygeneruje rysunek 2.6. (O ile znajdą się na nim wszystkie kluczowe elementy, rysunek generowany przez Twoje rozwiązanie nie musi wyglądać *dokładnie* tak jak rysunek 2.6).

Ćwiczenie 2.9

Zaimplementuj ortogonalny rozkład wektora. Zacznij od dwóch wektorów liczb losowych **t** i **r** i spróbuj odtworzyć treść rysunku 2.8 (ze względu na liczby losowe Twój wykres będzie wyglądał nieco inaczej). Następnie potwierdź, że obie składowe sumują się do **t** oraz że $\mathbf{t}_{\perp \mathbf{r}}$ i $\mathbf{t}_{\parallel \mathbf{r}}$ są względem siebie ortogonalne.



Rysunek 2.8. Ćwiczenie 2.9

Ćwiczenie 2.10

Ważną umiejętnością programisty jest umiejętność radzenia sobie z błędami. Załóżmy, że w Twoim kodzie znajduje się błąd polegający na tym, że w mianowniku ułamka z równania 2.13 znalazło się $\mathbf{t}^T\mathbf{t}$ zamiast $\mathbf{r}^T\mathbf{r}$ (łatwo popełnić tego rodzaju błąd i mówię to z własnego doświadczenia zdobytego podczas pisania tego rozdziału!). Zaimplementuj kod zawierający ten błąd i sprawdź, czy wynik obliczony przy jego użyciu różni się od poprawnego. Co możesz zrobić, aby sprawdzić, czy wynik jest poprawny? (W programowaniu porównywanie wartości z własnego kodu ze znanymi wynikami nazywamy **sprawdzaniem poprawności** (ang. *sanity-checking*)).

Skorowidz

A

algebra liniowa, 13
algorytm
  analizy głównych składowych, 239
  Grama-Schmidta, 143, 146
  k-średnich, 57, 60
analiza
  głównych składowych, PCA, 201, 236, 244
    rozkład według wartości osobliwych, 239
  liniowa dyskryminacyjna, LDA, 240, 244, 248
aproksymacja macierzami niskiego rzędu, 227, 242, 244, 252
atrybut shape, 21

B

baza, 48, 51, 52
  standardowa, 49
biblioteka
  Matplotlib, 64, 268
  NumPy, 64, 267
  Pandas, 182
  SciPy, 61, 160
  statsmodels, 186
broadcasting, 27

C

centroid, 58

D

diagonala, 67
diagonalizacja
  jednoczesna dwóch macierzy, 216
  macierzy, 217
  macierzy kwadratowej, 207
długość geometryczna, 28
dodawanie
  macierzy, 68, 95
  wektorów, 22, 24
  wektorów i skalarów, 25
dopasowywanie ogólnego modelu liniowego, 167
dowód matematyczny, 16
dźwignia, leverage, 179

E

eliminacja
  Gaussa, 156
  Gaussa-Jordana, 157, 158

F

filtrowanie, 62
  obrazu, 116
  szeregów czasowych, 56
forma kwadratowa macierzy, 212
f-stringi, 273
funkcja sigmoidalna, 272
funkcje, 264–266

G

geometryczna interpretacja
  iloczynu skalarnego, 31
  odwrotności, 135
  wektorów, 21
  wektorów własnych, 200
GitHub, 260
Google Colab, 260

H

hiperparametry, 193

I

IDE, 259
iloczyn
  Hadamarda, 33, 69
  mieszany, 34
  skalarny, 29, 31, 75
    geometryczna interpretacja, 31
    wektorów ortogonalnych, 32
  wektorowy, 34
  zewnętrzny, 33, 75, 232
implementacja analizy głównych składowych, 247
indeksowanie, 30, 64, 263, 267
iterator, 277

J

jądro
  Gaussa, 116
  lewostronne, 91
  macierzy, 104

K

kąt pochylenia wektora, 21
kernel, 279
klasteryzacja, 57
kod, 16
kombinacja liniowa, 42
komentarze, 264
kompresja
  danych, 202
  obrazów, 243
konwolucja, 116
korelacja, 54, 56, 60
kowariancja, 109, 242
kształt, shape, 117

L

LDA, linear discriminant analysis, 240

M

macierz, 64
  diagonalna, 67, 93
  dodatnio (pół)określona, 215, 217
  dołączona, 129
  dopełnień algebraicznych, 129
  Hilberta, 134
  jednostkowa, 67, 93
  korelacji, 183
  kowariancji danych, 109, 111
  kwadratowa, 67, 199
    diagonalizacja, 207
    określoność, 214
  liczb losowych, 66, 143
  minorów, 128
  nieodwracalna, 125
  niesymetryczna, 76
  niskiego rzędu, 242, 244, 252
  obrotu, 112
  ortogonalna, 112, 140, 146
  osobliwa, 125
    rozkład, 211
  permutacji, 160
  prostokątna, 67
  schodkowa, 151, 155
  schodkowa zredukowana, 157
  skalarna, 203
  specjalna, 66
  symetryczna, 75, 76, 208
  szeroka, 67
  trójkątna, 67, 94
  trójkątna górna, 145
  wysoka, 67
  zależności, 98, 184, 189
  zerowa, 67, 93
  źle uwarunkowana, 230
macierze
  znajdowanie wartości własnych, 203
  znajdowanie wektorów własnych, 205
metoda najmniejszych kwadratów, 168–170, 177, 181
metody, 265
mnożenie
  macierzy, 71, 95
    przez skalar, 69
    przez wektor, 72, 112, 204
    standardowe, 70

wektorów, 32
przez skalar, 24, 26
moduł wektora, 21, 28

N

niezależność liniowa, 43
zbioru wektorów, 98
norma
Frobeniusa, 83, 84, 103
macierzowa, 82, 103
wektorowa, 28, 238
normalizacja, 54

O

obliczanie
odwrotności, 125, 145
stabilność numeryczna, 133
wyznacznika, 100
odbicie Householdera, 147
odejmowanie
macierzy, 68
wektorów, 24
odległość euklidesowa, 58
odwrotność macierzy, 124, 158, *Patrz także* pseudoodwrotność
2 × 2, 126
diagonalnej, 128
geometryczna interpretacja, 135
jednostronna, 125, 129
kwadratowej, 128
lewostronna, 131, 168
pełna, 125
unikalność, 132
ogólny model liniowy, 166, 170, 172, 177
określoność macierzy, 214, 217
operacja transpozycji, 20, 27
operacje
na macierzach, 74, 75
na wektorach, 22
normalizacji, 54
operatory porównania, 275
orientacja wektora, 20
ortogonalizacja Grama-Schmidta, 142
ortogonalne wektory własne, 208
ortogonalność kolumn, 140
ortogonalny rozkład wektora, 34

P

PCA, principal component analysis, 236
pivot, 155
podobieństwo cosinusowe, 54–56
podprzestrzeń, 46, 48, 52
wektorowa, 46
podstawianie wsteczne, 157
procedura Grama-Schmidta, 142, 146, 147
przekątna macierzy, 67
przekształcanie macierzy, 160
przestrzeń macierzy, 103
jądro, 104
jądro lewostronne, 91
kolumnowa, 84
wierszowa, 88
przesuwanie macierzy, 68, 95
kwadratowej, 104
przeszukiwanie siatki, grid search, 191, 194, 198
pseudoodwrotność, 125
Moore'a-Penrose'a, 132, 231, 232
punkt przecięcia, 167
Python, 258
biblioteki, 266
formatowanie wyjścia, 273
funkcje, 264–266
indeksowanie, 263, 267
instrukcje zagnieżdżone, 277
komentarze, 264
komunikat o błędzie, 262, 265, 271
metody, 265
NumPy, 267
operator modulo, 278
operatory porównania, 275
pętle, 277
pomiar czasu obliczeń, 278
przepływ sterowania, 274
słowa kluczowe, 265, 266, 275–277
typy danych, 262
wizualizacje, 268
zmienne, 261

R

redukcja
szumów, 201
wymiarowości, 202

regresja
grzbietowa, 83
lasso, 83
wielomianowa, 188, 197
regularyzacja, 187, 194, 196
rozdzielność względem dodawania, 31
rozkład
ekonomiczny, 144
LU, 151, 160–162
macierzy kwadratowej, 203
macierzy osobliwych, 211
pełny, 144
QR, 142, 145, 176
według wartości osobliwych, SVD, 223–228, 231, 232, 236, 242, 252, 255
analiza głównych składowych, 239
według wartości własnych, 199, 211, 216, 217, 228, 236, 244
wektora, 34
rozpinanie, 46, 48
rozszerzanie macierzy, 97
rozwiązanie
minimalizujące normę, 187
trywialne, 25
rozwiązywanie
problemu najmniejszych kwadratów, 176
układów równań, 152, 154, 162
równania macierzowe, 153
równanie własne, 201
różnica dwóch wektorów, 24
rząd
macierzy, 91, 95, 104, 225, 232
macierzy specjalnych, 93
rzut ortogonalny, 36

S

skalar, 24, 26
sklearn, 247
slicing, 64, 267
słowo kluczowe
def, 265
elif, 276
else, 276
if, 275
return, 266
solver najmniejszych kwadratów, 168
sprawdzanie poprawności, sanity-checking, 40
sprowadzanie do postaci schodkowej, 154, 162
stała, 152, 167
statystyka, 201
suma
dwóch wektorów, 24
kwadratów błędów, 174
SVD, singular value decomposition, 223
symbol $\mathbb{R}$, 19
szeregi czasowe, 56
szerokość, width, 117
szum, 201, 243, 255

Ś

ślad macierzy, 84, 103

T

tabela regresji, 186
tablica, 21
tensor, 118
transpozycja
macierzy, 74, 110
wektorów, 20, 26
tworzenie
funkcji, 265
macierzy, 64
macierzy symetrycznych, 76
ogólnego modelu liniowego, 166
wektora, 19, 21
wektora jednostkowego, 28
typy danych, 262

U

uchwyt, handle, 114
układy równań, 151, 154, 162
unikalność odwrotności, 132
usuwanie szumów, 243, 255
uśrednianie wektorów, 26

W

wariancja, 229
wartości
odstające, outliers, 179
osobliwe, 96, 223–228, 231
zamiana na wariancję, 229
własne, 199, 212, 228

ważona kombinacja liniowa, 42, 51, 52, 72
wektor, 19
  bazowy, 206
  geometryczna interpretacja, 21
  jednostkowy, 28, 29, 90
  kolumnowy, 19
  wierszowy, 19
  zerowy, 25, 45
wektory własne, 200
  ortogonalne, 208
  rzeczywiste, 210
  skala nieokreśloności, 206
  znajdowanie, 203, 205
  znak, 206
wielomian charakterystyczny, 101–104, 204, 217
wizualizacja, 268
  macierzy, 64
  metody najmniejszych kwadratów, 172
  rozkładu LU, 161
  wektorów, 19
wizualna kontrola danych, 193
współczynnik
  korelacji, 54
  korelacji Pearsona, 31, 54
  uwarunkowania macierzy, 134, 230
współliniowość, 187, 195
wybielanie, whitening, 221
wykres osypiska, 201
wykrywanie cech, 56, 62
  na obrazie, 115, 122
wymiarowość, 20
wymiary Q i R, 144
wyraz wolny, 152, 167
wyśrodkowanie zmiennej, 54
wyznacznik
  macierzy, 99, 104
  macierzy osobliwej, 101

Z

zależność liniowa, 101
zastosowania
  macierzy, 109
  metody najmniejszych kwadratów, 181
  rozkładu
    według wartości osobliwych, 236, 243, 255
    według wartości własnych, 236
  rzędów macierzy, 97
  wektorów, 54
zbiory niezależne, 44
zbiór
  rozpinany, 46
  testowy, 250
  wektorów, 41, 52
zintegrowane środowisko programistyczne, IDE, 259
zmienne
  niezależne, 166
  zależne, 166
znajdowanie
  wartości własnych, 203
  wektorów własnych, 205

Notatki

Ta książka świetnie wyjaśnia przydatność algebry liniowej i przedstawia wiele jej praktycznych zastosowań

Thomas Nield, autor książek *Podstawy matematyki w data science* i *Pierwsze kroki z SQL*

Pozornie nie dzieje się nic złego, jeśli inżynier lub analityk danych nie rozumie algebry liniowej. Może korzystać z już istniejących narzędzi i nie przejmować się szczegółami ich implementacji. Warto jednak dokładnie poznać algorytmy używane w nauce o danych i dostosować do swoich potrzeb istniejące metody obliczeniowe, tutaj więc nowoczesna algebra liniowa okazuje się nieodzowna. Jeśli chcesz ją poznać w nowoczesnej, praktycznej formie, najlepiej posłużyć się kodem i zastosowaniem algebry liniowej w analizie danych czy symulacjach numerycznych.

To książka przeznaczona dla osób, które pracują ze zbiorami danych. Jest praktycznym przewodnikiem po koncepcjach algebry liniowej, pomyślanym tak, by ułatwić ich zrozumienie i zastosowanie w użytecznych obliczeniach. Poszczególne zagadnienia przedstawiono za pomocą kodu Pythona, wraz z przykładami ich wykorzystania w nauce o danych, uczeniu maszynowym, uczeniu głębokim, symulacjach i przetwarzaniu danych biomedycznych. Dzięki temu podręcznikowi nauczysz się rachunku macierzowego, poznasz metody rozkładu macierzy LU i QR, a także rozkład według wartości osobliwych. Zapoznasz się też z takimi zagadnieniami jak metoda najmniejszych kwadratów i analiza składowych głównych.

W książce między innymi:

- interpretacja i zastosowania wektorów i macierzy
- rachunek macierzowy
- pojęcie zależności liniowej, rzędu macierzy i macierzy pseudoodwrotnej
- metody rozkładu macierzy (w tym rozkłady LU i QR)
- wyznaczanie wartości własnych, wektorów własnych i wartości osobliwych
- zastosowania algebry liniowej, w tym metoda najmniejszych kwadratów i analiza składowych głównych

Mike X Cohen jest profesorem nadzwyczajnym neuronauki w Instytucie Donders Centrum Medycznego Uniwersytetu im. Radbouda w Nijmegen w Holandii. Od ponad dwudziestu lat uczy programowania, analizy danych, statystyki i powiązanych z nimi zagadnień, jest też autorem wielu podręczników.

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 978-83-289-0261-9
9 788328 902619
Cena: 77,00 zł