O'REILLY®

Podstawy matematyki w data science

Algebra liniowa, rachunek prawdopodobieństwa i statystyka

Thomas Nield

Helion

Tytuł oryginału: Essential Math for Data Science: Take Control of Your Data
with Fundamental Linear Algebra, Probability, and Statistics

Tłumaczenie: Grzegorz Werner

ISBN: 978-83-8322-013-0

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/pomads
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Przedmowa **7**

1. Podstawy matematyki oraz rachunku różniczkowego i całkowego **11**
Teoria liczb 11
Kolejność działań 13
Zmienne 14
Funkcje 15
Sumowanie 19
Potęgowanie 20
Logarytmy 23
Liczba Eulera i logarytmy naturalne 24
Liczba Eulera 24
Logarytmy naturalne 28
Granice 28
Pochodne 30
Pochodne cząstkowe 33
Reguła łańcuchowa 35
Całki 37
Podsumowanie 41
Ćwiczenia 42

2. Prawdopodobieństwo **43**
Zrozumieć prawdopodobieństwo 43
Prawdopodobieństwo a statystyka 45
Matematyka prawdopodobieństw 46
Prawdopodobieństwa łączne 46
Prawdopodobieństwa alternatywne 47
Prawdopodobieństwo warunkowe i twierdzenie Bayesa 48
Łączne i alternatywne prawdopodobieństwa warunkowe 51
Rozkład dwumianowy 52
Rozkład beta 54
Podsumowanie 60
Ćwiczenia 60

3. Statystyka opisowa i wnioskowanie statystyczne 62
Czym są dane? 62
Statystyka opisowa a wnioskowanie statystyczne 64
Populacje, próby i obciążenie 64
Statystyka opisowa 68
Średnia i średnia ważona 68
Mediana 69
Dominanta 71
Wariancja i odchylenie standardowe 71
Rozkład normalny 75
Dystrybuanta odwrotna 81
Standaryzacja Z 82
Wnioskowanie statystyczne 84
Centralne twierdzenie graniczne 84
Przedziały ufności 87
Wartości p 89
Testowanie hipotez 90
Rozkład t: analizowanie małych prób 97
Big data i błąd teksańskiego snajpera 98
Podsumowanie 99
Ćwiczenia 100

4. Algebra liniowa 101
Co to jest wektor? 101
Dodawanie i łączenie wektorów 105
Skalowanie wektorów 106
Powłoka i zależność liniowa 108
Przekształcenia liniowe 110
Wektory bazowe 110
Mnożenie macierzy przez wektor 113
Mnożenie macierzy 117
Wyznaczniki 119
Specjalne rodzaje macierzy 122
Macierz kwadratowa 122
Macierz jednostkowa 122
Macierz odwrotna 122
Macierz diagonalna 123
Macierz trójkątna 123
Macierz rzadka 123
Układy równań i macierze odwrotne 124
Wektory i wartości własne 127
Podsumowanie 129
Ćwiczenia 130

5. Regresja liniowa 131
Podstawowa regresja liniowa 132
Reszty i kwadraty błędu 136
Znajdowanie najlepiej dopasowanej linii 138
Równanie w formie zamkniętej 139
Techniki wykorzystujące macierze odwrotne 140
Metoda gradientu prostego 142
Nadmierne dopasowanie i wariancja 148
Metoda stochastycznego gradientu prostego 149
Współczynnik korelacji 151
Istotność statystyczna 153
Współczynnik determinacji 157
Błąd standardowy estymacji 158
Przedziały przewidywania 159
Podział danych na treningowe i testowe 162
Wielokrotna regresja liniowa 167
Podsumowanie 168
Ćwiczenia 168

6. Regresja logistyczna i klasyfikacja 169
Na czym polega regresja logistyczna? 169
Przeprowadzanie regresji logistycznej 172
Funkcja logistyczna 172
Dopasowywanie krzywej logistycznej 174
Regresja logistyczna z wieloma zmiennymi 179
Logarytm szansy 182
R-kwadrat 185
Wartości p 189
Podziały na dane treningowe i testowe 191
Macierz błędów 192
Twierdzenie Bayesa a klasyfikacja 195
Krzywa ROC/pole pod krzywą 196
Nierównowaga klas 197
Podsumowanie 198
Ćwiczenia 198

7. Sieci neuronowe 200
Kiedy używać sieci neuronowych i uczenia głębokiego? 200
Prosta sieć neuronowa 201
Funkcje aktywacji 204
Propagacja w przód 208

Propagacja wsteczna 213
Obliczanie pochodnych względem wag i biasów 213
Metoda gradientu stochastycznego 217
Używanie scikit-learn 220
Ograniczenia sieci neuronowych i uczenia maszynowego 221
Podsumowanie 224
Ćwiczenie 225

8. Porady zawodowe i droga naprzód ... 226
Nowa definicja data science 227
Krótka historia data science 229
Szukanie przewagi 231
Biegłość w SQL-u 231
Biegłość w programowaniu 233
Wizualizacja danych 236
Znajomość branży 238
Produktywna nauka 239
Praktyk czy doradca? 240
Na co trzeba uważać w pracy związanej z data science? 242
Definicja roli 242
Skupienie organizacyjne i akceptacja 243
Adekwatne zasoby 244
Rozsądne cele 245
Konkurowanie z istniejącymi systemami 246
Twoja rola nie jest tym, czego się spodziewałeś 248
Czy Twoja praca marzeń nie istnieje? 249
Co dalej? 249
Podsumowanie 250

A. Tematy dodatkowe ... 251

B. Odpowiedzi do ćwiczeń ... 269

Skorowidz ... 281

ROZDZIAŁ 1.

Podstawy matematyki oraz rachunku różniczkowego i całkowego

Pierwszy rozdział zaczniemy od przypomnienia, czym są liczby i jak działają zmienne oraz funkcje w układzie kartezjańskim. Następnie omówimy potęgi i logarytmy. Kolejnym krokiem będzie zapoznanie się z dwiema podstawowymi operacjami rachunku różniczkowego i całkowego: różniczkowaniem i całkowaniem.

Zanim zajmiemy się obszarami matematyki stosowanej, takimi jak prawdopodobieństwo, algebra liniowa, statystyka i uczenie maszynowe, zapewne powinniśmy powtórzyć kilka koncepcji z zakresu podstaw matematyki i rachunku różniczkowego i całkowego. Jeśli w tym momencie chcesz rzucić tę książkę i uciec z krzykiem, bez obaw! Obliczanie pochodnych i całek funkcji zaprezentuję w sposób, którego prawdopodobnie nie uczyli Cię na studiach. Będziemy używać Pythona, a nie ołówka i papieru. Nawet jeśli nie znasz się na pochodnych i różniczkach, nadal nie jest do powód do zmartwienia.

Postaram się, żeby dyskusja była zwięzła i praktyczna. Skupię się tylko na tym, co przyda się nam w kolejnych rozdziałach i co można podciągnąć pod szyld „podstawy matematyki".

Nie jest to pełna powtórka z matematyki!

Nie jest to w żadnym wypadku wyczerpujący przegląd matematyki na poziomie szkoły średniej i wyższej. Jeśli tego szukasz, świetną książką jest *No Bullshit Guide to Math and Physics* autorstwa Ivana Savova. Kilka pierwszych rozdziałów zawiera najlepszy przyspieszony kurs matematyki, jaki kiedykolwiek widziałem. Książka *Mathematics 1001* autorstwa dr Richarda Elwesa również zawiera pokaźną ilość wiedzy podzielonej na strawne porcje.

Teoria liczb

Czym są liczby? Obiecuję, że nie będę tu zbytnio filozofować, ale czy liczby nie są zdefiniowanym przez nas konstruktem? Dlaczego mamy cyfry od 0 do 9, a nie więcej lub mniej? Dlaczego mamy ułamki zwykłe i dziesiętne, a nie tylko liczby całkowite? Obszar matematyki, w którym rozmyślamy o liczbach i zastanawiamy się, dlaczego zaprojektowano je w określony sposób, jest znany jako teoria liczb.

Teoria liczb sięga czasów starożytnych, kiedy to matematycy studiowali różne systemy liczbowe, i wyjaśnia, dlaczego zaakceptowaliśmy je w znanej nam dziś postaci. Oto kilka różnych systemów liczbowych, o których mogłeś słyszeć:

Liczby naturalne

Są to liczby 1, 2, 3, 4, 5… i tak dalej. Zbiór ten obejmuje tylko liczby dodatnie i jest najstarszym znanym systemem. Liczby naturalne są tak stare, że już jaskiniowcy prowadzili „dokumentację" poprzez wydrapywanie kresek na kościach i ścianach jaskiń.

Liczby całkowite nieujemne

Do liczb naturalnych z czasem dołączono zero; zbiór ten nazywamy „liczbami całkowitymi nieujemnymi". Babilończycy rozwinęli też przydatny system pozycyjny z pustymi „kolumnami", którego dziś używamy do zapisu liczb większych niż 9, takich jak „10", „1000" lub „1090". Zera wskazują, że w danej kolumnie nie ma żadnej wartości.

Liczby całkowite

Do liczb całkowitych należą liczby naturalne dodatnie i ujemne, a także zero. My uważamy je za coś oczywistego, ale starożytni matematycy traktowali liczby ujemne bardzo podejrzliwie. Kiedy jednak odejmiesz 5 od 3, otrzymasz –2. Jest to przydatne na przykład w finansach podczas mierzenia zysków i strat. W roku 682 n.e. indyjski matematyk Brahmagupta pokazał, dlaczego liczby ujemne są potrzebne do rozwiązywania równań kwadratowych, a liczby całkowite zostały w końcu zaakceptowane.

Liczby wymierne

Każda liczba, którą można wyrazić za pomocą ułamka zwykłego, taka jak $^{2}/_{3}$, to liczba wymierna. Zbiór ten obejmuje również ułamki dziesiętne o skończonym rozwinięciu oraz liczby całkowite, ponieważ je także można wyrazić za pomocą ułamka, na przykład (odpowiednio) $^{687}/_{100} = 6{,}87$ oraz $^{2}/_{1} = 2$. Liczby wymierne szybko uznano za potrzebne, ponieważ nie zawsze da się mierzyć czas, zasoby i inne wielkości w dyskretnych jednostkach. Mleko nie zawsze kupuje się na litry. Czasem trzeba zmierzyć jego ilość w częściach litra. Jeśli będę biegł przez pięć minut, nie zmierzę przebytej odległości w całych kilometrach, skoro przebiegłem $^{9}/_{10}$ kilometra.

Liczby niewymierne

Liczb niewymiernych nie można wyrazić za pomocą ułamka. Zbiór obejmuje słynną liczbę π, pierwiastki kwadratowe niektórych liczb, na przykład $\sqrt{2}$, oraz liczbę Eulera *e*, którą poznamy później. Liczby te mają nieskończenie wiele (niepowtarzających się) cyfr w rozwinięciu dziesiętnym, na przykład 3,141592653589793238462…

Z liczbami niewymiernymi wiąże się ciekawa historia. Grecki matematyk Pitagoras uważał, że wszystkie liczby są wymierne. Wierzył w to tak żarliwie, że stworzył religię, która wyznawała liczbę 10. „Błogosław nam, boska liczbo, która zrodziłaś bogów i ludzi!", modlili się on i jego zwolennicy (dlaczego akurat „10" była tak szczególna, trudno powiedzieć). Legenda głosi, że jeden z jego uczniów, Hippasus, udowodnił, że nie wszystkie liczby są wymierne, na przykładzie pierwiastka kwadratowego z 2. Poważnie naruszyło to system wierzeń Pitagorasa, który nakazał utopić Hippasusa w morzu.

Tak czy owak, obecnie wiemy, że nie wszystkie liczby są wymierne.

Liczby rzeczywiste

Ten zbiór obejmuje zarówno liczby wymierne, jak i niewymierne. W praktyce, kiedy zajmujesz się inżynierią i analizą danych, możesz traktować wszystkie liczby dziesiętne, z którymi pracujesz, jak liczby rzeczywiste.

Liczby zespolone i urojone

Z tym typem liczb spotykamy się, kiedy wyciągamy pierwiastek kwadratowy z liczby ujemnej. Choć liczby urojone i zespolone są istotne w pewnych typach problemów, my raczej będziemy omijać je z daleka.

W data science większość pracy, jeśli nie całą, będziesz wykonywać przy użyciu liczb naturalnych, całkowitych i rzeczywistych. Liczby urojone mogą pojawiać się w bardziej zaawansowanych problemach, takich jak rozkład macierzy, o których wspomnimy w rozdziale 4.

Liczby zespole i urojone

Jeśli chciałbyś dowiedzieć się więcej o liczbach urojonych, na YouTubie znajduje się doskonała lista odtwarzania zatytułowana *Imaginary Numbers are Real* (*https://oreil.ly/bvyIq*).

Kolejność działań

Mam nadzieję, że znasz *kolejność działań*, czyli porządek, w którym rozwiązuje się poszczególne części wyrażenia matematycznego. Dla przypomnienia, najpierw oblicza się składniki w nawiasach, następnie potęgi, a wreszcie wykonuje mnożenie, dzielenie, dodawanie i odejmowanie.

Weźmy na przykład poniższe wyrażenie:

$$2 \cdot \frac{(3+2)^2}{5} - 4$$

Najpierw obliczamy zawartość nawiasu (3 + 2), która jest równa 5:

$$2 \cdot \frac{(\mathbf{5})^2}{5} - 4$$

Następnie obliczamy potęgę, co w tym przypadku oznacza podniesienie do kwadratu obliczonej przed chwilą sumy 5. W rezultacie otrzymujemy 25:

$$2 \cdot \frac{\mathbf{25}}{5} - 4$$

Dalej mamy mnożenie i dzielenie. Kolejność tych działań można zmieniać, ponieważ dzielenie jest równoważne mnożeniu (z wykorzystaniem ułamków). Pomnóżmy 2 przez $\frac{25}{5}$. Otrzymamy $\frac{50}{5}$:

$$\frac{50}{5} - 4$$

Następnie wykonujemy dzielenie 50 przez 5 i otrzymujemy 10:

$$\mathbf{10} - 4$$

Na koniec wykonujemy dodawanie i odejmowanie. Oczywiście 10 – 4 da nam w wyniku 6:

$$10 - 4 = 6$$

I rzeczywiście, gdybyśmy zapisali to wyrażenie w Pythonie, otrzymalibyśmy wartość 6,0, jak pokazano na listingu 1.1.

Listing 1.1. Obliczanie wyrażenia w Pythonie

```
moja_wartosc = 2 * (3 + 2)**2 / 5 - 4
print(moja_wartosc)        # wypisuje 6.0
```

Są to wprawdzie sprawy elementarne, ale mają one kluczowe znaczenie. W kodzie, nawet jeśli uzyskałbyś poprawny wynik bez nawiasów, warto używać ich w złożonych wyrażeniach, aby zachować kontrolę nad kolejnością ewaluacji wyrażeń.

Na listingu 1.2 grupuję część ułamkową w nawiasie, aby oddzielić ją od reszty wyrażenia.

Listing 1.2. Używanie nawiasów w Pythonie w celu zachowania przejrzystości

```
moja_wartosc = 2 * ((3 + 2)**2 / 5) - 4
print(moja_wartosc)        # wypisuje 6.0
```

Choć oba przykłady są technicznie poprawne, drugi jest bardziej czytelny. Jeśli Ty albo ktoś inny wprowadzicie zmiany w kodzie, nawiasy będą nadal wskazywać właściwą kolejność działań. Zapewnia to również dodatkową linię obrony przed usterkami w kodzie.

Zmienne

Jeśli zdarzyło Ci się pisać skrypty w Pythonie lub dowolnym innym języku programowania, z pewnością wiesz, czym jest zmienna. W matematyce *zmienna* jest nazwanym zamiennikiem nieokreślonej lub nieznanej liczby.

Możesz mieć zmienną x reprezentującą dowolną liczbę rzeczywistą i pomnożyć tę zmienną bez deklarowania jej wartości. Na listingu 1.3 przyjmujemy wartość zmiennej x od użytkownika i mnożymy ją przez 3.

Listing 1.3. Mnożenie zmiennej w Pythonie

```
x = int(input("Wprowadź liczbę\n"))
iloczyn = 3 * x
print(iloczyn)
```

Niektóre typy zmiennych mają standardowe nazwy. Jeśli te nazwy i koncepcje są Ci jeszcze nieznane, nie przejmuj się! Niektórzy czytelnicy mogą jednak wiedzieć, że używamy litery theta θ na oznaczenie kątów, a beta β jako parametru w regresji liniowej. Greckie symbole nie byłyby wygodnymi nazwami zmiennych w Pythonie, więc prawdopodobnie nadalibyśmy tym zmiennym nazwy `theta` i `beta`, jak pokazano na listingu 1.4.

Listing 1.4. Nazwy greckich zmiennych w Pythonie

```
beta = 1.75
theta = 30.0
```

Zauważ również, że nazwy zmiennych czasem opatruje się indeksami dolnymi, aby używać kilku egzemplarzy danej nazwy. W praktyce można traktować je jako oddzielne zmienne. Jeśli napotkasz zmienne x_1, x_2 oraz x_3, po prostu traktuj je jako oddzielne zmienne, jak pokazano na listingu 1.5.

Listing 1.5. Zapisywanie zmiennych z indeksami dolnymi w Pythonie

```
x1 = 3    # albo x_1 = 3
x2 = 10   # albo x_2 = 10
x3 = 44   # albo x_3 = 44
```

Funkcje

Funkcje to wyrażenia, które definiują relację między dwiema lub wieloma zmiennymi. Mówiąc ściślej, funkcja przyjmuje *zmienne wejściowe* (nazywane również *dziedzinowymi* lub *niezależnymi*), umieszcza je w wyrażeniu i generuje *zmienną wyjściową* (zwaną też *zależną*).

Spójrz na tę prostą funkcję liniową:

$$y = 2x + 1$$

Dla dowolnej wartości x podstawiamy do wyrażenia tę wartość x, aby obliczyć y. Kiedy $x = 1$, $y = 3$. Kiedy $x = 2$, $y = 5$. Kiedy $x = 3$, $y = 7$ i tak dalej, jak pokazano w tabeli 1.1.

Tabela 1.1. Różne wartości funkcji $y = 2x + 1$

x	2x + 1	y
0	2(0) + 1	1
1	2(1) + 1	3
2	2(2) + 1	5
3	2(3) + 1	7

Funkcje są przydatne, ponieważ modelują przewidywalną relację między zmiennymi, na przykład ilu pożarów y możemy oczekiwać przy temperaturze x. Będziemy używać funkcji liniowych do przeprowadzania regresji liniowych w rozdziale 5.

Inną konwencją zapisu zmiennej niezależnej y jest jawne oznaczenie jej jako funkcji x, na przykład $f(x)$. Zamiast więc zapisywać funkcję w postaci $y = 2x + 1$, możemy również wyrazić ją w następujący sposób:

$$f(x) = 2x + 1$$

Na listingu 1.6 pokazano, jak można zadeklarować funkcję matematyczną i iteracyjnie wywoływać ją w Pythonie.

Listing 1.6. Deklarowanie funkcji liniowej w Pythonie

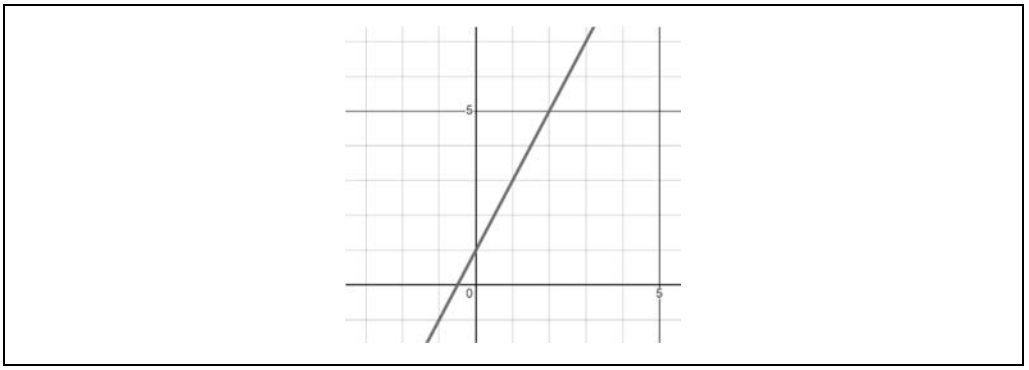
```
def f(x):
    return 2 * x + 1

x_values = [0, 1, 2, 3]

for x in x_values:
    y = f(x)
    print(y)
```

Kiedy pracujemy z liczbami rzeczywistymi, subtelną, ale ważną cechą funkcji jest to, że często mają one nieskończoną liczbę wartości x i wynikowych wartości y. Zadaj sobie pytanie: ile wartości x możemy podstawić do funkcji $y = 2x + 1$? Zamiast ograniczać się do 0, 1, 2, 3…, dlaczego nie użyć wartości 0, 0,5, 1, 1,5, 2, 2,5, 3, jak pokazano w tabeli 1.2?

Tabela 1.2. Różne wartości funkcji $y = 2x + 1$

x	2x + 1	y
0.0	2(0) + 1	1
0.5	2(0.5) + 1	2
1.0	2(1.0) + 1	3
1.5	2(1.5) + 1	4
2.0	2(2.0) + 1	5
2.5	2(2.5) + 1	6
3.0	2(3.0) + 1	7

Czemu nie zwiększać x o jedną czwartą? Albo o jedną dziesiątą? Przyrosty mogą być nieskończenie małe, co pokazuje, że $y = 2x + 1$ jest *funkcją ciągłą*, która generuje wartość y dla każdej możliwej wartości x. Pozwala to zwizualizować naszą funkcję jako linię, jak pokazano na rysunku 1.1.



Rysunek 1.1. Wykres funkcji $y = 2x + 1$

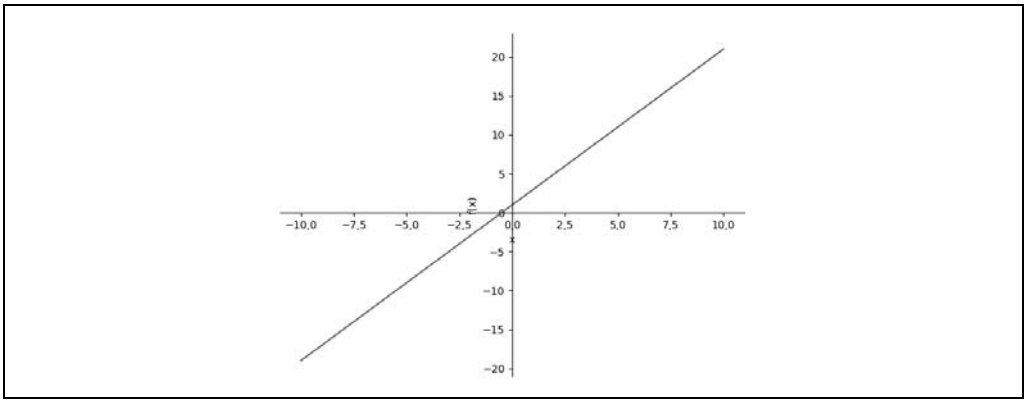
Kiedy tworzymy wykres na dwuwymiarowej płaszczyźnie z dwiema osiami liczbowymi (po jednej na każdą zmienną), nazywamy ją *układem kartezjańskim*, *układem x-y* lub *układem współrzędnych*. Śledzimy daną wartość x, wyszukujemy odpowiednią wartość y i kreślimy przecięcia jako linię. Zauważ, że ze względu na naturę liczb rzeczywistych (czy też, jeśli wolisz, dziesiętnych) istnieje

nieskończona liczba wartości *x*. Dlatego kiedy kreślimy funkcję *f*(*x*), otrzymujemy ciągłą linię bez żadnych przerw. Na linii tej, a także na dowolnej jej części, znajduje się nieskończona liczba punktów.

Gdybyś chciał wykreślić tę funkcję w Pythonie, dostępnych jest wiele bibliotek do tworzenia wykresów, od Plotly do matplotlib. W książce tej do wielu zadań będziemy używać biblioteki SymPy, a pierwszym jej zastosowaniem będzie kreślenie funkcji. SymPy wykorzystuje matplotlib, więc upewnij się, że masz zainstalowany ten pakiet. W przeciwnym razie na Twojej konsoli pojawi się brzydki wykres tekstowy. Następnie po prostu zadeklaruj zmienną *x* w SymPy za pomocą wywołania `symbols()`, zadeklaruj swoją funkcję, a następnie wykreśl ją w sposób pokazany na listingu 1.7 i rysunku 1.2.

Listing 1.7. Kreślenie funkcji liniowej w Pythonie za pomocą SymPy

```
from sympy import *
x = symbols('x')
f = 2*x + 1
plot(f)
```



Rysunek 1.2. Kreślenie funkcji liniowej za pomocą SymPy

Listing 1.8 i rysunek 1.3 to kolejny przykład, tym razem pokazujący funkcję $f(x) = x^2 + 1$.

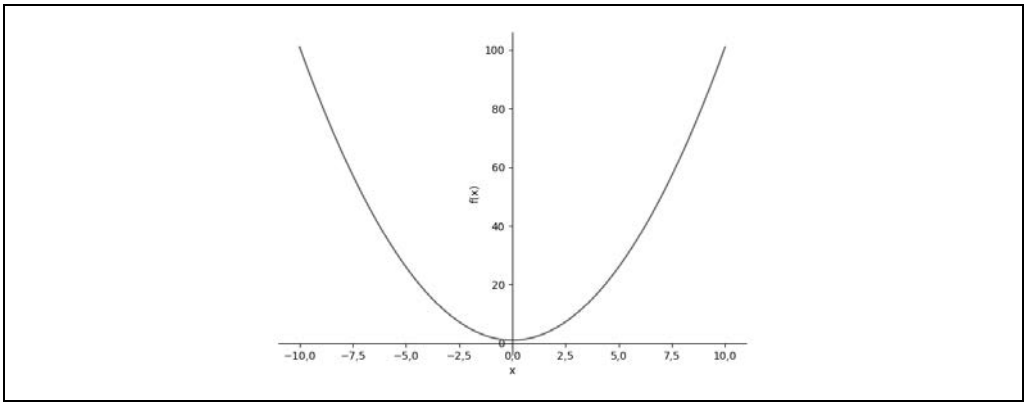
Listing 1.8. Kreślenie funkcji kwadratowej

```
from sympy import *
x = symbols('x')
f = x**2 + 1
plot(f)
```

Zauważ, że na rysunku 1.3. nie otrzymujemy linii prostej, ale gładką, symetryczną krzywą nazywaną parabolą. Jest ciągła, ale nie liniowa, ponieważ generowane przez nią wartości nie leżą na linii prostej. Praca z takimi funkcjami jest matematycznie trudniejsza, ale nauczymy się kilku sztuczek, które nam to ułatwią.

Funkcje krzywoliniowe

Kiedy funkcja jest ciągła, ale jej wykres jest zakrzywiony, a nie prosty i liniowy, nazywamy ją *funkcją krzywoliniową*.



Rysunek 1.3. Kreślenie funkcji kwadratowej za pomocą SymPy

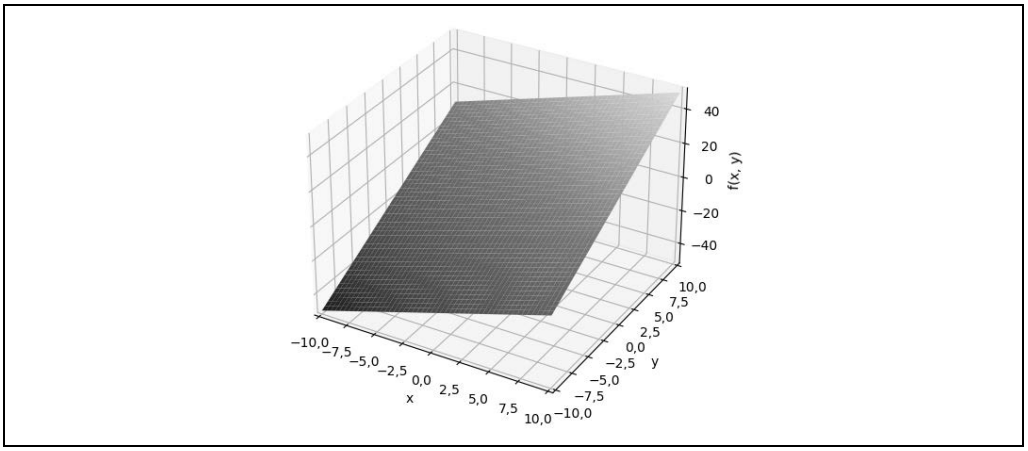
Zauważ, że funkcje mogą mieć wiele zmiennych wejściowych, nie tylko jedną. Możemy na przykład mieć funkcję z niezależnymi zmiennymi x i y. Jak widzisz, zmienna y nie jest zależna jak w poprzednich przykładach.

$$f(x, y) = 2x + 3y$$

Ponieważ mamy dwie zmienne niezależne (x i y) oraz jedną zmienną zależną (wynik $f(x, y)$), musimy utworzyć ten wykres w trzech wymiarach, aby uzyskać płaszczyznę wartości, a nie linię, jak pokazano na listingu 1.9 i rysunku 1.4.

Listing 1.9. Deklarowanie funkcji z dwiema zmiennymi niezależnymi w Pythonie

```
from sympy import *
from sympy.plotting import plot3d

x, y = symbols('x y')
f = 2*x + 3*y
plot3d(f)
```



Rysunek 1.4. Tworzenie trójwymiarowego wykresu funkcji za pomocą SymPy

Bez względu na to, ile masz zmiennych niezależnych, Twoja funkcja zwykle będzie generować tylko jedną zmienną zależną. Rozwiązując równanie z wieloma zmiennymi zależnymi, prawdopodobnie będziesz używać oddzielnej funkcji dla każdej z nich.

Sumowanie

Obiecałem, że w tej książce nie będę używał równań pełnych greckich symboli. Jeden z nich jest jednak tak powszechnie używany i przydatny, że byłbym niesumienny, gdybym go nie omówił. *Sumowanie*, oznaczane literą sigma Σ, oznacza dodawanie wielu elementów.

Na przykład, jeśli chciałbym iterować po liczbach od 1 do 5, pomnożyć każdą z nich przez 2 i zsumować iloczyny, wyraziłbym to za pomocą sumowania w pokazany niżej sposób. Na listingu 1.10 pokazano, jak zapisać to w Pythonie.

$$\sum_{i=1}^{5} 2i = 2(1) + 2(2) + 2(3) + 2(4) + 2(5) = 30$$

Listing 1.10. Sumowanie w Pythonie

```
suma = sum(2*i for i in range(1,6))
print(suma)
```

Zauważ, że `i` to zmienna zastępcza reprezentująca indeks podczas kolejnych iteracji pętli; wartość wskazywaną przez ten indeks mnożymy przez 2, a następnie dodajemy do sumy. Kiedy iterujemy po danych, czasem używamy zmiennych takich jak x_i, które wskazują element kolekcji znajdujący się pod indeksem i.

Funkcja range()

Pamiętaj, że funkcja `range()` w Pythonie jest prawostronnie otwarta, co oznacza, że jeśli wywołasz `range(1,4)`, funkcja będzie iterować po liczbach 1, 2 i 3. Wartość 4 zostanie wykluczona jako górna granica.

Często używa się też zmiennej n na oznaczenie liczby elementów w kolekcji, na przykład rekordów w zbiorze danych. Oto przykład, w którym iterujemy po kolekcji liczb o rozmiarze n. Mnożymy każdą liczbę przez 10 i je sumujemy:

$$\sum_{i=1}^{n} 10x_i$$

Na listingu 1.11 używamy Pythona do wykonania tego wyrażenia na kolekcji czterech liczb. Zwróć uwagę, że w Pythonie (i większości innych języków programowania) zwykle numerujemy elementy od indeksu 0, podczas gdy w matematyce zaczynamy od indeksu 1. Dlatego odpowiednio przesuniemy iterację. Zaczniemy od 0 w wywołaniu `range()`.

Listing 1.11. Sumowanie elementów w Pythonie

```
x = [1, 4, 6, 2]
n = len(x)

suma = sum(10*x[i] for i in range(0,n))
print(suma)
```

Oto sedno sumowania. W skrócie sumowanie Σ oznacza „dodaj do siebie kilka rzeczy” oraz wykorzystuje indeks *i* oraz wartość maksymalną *n* do wyrażenia iteracji składających się na sumę. Będziemy często spotykać je w pozostałej części książki.

Sumowanie w SymPy

Możesz wrócić do tej ramki później, kiedy dowiesz się więcej o SymPy. Pakiet SymPy, którego używamy tu do tworzenia wykresów funkcji, w rzeczywistości jest symboliczną biblioteką matematyczną; dalej w tym rozdziale wyjaśnimy dokładniej, co to znaczy. Zapamiętaj jednak na przyszłość, że operację sumowania w SymPy wykonuje się za pomocą operatora `Sum()`. W poniższym kodzie iterujemy po elementach *i* od 1 do *n*, mnożymy każdy element *i* oraz sumujemy iloczyny. Używamy jednak funkcji `subs()`, aby ustawić *n* na 5, co powoduje zsumowanie wszystkich elementów *i* od 1 do 5:

```
from sympy import *

i,n = symbols('i n')

# iterujemy po elementach i od 1 do n,
# następnie mnożymy je i sumujemy
suma = Sum(2*i,(i,1,n))
# ustawiamy n na 5,
# iterujemy po liczbach od 1 do 5
suma_do_5 = suma.subs(n, 5)
print(suma_do_5.doit()) # 30
```

Zauważ, że sumowanie w SymPy jest „leniwe”, co oznacza, że nie jest automatycznie obliczane ani upraszczane. Używamy zatem funkcji `doit()`, aby wykonać wyrażenie.

Potęgowanie

Potęgowanie polega na mnożeniu liczby przez samą siebie określoną liczbę razy. Kiedy podnosisz dwa do trzeciej potęgi (co zapisuje się jako 2^3, z liczbą 3 w indeksie górnym), oznacza to pomnożenie przez siebie trzech dwójek:

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

Podstawą jest zmienna lub liczba, którą potęgujemy, a *wykładnikiem* liczba, która określa, ile razy mamy pomnożyć podstawę. W wyrażeniu 2^3 liczba 2 jest podstawą, a 3 wykładnikiem.

Wykładniki mają kilka ciekawych właściwości. Przypuśćmy, że chcemy pomnożyć x^2 i x^3. Zobacz, co się dzieje, kiedy zastępuję wykładniki prostym mnożeniem, a następnie konsoliduję czynniki pod jednym wykładnikiem:

$$x^2x^3 = (x \cdot x) \cdot (x \cdot x \cdot x) = x^{2+3} = x^5$$

Kiedy mnożymy potęgi o tej samej podstawie, po prostu dodajemy wykładniki, co określa się nazwą *reguły iloczynowej*. Podkreślę jeszcze raz, że podstawy wszystkich mnożonych potęg muszą być takie same, żeby reguła ta miała zastosowanie.

Zbadajmy teraz dzielenie. Co się dzieje, gdy dzielimy x^2 przez x^5?

$$\frac{x^2}{x^5}$$

$$\frac{x \cdot x}{x \cdot x \cdot x \cdot x \cdot x}$$

$$\frac{1}{x \cdot x \cdot x}$$

$$\frac{1}{x^3} = x^{-3}$$

Jak widzisz, kiedy dzielimy x^2 przez x^5, możemy skreślić dwa „iksy" w liczniku i mianowniku, a otrzymamy $\frac{1}{x^3}$. Kiedy czynnik występuje zarówno w liczniku, jak i mianowniku, możemy skreślić ten czynnik.

A co oznacza x^{-3}? To dobry moment, żeby wprowadzić wykładniki ujemne, które są alternatywnym sposobem zapisu operacji potęgowania w mianowniku ułamka. Na przykład $\frac{1}{x^3}$ to to samo co x^{-3}:

$$\frac{1}{x^3} = x^{-3}$$

Wracając do reguły iloczynowej, widzimy, że ma ona zastosowanie również do wykładników ujemnych. Aby zrozumieć to intuicyjnie, podejdźmy do problemu od innej strony. Możemy wyrazić dzielenie dwóch potęg poprzez zmianę wykładnika „5" w x^5 w liczbę ujemną, a następnie pomnożenie przez x^2. Dodawanie liczby ujemnej jest równoważne odejmowaniu. Można zatem wykorzystać regułę iloczynową do zsumowania wykładników mnożonych potęg, jak pokazano poniżej:

$$\frac{x^2}{x^5} = x^2 \frac{1}{x^5} = x^2 x^{-5} = x^{2+-5} = x^{-3}$$

Wreszcie, czy potrafisz domyślić się, dlaczego dowolna liczba podniesiona do potęgi 0 jest równa 1?

$$x^0 = 1$$

Najłatwiej zrozumieć to poprzez przypomnienie sobie, że dowolna liczba podzielona przez siebie samą daje w wyniku 1. Jeśli mamy ułamek $\frac{x^3}{x^3}$, jest algebraicznie oczywiste, że skraca się on do 1. Ale jednocześnie wyrażenie to sprowadza się do x^0:

$$1 = \frac{x^3}{x^3} = x^3 x^{-3} = x^{3+-3} = x^0$$

Zgodnie z cechą przechodniości, która stwierdza, że jeśli $a = b$ i $b = c$, to $a = c$, możemy stwierdzić, że $x^0 = 1$.

Upraszczanie wyrażeń za pomocą SymPy

Jeśli upraszczanie wyrażeń algebraicznych nie jest Twoją mocną stroną, możesz wykorzystać do tego bibliotekę SymPy. Oto, jak możesz uprościć poprzedni przykład:

```
from sympy import *

x = symbols('x')
wyrazenie = x**2 / x**5
print(wyrazenie) # x**(-3)
```

A co z wykładnikami ułamkowymi? Są one alternatywnym sposobem zapisu pierwiastków, takich jak pierwiastek kwadratowy. Dla przypomnienia, $\sqrt{4}$ oznacza: „Jaka liczba pomnożona przez samą siebie daje 4?", a liczbą tą jest oczywiście 2. Zauważ, że $4^{\frac{1}{2}}$ to to samo co $\sqrt{4}$:

$$4^{\frac{1}{2}} = \sqrt{4} = 2$$

Pierwiastki sześcienne są podobne do kwadratowych, ale oznaczają liczbę, którą trzeba pomnożyć przez siebie trzykrotnie, aby otrzymać wskazany wynik. Pierwiastek sześcienny z 8 oznacza: „Jaka liczba pomnożona trzykrotnie przez samą siebie daje 4?". Jest to liczba 2, ponieważ 2·2·2 = 8. Pierwiastek sześcienny można wyrazić jako wykładnik ułamkowy, to znaczy zapisać $\sqrt[3]{8}$ jako $8^{\frac{1}{3}}$:

$$8^{\frac{1}{3}} = \sqrt[3]{8} = 2$$

Wracając do początku, co się stanie, kiedy trzykrotnie pomnożysz pierwiastek sześcienny z 8? Usunie to pierwiastek sześcienny i da w wyniku 8. Jeśli wyrazimy pierwiastek sześcienny jako potęgę z wykładnikiem ułamkowym $8^{\frac{1}{3}}$, stanie się jasne, że dodajemy do siebie wykładniki, aby otrzymać wykładnik równy 1. To również usuwa pierwiastek sześcienny:

$$\sqrt[3]{8} \cdot \sqrt[3]{8} \cdot \sqrt[3]{8} = 8^{\frac{1}{3}} \cdot 8^{\frac{1}{3}} \cdot 8^{\frac{1}{3}} = 8^{\frac{1}{3}+\frac{1}{3}+\frac{1}{3}} = 8^1 = 8$$

I ostatnia właściwość: potęgowanie potęgi powoduje mnożenie wykładników. Zatem $(8^3)^2$ można uprościć do 8^6:

$$(8^3)^2 = 8^{3 \cdot 2} = 8^6$$

Jeśli nie wiesz, dlaczego tak się dzieje, spróbuj rozwinąć wyrażenie, a przekonasz się, że wynika to z reguły sumowania wykładników:

$$(8^3)^2 = 8^3 \cdot 8^3 = 8^{3+3} = 8^6$$

Wreszcie, co oznacza wykładnik ułamkowy z licznikiem innym niż 1, taki jak $8^{\frac{2}{3}}$? Cóż, oznacza to wyciągnięcie pierwiastka sześciennego z 8, a następnie podniesienie go do kwadratu. Spójrz:

$$8^{\frac{2}{3}} = \left(8^{\frac{1}{3}}\right)^2 = 2^2 = 4$$

Wykładnikami mogą być nawet liczby niewymierne, jak w wyrażeniu 8^{π}, które ma wartość 687,2913. Wydaje się to mało intuicyjne, i nic dziwnego! Aby oszczędzić czas, nie będziemy zajmować się tym bliżej, ponieważ wymaga to pewnej wiedzy z zakresu rachunku różniczkowego i całkowego. Zasadniczo możemy jednak obliczać potęgi z wykładnikiem niewymiernym poprzez przybliżanie ich za pomocą liczby wymiernej. Tak właśnie robią komputery, ponieważ i tak mogą obliczać wynik tylko do ograniczonej liczby miejsc dziesiętnych.

Na przykład π ma nieskończone rozwinięcie dziesiętne. Jeśli jednak weźmiemy pierwszych 11 cyfr, 3,1415926535, możemy przybliżyć π jako liczbę wymierną $^{31415926535}/_{10000000000}$. Rzeczywiście, daje nam to 687,2913, co powinno w przybliżeniu odpowiadać wynikowi obliczonemu przez dowolny kalkulator:

$$8^{\pi} \approx 8^{\frac{31415926535}{10000000000}} \approx 687{,}2913$$

Logarytmy

Logarytm to funkcja matematyczna znajdująca wykładnik, do którego należy podnieść określoną podstawę, aby otrzymać określoną liczbę. Początkowo nie wydaje się szczególnie interesująca, ale w rzeczywistości ma wiele zastosowań. Od mierzenia siły trzęsień ziemi do ustawiania głośności zestawu stereo, logarytmy są wszechobecne. Używa się ich również często w uczeniu maszynowym oraz inżynierii i analizie danych. Logarytmy są na przykład kluczowym aspektem regresji logistycznych omawianych w rozdziale 6.

Zacznij od zadania sobie pytania: „*Do jakiej potęgi* podnieść 2, aby otrzymać 8”? Można wyrazić to matematycznie, używając *x* jako wykładnika:

$$2^x = 8$$

Intuicyjnie znamy odpowiedź, $x = 3$, ale potrzebujemy bardziej eleganckiego sposobu na zapisanie tego działania matematycznego. Właśnie do tego służy funkcja *log*().

$$\log_2 8 = x$$

Jak widać w powyższym wyrażeniu logarytmicznym, mamy podstawę 2 i szukamy potęgi, do której musimy podnieść tę podstawę, aby otrzymać 8. Bardziej ogólnie, możemy zapisać zmienny wykładnik jako logarytm:

$$a^x = b$$

$$\log_a b = x$$

Algebraicznie rzecz biorąc, jest to sposób na wyizolowanie zmiennej *x*, co jest ważne, kiedy chcemy obliczyć *x*. Listing 1.12 pokazuje, jak obliczyć ten logarytm w Pythonie.

Listing 1.12. Używanie funkcji log w Pythonie

```
from math import log

# 2 podniesione do jakiej potęgi daje 8?
x = log(8, 2)

print(x) # wypisuje 3.0
```

Kiedy na platformie takiej jak Python nie określisz podstawy w funkcji `log()`, zwykle używana jest podstawa domyślna. W niektórych dziedzinach, takich jak pomiary trzęsień ziemi, domyślną podstawą logarytmu jest 10. Jednak w data science domyślną podstawą logarytmu jest liczba Eulera *e*. Python używa tej ostatniej, o której porozmawiamy za chwilę.

Podobnie jak potęgi, logarytmy mają kilka właściwości związanych z mnożeniem, dzieleniem, potęgowaniem itd. Aby nie tracić czasu i nie zbaczać z tematu, zaprezentuję je krótko w tabeli 1.3. Kluczową ideą, na której należy się skupić, jest to, że logarytm znajduje dla danej podstawy wykładnik pozwalający uzyskać pewną liczbę.

Tabela 1.3. Właściwości potęg i logarytmów

Działanie	Właściwość potęgi	Właściwość logarytmu
Mnożenie	$x^m \cdot x^n = x^{m+n}$	$log(a \cdot b) = log(a) + log(b)$
Dzielenie	$\frac{x^m}{x^n} = x^m - x^n$	$log\left(\frac{a}{b}\right) = log(a) - log(b)$
Potęgowanie	$(x^m)^n = x^{mn}$	$log(a^n) = n \cdot log(a)$
Wykładnik zerowy	$x^0 = 1$	$log(1) = 0$
Odwrotność	$x^{-1} = \frac{1}{x}$	$log(x^{-1}) = log\left(\frac{1}{x}\right) = -log(x)$

Jeśli chciałbyś zapoznać się bliżej z właściwościami logarytmicznymi, w tabeli 1.3 pokazano obok siebie działania na potęgach i logarytmach, żebyś mógł do niej zaglądać w razie potrzeby.

Liczba Eulera i logarytmy naturalne

Istnieje liczba, która pojawia się często w matematyce, nazywana liczbą Eulera *e*. Jest to specjalna liczba, podobnie jak π, a jej wartość w przybliżeniu wynosi 2,71828. Wszechobecność liczby *e* wynika z tego, że upraszcza ona wiele matematycznych problemów. Opiszemy ją w kontekście potęg i logarytmów.

Liczba Eulera

Kiedy chodziłem do liceum, mój nauczyciel matematyki zademonstrował liczbę Eulera w kilku zadaniach związanych z potęgowaniem. Wreszcie zapytałem: „Panie Nowe, ale czym właściwie jest liczba *e*? Skąd się wzięła?”. Pamiętam, że nigdy nie byłem w pełni zadowolony z wyjaśnień na przykładach populacji królików i innych zjawisk naturalnych. Mam nadzieję, że podane niżej wytłumaczenie będzie bardziej satysfakcjonujące.

Dlaczego tak często używa się liczby Eulera?

Ważną cechą liczby Eulera jest to, że jej funkcja wykładnicza jest pochodną samej siebie, co przydaje się w funkcjach wykładniczych i logarytmicznych. Dowiemy się więcej o pochodnych dalej w tym rozdziale. W wielu zastosowaniach, w których podstawa logarytmu nie ma większego znaczenia, możemy wybrać tę pozwalającą uzyskać najprostszą pochodną, i jest to właśnie liczba Eulera. Dlatego też jest ona podstawą domyślną w wielu funkcjach związanych z data science.

Oto mój ulubiony sposób wyprowadzania liczby Eulera. Przypuśćmy, że pożyczasz komuś 100 złotych z 20-procentowym rocznym oprocentowaniem. Odsetki zwykle nalicza się co miesiąc, więc miesięczna stopa oprocentowania wynosi 0,2:12 = 0,01666. Jakie będzie saldo pożyczki po dwóch latach? Aby uprościć sprawę, załóżmy, że pożyczka nie wymaga spłat (i nie są dokonywane żadne spłaty) przed upływem tych dwóch lat.

Korzystając z omówionych dotychczas właściwości potęgowania (ewentualnie zaglądając do podręcznika finansowości), możemy znaleźć wzór na obliczanie odsetek. Pozwala on obliczyć końcowe saldo *A* przy początkowej inwestycji *P*, stopie oprocentowania *r* oraz przedziale czasu *t* (liczbie lat) podzielonym na *n* okresów (liczba miesięcy w każdym roku). Wzór ten wygląda następująco:

$$A = P \cdot \left(1 + \frac{r}{n}\right)^{nt}$$

Gdybyśmy zatem obliczali odsetki co miesiąc, dług wzrósłby do 148,69 zł, jak pokazano poniżej:

$$A = P \cdot \left(1 + \frac{r}{n}\right)^{nt}$$

$$100 \cdot \left(1 + \frac{0{,}2}{12}\right)^{12 \cdot 2} = 148{,}6914618$$

Jeśli chcesz zrobić to w Pythonie, wypróbuj kod z listingu 1.13.

Listing 1.13. Obliczanie odsetek składanych w Pythonie

```
from math import exp
p = 100
r = .20
t = 2.0
n = 12
a = p * (1 + (r/n))**(n * t)
print(a) # wypisuje 148.69146179463576
```

A co by się stało, gdybyśmy naliczali odsetki codziennie? Zmieńmy *n* na 365:

$$A = P \cdot \left(1 + \frac{r}{n}\right)^{nt}$$

$$100 \cdot \left(1 + \frac{0{,}2}{365}\right)^{365 \cdot 2} = 149{,}1661279$$

Aha! Naliczając odsetki co dzień, a nie co miesiąc, po dwóch latach zarobilibyśmy dodatkowo 47,4666 groszy. Gdybyśmy byli zachłanni, czemu nie mielibyśmy naliczać odsetek co godzinę, jak pokazano niżej? Czy w ten sposób zarobilibyśmy jeszcze więcej? Rok ma 8760 godzin, więc ustawmy *n* na tę wartość:

$$A = P \cdot \left(1 + \frac{r}{n}\right)^{nt}$$

$$100 \cdot \left(1 + \frac{0{,}2}{8760}\right)^{8760 \cdot 2} = 149{,}1817886$$

Wycisnęliśmy w przybliżeniu 2 dodatkowe grosze odsetek! Ale czy doświadczamy malejących zysków? Spróbujmy naliczać odsetki co minutę! Rok ma 525 600 minut, więc ustawmy taką wartość *n*:

$$A = P \cdot \left(1 + \frac{r}{n}\right)^{nt}$$

$$100 \cdot \left(1 + \frac{0{,}2}{525\,600}\right)^{525\,600 \cdot 2} = 149{,}1824584$$

OK, w miarę coraz częstszego naliczania odsetek zyskujemy coraz mniejsze ułamki centa. Gdybyśmy zatem zmniejszali okres w nieskończoność aż do punktu, w którym naliczalibyśmy odsetki w sposób ciągły, do czego by to doprowadziło?

Pozwól, że przedstawię Ci liczbę Eulera *e*, która w przybliżeniu wynosi 2,71828. Oto wzór na „ciągłe" naliczanie odsetek, co oznacza, że naliczamy je bez przerwy:

$$A = P \cdot e^{rt}$$

Wróćmy do naszego przykładu i obliczmy saldo pożyczki po dwóch latach, gdyby odsetki były naliczane w sposób ciągły:

$$A = P \cdot e^{rt}$$

$$A = 100 \cdot e^{0{,}2 \cdot 2} = 149{,}1824698$$

Nie powinno to szczególnie dziwić, zważywszy, że naliczanie odsetek co minutę dało nam saldo 149,1824584. Jest ono bardzo bliskie wartości 149,1824698 uzyskanej w przypadku naliczania odsetek w sposób ciągły.

Aby użyć *e* jako podstawy logarytmu w Pythonie, Excelu i na innych platformach, zwykle korzysta się z funkcji `exp()`. Przekonasz się, że liczba *e* jest używana tak często, że stanowi domyślną podstawę zarówno funkcji wykładniczych, jak i logarytmicznych.

Na listingu 1.14 obliczamy odsetki „ciągłe" w Pythonie za pomocą funkcji `exp()`.

Listing 1.14. Obliczanie odsetek „ciągłych” w Pythonie

```
from math import exp

p = 100   # kapitał, kwota początkowa
r = .20   # stopa procentowa, roczna
t = 2.0   # czas, liczba lat

a = p * exp(r*t)

print(a)  # wypisuje 149,18246976412703
```

Jak zatem wyprowadzić stałą *e*? Porównaj wzór na okresowe naliczanie odsetek z wzorem na naliczanie odsetek w sposób ciągły. Są strukturalnie podobne, ale występują w nich pewne różnice:

$$A = P \cdot \left(1 + \frac{r}{n}\right)^{nt}$$

$$A = P \cdot e^{rt}$$

Mówiąc bardziej technicznie, *e* jest wartością graniczną wyrażenia $\left(1 + \frac{1}{n}\right)^n$, kiedy *n* rośnie, dążąc do nieskończoności. Spróbuj poeksperymentować z rosnącymi wartościami *n*. W miarę ich zwiększania zauważysz coś ciekawego:

$$\left(1 + \frac{1}{n}\right)^n$$

$$\left(1 + \frac{1}{100}\right)^{100} = 2{,}70481382942$$

$$\left(1 + \frac{1}{1000}\right)^{1000} = 2{,}71692393224$$

$$\left(1 + \frac{1}{10000}\right)^{10000} = 2{,}71814592682$$

$$\left(1 + \frac{1}{100000}\right)^{100000} = 2{,}71828169413$$

Zwiększając *n*, odnotowujesz coraz mniejsze zyski i zbliżasz się do wartości 2,71828, czyli *e*. Przekonasz się, że liczby *e* używa się nie tylko do badania populacji i ich wzrostu. Odgrywa ona kluczową rolę w wielu obszarach matematyki.

Dalej w tej książce wykorzystamy liczbę Eulera do budowania rozkładów normalnych w rozdziale 3. i regresji logistycznych w rozdziale 6.

Logarytmy naturalne

Kiedy podstawą logarytmu jest *e*, nazywamy go *logarytmem naturalnym*. W zależności od platformy, do obliczania algorytmu naturalnego czasem używa się funkcji `ln()`, a nie `log()`. Zamiast zatem wyrażać logarytm naturalny jako $\log_e 10$, aby znaleźć potęgę *e*, która daje 10, zapisalibyśmy to skrótowo jako `ln(10)`:

$$\log_e 10 = \ln(10)$$

Jednakże w Pythonie algorytm naturalny oblicza się za pomocą funkcji `log()`. Jak wspomniano wcześniej, domyślną podstawą funkcji `log()` jest *e*. Po prostu pomiń drugi argument, który określa podstawę logarytmu, a funkcja jako podstawy użyje *e*, jak pokazano na listingu 1.15.

Listing 1.15. Obliczanie logarytmu naturalnego liczby 10 w Pythonie

```
from math import log
# e podniesione do jakiej potęgi daje 10?
x = log(10)
print(x)  # wypisuje 2.302585092994046
```

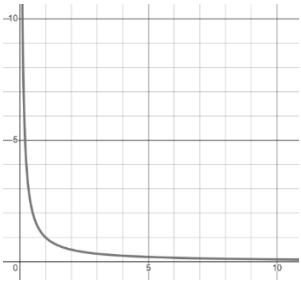
Będziemy używać liczby *e* w wielu miejscach niniejszej książki. Jeśli masz ochotę, poeksperymentuj z potęgami i logarytmami w Excelu, Pythonie, witrynie Desmos.com albo na dowolnej innej platformie matematycznej. Utwórz kilka wykresów i zapamiętaj, jak wyglądają przebiegi tych funkcji.

Granice

Jak widzieliśmy w przypadku liczby Eulera, interesująca bywa sytuacja, w której zmniejszamy albo zwiększamy zmienną wejściową, a zmienna wyjściowa zbliża się do pewnej wartości, ale nigdy jej nie osiąga. Zbadajmy tę koncepcję w sposób formalny.

Przyjrzyj się poniższej funkcji, wykreślonej na rysunku 1.5:

$$f(x) = \frac{1}{x}$$



Rysunek 1.5. Funkcja, która w nieskończoność zbliża się do 0, ale nigdy nie osiąga 0

Przyglądamy się tylko dodatnim wartościom *x*. Zauważ, że w miarę jak *x* rośnie, $f(x)$ dąży do 0. Co ciekawe, $f(x)$ nigdy nie osiąga wartości 0, tylko coraz bardziej się do niej zbliża.

Zatem losem tej funkcji — w miarę jak x dąży do nieskończoności — jest nieustannie zbliżać się do 0, ale nigdy nie osiągnąć tej wartości. Sytuację, w której zbliżamy się do jakiejś wartości, ale nigdy jej nie osiągamy, wyrażamy za pomocą granicy:

$$\lim_{x\to\infty} \frac{1}{x} = 0$$

Czytamy to w następujący sposób: „kiedy x dąży do nieskończoności, funkcja $^1/_x$ dąży do 0 (ale nigdy nie osiąga 0)". Takie „zbliżanie, ale bez dotykania" będziemy spotykać bardzo często, zwłaszcza kiedy zajmiemy się pochodnymi i całkami.

Za pomocą SymPy możemy na przykład obliczyć, do jakiej wartości zbliża się funkcja $f(x) = \frac{1}{x}$, kiedy x dąży do nieskończoności ∞ (listing 1.16). Zauważ, że ∞ zapisuje się w SymPy jako `oo`.

Listing 1.16. Obliczanie granic za pomocą SymPy

```
from sympy import *
x = symbols('x')
f = 1 / x
wynik = limit(f, x, oo)
print(wynik) # 0
```

Jak widziałeś, w ten sposób odkryliśmy liczbę Eulera *e*. Jest ona wynikiem zwiększania w nieskończoność wartości n w poniższej funkcji:

$$\lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = e = 2{,}71828169413\ldots$$

Co ciekawe, kiedy spróbujemy obliczyć wartość *e* za pomocą granic w SymPy (jak pokazano w poniższym fragmencie kodu), SymPy natychmiast rozpozna ją jako liczbę Eulera. Możemy wywołać funkcję `evalf()`, aby wyświetlić ją jako liczbę:

```
from sympy import *
n = symbols('n')
f = (1 + (1/n))**n
wynik = limit(f, n, oo)
print(wynik) # E
print(wynik.evalf()) # 2.71828182845905
```

Potęga SymPy

SymPy (*https://oreil.ly/mgLyR*) to zaawansowany i niezwykle użyteczny system algebry komputerowej (Computer Algebra System, CAS), który wykonuje dokładne obliczenia symboliczne zamiast przybliżonych obliczeń na wartościach dziesiętnych. Przydaje się w sytuacjach, gdy użyłbyś „papieru i ołówka" do rozwiązania zadania matematycznego, i ma tę dodatkową zaletę, że korzysta ze znajomej składni Pythona. Zamiast reprezentować pierwiastek kwadratowy z 2 poprzez przybliżoną wartość 1,4142135623730951, zachowuje go dokładnie jako *sqrt*(2).

Czemu więc nie używać SymPy do wszystkiego, co ma związek z matematyką? Choć będziemy korzystać z tego pakietu w niniejszej książce, trzeba też umieć wykonywać obliczenia matematyczne w Pythonie przy użyciu zwykłych liczb dziesiętnych, ponieważ takie podejście obowiązuje

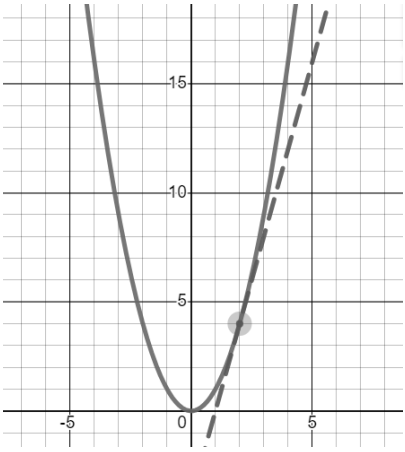
w scikit-learn i innych bibliotekach data science. Komputery znacznie szybciej przetwarzają liczby dziesiętne niż symbole. SymPy zaczyna mieć również problemy, kiedy wyrażenia matematyczne stają się zbyt duże. Trzymaj jednak to przydatne narzędzie pod ręką i nie mów o nim licealistom ani studentom. Mogliby dosłownie wykorzystać je do robienia zadań domowych z matematyki.

Pochodne

Wróćmy do funkcji i przyjrzymy się im z perspektywy rachunku różniczkowego i całkowego, zaczynając od pochodnych. *Pochodna* informuje nas o nachyleniu funkcji i jest przydatną miarą tempa zmian w dowolnym punkcie funkcji.

Dlaczego pochodne są ważne? Często używa się ich w uczeniu maszynowym i innych algorytmach matematycznych, zwłaszcza w metodzie gradientu prostego. Kiedy nachylenie jest równe 0, oznacza to, że osiągnęliśmy minimum lub maksimum zmiennej wyjściowej. Koncepcja ta okaże się przydatna później, gdy zajmiemy się regresją liniową (rozdział 5.), regresją logistyczną (rozdział 6.) i sieciami neuronowymi (rozdział 7.).

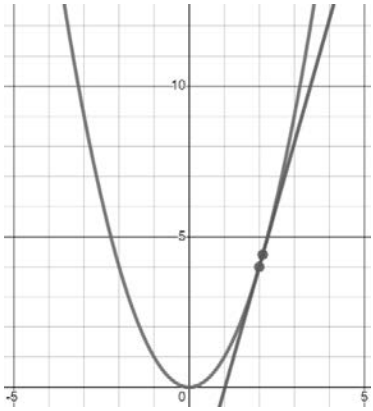
Zacznijmy od prostego przykładu. Przyjrzymy się funkcji $f(x) = x^2$ na rysunku 1.6. Jak „stroma" jest krzywa w punkcie $x = 2$?



Rysunek 1.6. Wyznaczanie stromizny w danym punkcie funkcji

Zauważ, że możemy mierzyć „stromiznę" przy dowolnym punkcie krzywej i zwizualizować ją jako linię styczną. Możesz myśleć o *stycznej* jako o linii prostej, która „ledwie dotyka" krzywej w danym punkcie. Określa ona również nachylenie krzywej w tym punkcie. Możesz z grubsza wyznaczyć styczną dla danej wartości x_1, tworząc linię przecinającą punkty $(x_1, f(x_1))$ oraz $(x_2, f(x_2))$ dla wartości x_2 *bardzo zbliżonej* do wartości x.

Weźmy $x_1 = 2$ oraz pobliską wartość $x_2 = 2{,}1$, które po podstawieniu do funkcji $f(x) = x^2$ dają $f(2) = 4$ oraz $f(2{,}1) = 4{,}41$, jak pokazano na rysunku 1.7. Wynikowa linia, która przechodzi przez te dwa punkty, ma nachylenie 4,1.



Rysunek 1.7. Przybliżony sposób obliczania nachylenia

Możesz szybko obliczyć nachylenie *m* krzywej przebiegającej między dwoma punktami za pomocą prostej formuły „różnica odległości w pionie dzielona przez różnicę odległości w poziomie”:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$m = \frac{4{,}41 - 4{,}0}{2{,}1 - 2{,}0}$$

$$m = 4{,}1$$

Gdybym jeszcze bardziej zmniejszył różnicę *x* między dwoma punktami, na przykład przyjmując wartości $x_1 = 2$ oraz $x_2 = 2{,}00001$, które dałyby w wyniku $f(2) = 4$ oraz $f(2{,}00001) = 4{,}00004$, bylibyśmy *bardzo* blisko rzeczywistego nachylenia równego 4. Zatem im mniejsza odległość między sąsiadującymi wartościami, tym bardziej zbliżamy się do wartości nachylenia w danym punkcie krzywej. Podobnie jak w przypadku wielu ważnych koncepcji w matematyce, znajdujemy coś znaczącego i zbliżamy się do nieskończenie dużych lub nieskończenie małych wartości.

Na listingu 1.17 pokazano kalkulator pochodnych napisany w Pythonie.

Listing 1.17. Kalkulator pochodnych napisany w Pythonie

```
def pochodna_x(f, x, rozmiar_kroku):
    m = (f(x + rozmiar_kroku) - f(x)) / ((x + rozmiar_kroku) - x)
    return m

def moja_funkcja(x):
    return x**2

nachylenie_w_punkcie_2 = pochodna_x(moja_funkcja, 2, .00001)
print(nachylenie_w_punkcie_2) # wypisuje 4.000010000000827
```

Dobra wiadomość jest taka, że istnieje bardziej elegancki sposób obliczania nachylenia w dowolnym punkcie funkcji. Używaliśmy już SymPy do tworzenia wykresów, a teraz pokażę Ci, jak wykonywać zadania takie jak obliczanie pochodnych z wykorzystaniem reprezentacji symbolicznej.

W pochodnej funkcji wykładniczej, takiej jak $f(x) = x^2$, wykładnik staje się mnożnikiem i zmniejsza się o 1, co daje nam pochodną $\frac{d}{dx}x^2 = 2x$. Zapis $\frac{d}{dx}$ wskazuje *pochodną względem x*, co oznacza, że interesuje nas, jak szybko zmienia się wartość funkcji w miarę zmian wartości *x*. Jeśli zatem chcemy znaleźć nachylenie dla wartości $x = 2$ i mamy funkcję pochodną, możemy po prostu podstawić odpowiednią wartość *x*, aby obliczyć nachylenie:

$$f(x) = x^2$$

$$\frac{d}{dx}f(x) = \frac{d}{dx}x^2 = 2x$$

$$\frac{d}{dx}f(2) = 2(2) = 4$$

Jeśli chcesz nauczyć się reguł, które pozwalają obliczać pochodne ręcznie, znajdziesz je w każdym podręczniku rachunku różniczkowego i całkowego. Istnieją jednak przydatne narzędzia pozwalające obliczać je w sposób symboliczny. Biblioteka SymPy jest bezpłatna, ma otwarty kod źródłowy i dobrze adaptuje się do składni Pythona. Na listingu 1.18 pokazano, jak obliczyć pochodną funkcji $f(x) = x^2$ w SymPy.

Listing 1.18. Wyznaczanie funkcji pochodnej w SymPy

```
from sympy import *
# Deklarujemy „x" w SymPy
x = symbols('x')
# Teraz używamy zwykłej składni Pythona, aby zadeklarować funkcję
f = x**2
# Obliczamy pochodną funkcji
dx_f = diff(f)
print(dx_f) # wypisuje 2*x
```

Zatem po zadeklarowaniu zmiennych za pomocą funkcji `symbols()` w SymPy możemy użyć zwykłej składni Pythona do zadeklarowania naszej funkcji. Następnie możemy użyć wywołania `diff()` do obliczenia funkcji pochodnej. Na listingu 1.19 zapisujemy wyznaczoną w ten sposób funkcję pochodną za pomocą zwykłego kodu Pythona i po prostu deklarujemy ją jako kolejną funkcję.

Listing 1.19. Kalkulator pochodnych w Pythonie

```
def f(x):
    return x**2

def dx_f(x):
    return 2*x

nachylenie_w_punkcie_2 = dx_f(2.0)
print(nachylenie_w_punkcie_2) # wypisuje 4.0
```

Gdybyś wolał nadal używać biblioteki SymPy, możesz wywołać funkcję `subs()`, aby zastąpić zmienną *x* wartością 2, jak pokazano na listingu 1.20.

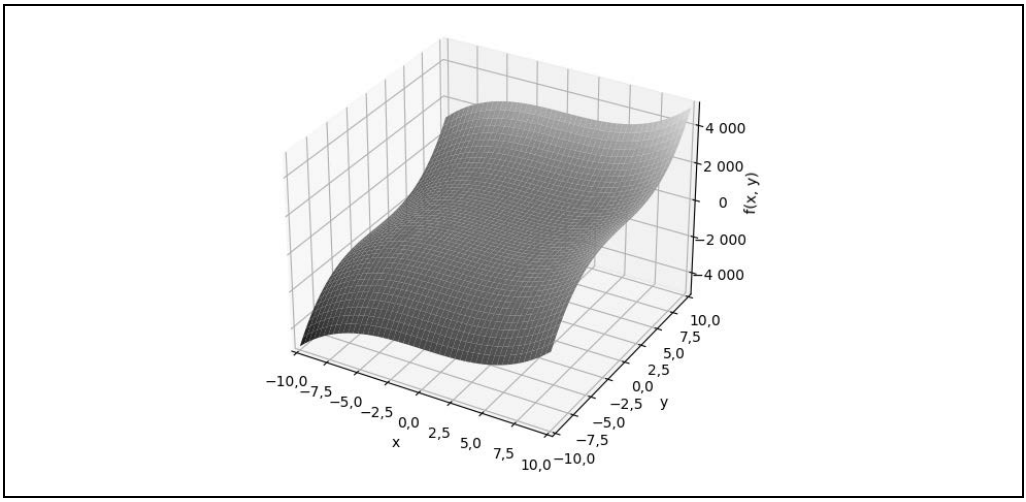
Listing 1.20. Używanie funkcji podstawiania w SymPy

```
# Oblicza nachylenie w punkcie x = 2
print(dx_f.subs(x,2)) # wypisuje 4
```

Pochodne cząstkowe

Inną koncepcją, którą napotkamy w tej książce, są *pochodne cząstkowe*. Będziemy używać ich w rozdziałach 5., 6. i 7. Są to pochodne funkcji mające wiele zmiennych wejściowych.

Pomyśl o tym w następujący sposób: zamiast nachylenia jednowymiarowej funkcji mamy nachylenia względem wielu zmiennych w różnych kierunkach. Dla pochodnej względem każdej zmiennej zakładamy, że pozostałe zmienne pozostają ustalone. Przyjrzyj się trójwymiarowemu wykresowi funkcji $f(x, y) = 2x^3 + 3y^3$ na rysunku 1.8, a zobaczysz, że mamy tu nachylenia w dwóch kierunkach dla dwóch zmiennych.



Rysunek 1.8. Tworzenie trójwymiarowego wykresu funkcji wykładniczej

Weźmy funkcję $f(x, y) = 2x^3 + 3y^3$. Zmienne *x* i *y* mają własne pochodne $\frac{d}{dx}$ i $\frac{d}{dy}$. Reprezentują one wartości nachylenia względem obu zmiennych na powierzchni wielowymiarowej. Kiedy mamy do czynienia z wieloma wymiarami, te „nachylenia" technicznie określamy mianem *gradientów*. Są to pochodne względem zmiennych *x* i *y*, a kod SymPy służący do ich obliczania pokazano poniżej.

$$f(x, y) = 2x^3 + 3y^3$$

$$\frac{d}{dx} 2x^3 + 3y^3 = 6x^2$$

$$\frac{d}{dy} 2x^3 + 3y^3 = 9y^2$$

Na listingu 1.21 i rysunku 1.8 pokazano, jak obliczyć pochodne cząstkowe odpowiednio względem x i y za pomocą SymPy.

Listing 1.21. Obliczanie pochodnych cząstkowych za pomocą SymPy

```
from sympy import *
from sympy.plotting import plot3d
# Deklarujemy x i y w SymPy
x,y = symbols('x y')
# Teraz używamy zwykłej składni Pythona do zadeklarowania funkcji
f = 2*x**3 + 3*y**3
# Obliczamy pochodne cząstkowe względem x i y
dx_f = diff(f, x)
dy_f = diff(f, y)
print(dx_f)     # wypisuje 6*x**2
print(dy_f)     # wypisuje 9*y**2
# Tworzymy wykres funkcji
plot3d(f)
```

Zatem dla wartości (x, y) równych (1, 2) nachylenie względem x wynosi $6(1) = 6$, a nachylenie względem y wynosi $9(2)^2 = 36$.

Używanie granic do obliczania pochodnych

Chciałbyś dowiedzieć się, jak wykorzystuje się granice podczas obliczania pochodnych? Jeśli czujesz, że dobrze rozumiesz to, czego nauczyliśmy się do tej pory, kontynuuj! Jeśli potrzebujesz trochę czasu na przetrawienie świeżo nabytej wiedzy, możesz wrócić do tej ramki nieco później.

SymPy pozwala na ciekawe eksploracje matematyczne. Weźmy naszą funkcję $f(x) = x^2$; przybliżyliśmy jej nachylenie w punkcie $x = 2$, rysując linię przechodzącą przez pobliski punkt $x = 2{,}0001$, a więc dodając przyrost 0,0001. A gdybyśmy użyli granicy, aby w nieskończoność zmniejszać ten przyrost s i zobaczyć, do jakiej wartości dąży nachylenie?

$$\lim_{s \to \infty} \frac{(x+s)^2 - x^2}{(x+s) - x}$$

W naszym przykładzie jesteśmy zainteresowani nachyleniem w punkcie $x = 2$, więc podstawmy odpowiednią wartość:

$$\lim_{s \to \infty} \frac{(2+s)^2 - 2^2}{(2+s) - 2} = 4$$

Zmniejszając w nieskończoność przyrost s w taki sposób, żeby zbliżał się do wartości 0, ale nigdy jej nie osiągał (pamiętaj, że sąsiedni punkt nie może dotknąć punktu $x = 2$, w przeciwnym razie nie mielibyśmy linii!), możemy użyć limitu, aby przekonać się, że nachylenie dąży do wartości 4, jak pokazano na listingu 1.22.

Listing 1.22. Używanie granic do obliczania nachylenia

```
from sympy import *

# „x" i wielkość przyrostu „s"
x, s = symbols('x s')

# deklarujemy funkcję
f = x**2

# nachylenie linii biegnącej między dwoma punktami oddalonymi o „s"
# używamy wzoru „różnica wysokości w pionie przez różnicę odległości w poziomie"
nachylenie_f = (f.subs(x, x + s) - f) / ((x+s) - x)

# podstawiamy 2 za x
nachylenie_2 = nachylenie_f.subs(x, 2)

# obliczamy nachylenie w punkcie x = 2
# w nieskończoność zmniejszamy przyrost _s_ tak, aby zbliżał się do 0
wynik = limit(nachylenie_2, s, 0)

print(wynik) # 4
```

A jeśli nie przypiszemy konkretnej wartości do zmiennej *x* i pozostawimy ją tak, jak jest? Co się stanie, jeśli będziemy w nieskończoność zmniejszać wartość przyrostu *s* i zbliżać go do zera? Spójrz na listing 1.23.

Listing 1.23. Używanie granic do obliczania pochodnej

```
from sympy import *

# „x" i wielkość przyrostu „s"
x, s = symbols('x s')

# deklarujemy funkcję
f = x**2

# nachylenie linii biegnącej między dwoma punktami oddalonymi o „s"
# używamy wzoru „różnica wysokości w pionie przez różnicę odległości w poziomie"
nachylenie_f = (f.subs(x, x + s) - f) / ((x+s) - x)

# obliczamy funkcję pochodną
# w nieskończoność zmniejszamy przyrost _s_ tak, aby zbliżał się do 0
wynik = limit(nachylenie_f, s, 0)
print(wynik) # 2x
```

Otrzymaliśmy naszą funkcję pochodną $2x$. Biblioteka SymPy jest na tyle „inteligentna", że nigdy nie pozwala, aby przyrost osiągnął wartość 0, a tylko w nieskończoność dążył do 0. W ten sposób możemy wyznaczyć pochodną $2x$ funkcji $f(x) = x^2$.

Reguła łańcuchowa

W rozdziale 7., kiedy zaczniemy budować sieć neuronową, będziemy potrzebować specjalnej sztuczki matematycznej nazywanej *regułą łańcuchową*. Łącząc warstwy sieci neuronowej, będziemy musieli rozdzielać pochodne z każdej warstwy. Na razie jednak nauczmy się reguły łańcuchowej na prostym przykładzie algebraicznym. Przypuśćmy, że masz dwie funkcje:

$$y = x^2 + 1$$
$$z = y^3 - 2$$

Zauważ, że funkcje te są powiązane, ponieważ *y* jest zmienną wyjściową pierwszej funkcji, ale wejściową drugiej. Oznacza to, że możemy podstawić pierwszą funkcję *y* do drugiej funkcji *z* w następujący sposób:

$$z = (x^2 + 1)^3 - 2$$

Jaka jest zatem pochodna *z* względem *x*? Mamy już podstawienie wyrażające *z* w kategoriach *x*. Policzmy pochodną za pomocą SymPy, jak pokazano na listingu 1.24.

Listing 1.24. Znajdowanie pochodnej z względem x

```
from sympy import *
z = (x**2 + 1)**3 - 2
dz_dx = diff(z, x)
print(dz_dx) # 6*x*(x**2 + 1)**2
```

Zatem pochodną funkcji *z* względem *x* jest $6x(x^2 + 1)^2$:

$$\frac{dz}{dx}((x^2 + 1)^3 - 2)$$
$$= 6x(x^2 + 1)^2$$

Zacznijmy jednak od początku i zróbmy to inaczej. Jeśli oddzielnie obliczymy pochodne funkcji *y* i *z* oraz pomnożymy je przez siebie, również otrzymamy pochodną funkcji *z* względem *x*! Spróbujmy:

$$\frac{dy}{dx}(\mathrm{x}^2 + 1) = 2x$$
$$\frac{dz}{dy}(\mathrm{y}^3 - 2) = 3y^2$$
$$\frac{dz}{dx} = (2\mathrm{x})(3y^2) = 6xy^2$$

OK, $6xy^2$ nie wygląda jak $6x(x^2 + 1)^2$, ale to tylko dlatego, że jeszcze nie podstawiliśmy funkcji *y*. Zróbmy to, aby cała pochodna $\frac{dz}{dx}$ była wyrażona w kategoriach *x* bez *y*.

$$\frac{dz}{dx} = 6xy^2 = 6x(\mathrm{x}^2 + 1)^2$$

Teraz widzimy, że otrzymaliśmy tę samą funkcję pochodną $6x(x^2 + 1)^2$!

Jest to *reguła łańcuchowa*, która mówi, że dla danej funkcji *y* (ze zmienną wejściową *x*) złożonej z inną funkcją *z* (ze zmienną wejściową *y*) możemy znaleźć pochodną *z* względem *x* poprzez pomnożenie dwóch odpowiednich pochodnych:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Na listingu 1.25 pokazano kod SymPy, który dokonuje tego porównania i pokazuje, że pochodna uzyskana poprzez zastosowanie reguły łańcuchowej jest równa pochodnej podstawionej funkcji.

Listing 1.25. Obliczanie pochodnej dz/dx przy użyciu i bez użycia reguły łańcuchowej daje ten sam wynik

```
from sympy import *
x, y = symbols('x y')
# pochodna pierwszej funkcji
# poprzedzamy y znakiem podkreślenia, aby zapobiec konfliktowi zmiennych
_y = x**2 + 1
dy_dx = diff(_y)
# pochodna drugiej funkcji
z = y**3 - 2
dz_dy = diff(z)
# obliczamy pochodną przy użyciu i bez użycia
# reguły łańcuchowej, podstawiając funkcję y
dz_dx_z_regula = (dy_dx * dz_dy).subs(y, _y)
dz_dx_bez_reguly = diff(z.subs(y, _y))
# dowodzimy reguły łańcuchowej, pokazując, że obie pochodne są równe
print(dz_dx_z_regula) # 6*x*(x**2 + 1)**2
print(dz_dx_bez_reguly) # 6*x*(x**2 + 1)**2
```

Reguła łańcuchowa jest kluczową częścią trenowania sieci neuronowej z odpowiednimi wagami i biasami. Zamiast rozplątywać pochodne poszczególnych węzłów w sposób przypominający zdejmowanie warstw cebuli, możemy pomnożyć pochodne w każdym węźle, co jest matematycznie znacznie prostsze.

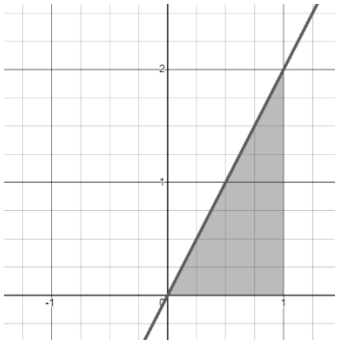
Całki

Odwrotnością pochodnej jest *całka*, która znajduje pole obszaru pod krzywą w danym zakresie. W rozdziałach 2. i 3. obliczymy pola pod rozkładami prawdopodobieństwa. Choć nie będziemy używać całek bezpośrednio, a zamiast tego posłużymy się skumulowanymi funkcjami gęstości, które są już scałkowane, warto wiedzieć, jak całki pozwalają obliczyć pole pod krzywą. W dodatku A znajdziesz przykłady używania tego podejścia na rozkładach prawdopodobieństwa.

Wykorzystam tu intuicyjne podejście do nauki całek, nazywane sumami Riemanna, które elastycznie adaptuje się do każdej funkcji ciągłej. Zauważmy najpierw, że znajdowanie pola obszaru w pewnym zakresie pod linią prostą jest łatwe. Przypuśćmy, że mam funkcję $f(x) = 2x$ i chcę znaleźć pole obszaru znajdującego się pod tą linią w zakresie od 0 do 1, zacieniowanego na rysunku 1.9.

Próbuję obliczyć pole obszaru ograniczonego linią oraz osią x, w zakresie wartości x od 0,0 do 1,0. Jeśli pamiętasz podstawowe wzory geometryczne, pole A trójkąta wynosi $A = \frac{1}{2}bh$, gdzie b jest długością podstawy, a h wysokością. Podstawiając liczby do wzoru, otrzymujemy pole obszaru równe 1,0, jak pokazano poniżej:

$$A = \frac{1}{2}bh$$

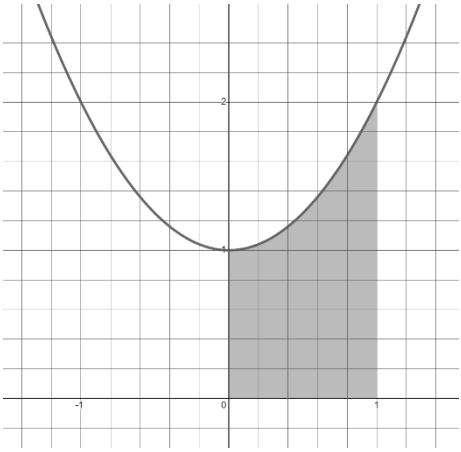


Rysunek 1.9. Obliczanie pola obszaru pod funkcją liniową

$$A = \frac{1}{2} \cdot 1 \cdot 2$$

$$A = 1$$

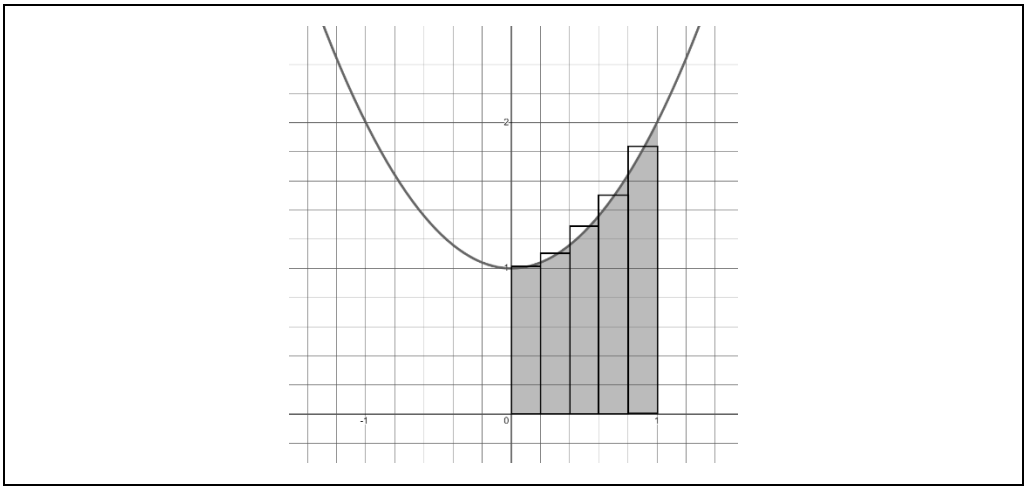
Nie było tak źle, prawda? Spójrzmy jednak na trudniejszy przypadek, funkcję $f(x) = x^2 + 1$. Jakie jest pole zacieniowanego obszaru między 0 a 1 na rysunku 1.10?



Rysunek 1.10. Obliczanie pola obszaru pod funkcją nieliniową nie jest takie proste

Ponownie jesteśmy zainteresowani polem obszaru pod krzywą i nad osią *x*, w zakresie *x* od 0 do 1. Krzywizna nie daje nam tu oczywistego geometrycznego wzoru na pole, ale możemy posłużyć się sprytną sztuczką.

Co by było, gdybyśmy umieścili pod krzywą pięć prostokątów o równych podstawach, jak pokazano na rysunku 1.11, które rozciągałyby się od osi *x* do miejsca, gdzie ich środkowy punkt stykałby się z krzywą?



Rysunek 1.11. Umieszczanie prostokątów pod krzywą w celu przybliżenia obszaru

Pole prostokąta wynosi A = długość · szerokość, więc możemy łatwo zsumować pola prostokątów. Czy dałoby to nam dobre przybliżenie pola obszaru pod krzywą? A gdybyśmy użyli 100 prostokątów? 1000? 100 000? W miarę zwiększania liczby prostokątów i zmniejszania ich szerokości, czy nie zbliżalibyśmy się coraz bardziej do pola obszaru pod krzywą? Owszem, i jest to kolejny przypadek, w którym zwiększamy/zmniejszamy coś do nieskończoności, aby zbliżyć się do rzeczywistej wartości.

Wypróbujmy to w Pythonie. Będziemy potrzebować funkcji przybliżającej całkę, którą nazwiemy `przybliz_calke()`. Argumenty `a` i `b` będą odpowiednio określać początek i koniec zakresu x. Argument `n` będzie określał liczbę upakowywanych prostokątów, a `f` będzie całkowaną funkcją. Na listingu 1.26 implementujemy tę funkcję, a następnie używamy jej do scałkowania funkcji $f(x) = x^2 + 1$ za pomocą pięciu prostokątów w zakresie od 0,0 do 1,0.

Listing 1.26. Przybliżone całkowanie w Pythonie

```
def przybliz_calke(a, b, n, f):
    delta_x = (b - a) / n
    suma = 0

    for i in range(1, n + 1):
        pkt_srodkowy = 0.5 * (2 * a + delta_x * (2 * i - 1))
        suma += f(pkt_srodkowy)

    return suma * delta_x
def moja_funkcja(x):
    return x**2 + 1

pole = przybliz_calke(a=0, b=1, n=5, f=moja_funkcja)
print(pole) # wypisuje 1.33
```

Otrzymaliśmy pole równe 1,33. Co się stanie, jeśli użyjemy 1000 prostokątów? Wypróbujmy to na listingu 1.27.

Listing 1.27. Kolejne przybliżone całkowanie w Pythonie

```
pole = przybliz_calke(a=0, b=1, n=1000, f=moja_funkcja)
print(pole) # wypisuje 1.333333250000001
```

OK, zyskujemy nieco precyzji i więcej miejsc dziesiętnych. Co się stanie, jeśli użyjemy miliona prostokątów, jak na listingu 1.28?

Listing 1.28. Jeszcze jedno przybliżone całkowanie w Pythonie

```
pole = przybliz_calke(a=0, b=1, n=1_000_000, f=moja_funkcja)
print(pole) # wypisuje 1.3333333333332733
```

OK, myślę, że odnotowujemy tu malejące zyski i zbliżamy się do wartości 1,333…, gdzie „3" powtarza się w nieskończoność. Gdyby była to liczba wymierna, prawdopodobnie byłyby to $^4/_3$ = 1,333… W miarę jak zwiększamy liczbę prostokątów, przybliżenie zbliża się do tej wartości na coraz mniejszych pozycjach dziesiętnych.

Teraz, kiedy wiemy mniej więcej, co próbujemy osiągnąć i w jaki sposób, wypróbujmy bardziej precyzyjne podejście z wykorzystaniem biblioteki SymPy, która — tak się składa — obsługuje liczby wymierne, jak pokazano na listingu 1.29.

Przykład 1.29. Całkowanie za pomocą SymPy

```
from sympy import *

# Deklarujemy „x" w SymPy
x = symbols('x')

# Teraz używamy zwykłej składni Pythona do zadeklarowania funkcji
f = x**2 + 1

# Obliczamy całkę funkcji względem x
# w obszarze między x = 0 a 1
pole = integrate(f, (x, 0, 1))

print(pole) # wypisuje 4/3
```

Znakomicie! A więc pole rzeczywiście wynosi $^4/_3$, co jest wartością, do której zbiegała się nasza poprzednia metoda. Niestety zwykły Python (i wiele innych języków programowania) obsługuje tylko liczby dziesiętne, ale systemy algebry komputerowej, takie jak SymPy, potrafią operować na dokładnych liczbach wymiernych. Będziemy używać całek do znajdowania pola obszarów pod krzywymi w rozdziałach 2. i 3., ale w nich całą pracę wykona za nas pakiet scikit-learn.

Obliczanie całek przy użyciu granic

Dla ciekawskich, oto jak można obliczać całki oznaczone z wykorzystaniem granic w SymPy. Jeśli czujesz się przytłoczony nadmiarem informacji, pomiń tę ramkę lub wróć do niej później. Jeśli jednak czujesz się komfortowo i chcesz dowiedzieć się, jak wyprowadza się całki przy użyciu granic, kontynuuj!

Główny pomysł bardzo przypomina to, co robiliśmy wcześniej: upakowujemy prostokąty pod krzywą i zmniejszamy je w nieskończoność, aż zbliżymy się do dokładnej wartości pola.

Oczywiście prostokąty nie mogą mieć szerokości 0…, muszą zbliżać się do zera, ale nigdy nie osiągnąć tej wartości. To kolejny przypadek wykorzystania granic.

W witrynie Khan Academy znajduje się doskonały artykuł (*https://oreil.ly/sBmCy*), który wyjaśnia, jak używać granic w połączeniu z sumami Riemanna, natomiast na listingu 1.30 pokazano, jak zrobić to w SymPy.

Listing 1.30. Obliczanie całek przy użyciu granic

```
from sympy import *

# Deklarujemy zmienne w SymPy
x, i, n = symbols('x i n')

# Deklarujemy funkcję i zakres
f = x**2 + 1
dolny, gorny = 0, 1

# Obliczamy szerokość i wysokość każdego prostokąta o indeksie „i”
delta_x = ((gorny - dolny) / n)
x_i = (dolny + delta_x * i)
fx_i = f.subs(x, x_i)

# Iterujemy po wszystkich „n” prostokątach i sumujemy ich pola
liczba_prostokatow = Sum(delta_x * fx_i, (i, 1, n)).doit()

# Obliczamy pole, zwiększamy liczbę
# prostokątów „n” do nieskończoności
pole = limit(liczba_prostokatow, n, oo)

print(pole) # wypisuje 4/3
```

Określamy długość podstawy każdego prostokąta `delta_x` oraz początek każdego prostokąta `x_i`, gdzie `i` jest indeksem prostokąta. `fx_i` to wysokość prostokąta o indeksie `i`. Deklarujemy liczbę prostokątów `n` i sumujemy ich pola `delta_x * fx_i`, ale nie mamy jeszcze wartości pola, ponieważ nie przypisaliśmy wartości zmiennej `n`. Zamiast tego zwiększamy `n` do nieskończoności, aby sprawdzić, do jakiej wartości będzie zbiegać się pole, i uzyskujemy $^4/_3$!

Podsumowanie

W tym rozdziale omówiliśmy kilka podstawowych działań, których będziemy używać dalej w tej książce. Od teorii liczb do logarytmów oraz rachunku różniczkowego i całkowego, przypomnieliśmy kilka ważnych koncepcji matematycznych mających zastosowanie w data science, uczeniu maszynowym i analityce. Może zastanawiasz się, dlaczego te koncepcje są użyteczne. Dowiesz się już niebawem!

Zanim przejdziemy do omówienia prawdopodobieństwa, poświęć chwilę na ponowne przejrzenie zaprezentowanego tu materiału, a następnie wykonaj poniższe ćwiczenia. Możesz wracać do niniejszego rozdziału podczas lektury książki i w razie potrzeby odświeżać wiedzę, kiedy zaczniesz stosować te idee matematyczne.

Ćwiczenia

1. Czy liczba 62,6738 jest wymierna, czy niewymierna? Dlaczego?
2. Oblicz wartość wyrażenia: $10^7 10^{-5}$
3. Oblicz wartość wyrażenia: $81^{\frac{1}{2}}$
4. Oblicz wartość wyrażenia: $25^{\frac{3}{2}}$
5. Zakładając, że nie są dokonywane żadne spłaty, ile warta byłaby po 3 latach pożyczka w wysokości 1000 złotych z 5-procentowymi odsetkami naliczanymi co miesiąc?
6. Zakładając, że nie są dokonywane żadne spłaty, ile warta byłaby po 3 latach pożyczka w wysokości 1000 złotych z 5-procentowymi odsetkami naliczanymi w sposób ciągły?
7. Jakie jest nachylenie prostej $f(x) = 3x^2 + 1$ w punkcie $x = 3$?
8. Jakie jest pole obszaru pod krzywą $f(x) = 3x^2 + 1$ w zakresie x od 0 do 2?

Odpowiedzi znajdują się w dodatku B.

Skorowidz

A

algebra liniowa, 101
analizowanie małych prób, 97

B

biasy, 211
biblioteka
  NumPy, 126
  Pandas, 152
  PyTorch, 220
  scikit-learn, 153
  statsmodel, 153
  SymPy, 17
  TensorFlow, 220
big data, 98
błąd
  autoselekcji, 66
  konfirmacji, 66
  przeżywalności, 67
  standardowy estymacji, 158
  teksańskiego snajpera, 98

C

całka, 37
CDF, cumulative density function, 79
centralne twierdzenie graniczne, 84
czułość, 195

D

dane, 62
  testowe, 162, 191
  treningowe, 162, 191
data science, 227
  biblioteki w Javie, 236
  dyscypliny, 228
  narzędzia, 228
  praca marzeń, 249
  rynek pracy, 242
deklarowanie trójwymiarowego wektora, 104
dodawanie
  logarytmiczne, 177
  wektorów, 105
dominanta, 71
dopasowywanie
  krzywej logistycznej, 171, 174
    estymacja największej wiarygodności, 176
    metoda gradientu prostego, 178
    używanie SciPy, 174
  linii regresji, 133, 138
    metoda gradientu prostego, 142, 144
    metoda stochastycznego gradientu prostego, 149
    odwracanie macierzy, 140
    rozkład macierzy, 141
    równanie w formie zamkniętej, 139
dystrybuanta, 79
  odwrotna, 81

E

eksploracja danych, 63
elementy odstające, 70
estymacja, 158
  największej wiarygodności, MLE, 176

F

funkcja, 15
  array(), 102, 114
  binom.pmf(), 53
  corr(), 152
  diff(), 32, 216
  doit(), 20
  dot(), 118
  evalf(), 29
  exp(), 26, 177
  fit(), 135
  flatten(), 175
  ln(), 28
  log(), 23, 177
  matmul(), 117, 118
  norm.ppf(), 82
  predict(), 175
  predict_prob(), 175
  range(), 19
  subs(), 20, 33
  symbols(), 32
funkcje, 15
  aktywacji, 204, 207, 210
    logistyczne, 170, 172, 182, 206
    przeciekające ReLU, 207
    ReLU, 205, 207
    softmax, 207
    tangens hiperboliczny, 207
  ciągłe, 16
  gęstości prawdopodobieństwa, PDF, 78
  krzywoliniowe, 17
  kumulacyjne gęstości, CDF, 79
  kwadratowe, 18
  kwantylowe, 81
  liniowe, 17, 207
  propagacji w przód, 210
  straty, 147
  wykres, 16, 18, 33, 205

G

generowanie liczb losowych, 82
gradient, 33
  prosty, 142
  stochastyczny, 217
granica, 28

H

hipoteza
  alternatywna, 90
  zerowa, 90, 154
histogram, 76

I

implementacja sieci neuronowej, 217
inwersja, 113, 120
  liniowa, 140
istotność statystyczna, 89, 153
  testowanie, 156

J

Java
  biblioteki data science, 236
język
  Go, 234
  SQL, 231
Jupyter, 235

K

klasyfikacja, 169, 181, 195
kolejność działań, 13
korelacja, 151
  liniowa, 133
  Pearsona, 151
kreślenie
  funkcji logistycznej, 173
  funkcji straty, 147
krytyczna wartość z, 87
krzywa
  CDF, 79
  PDF, 79
  ROC, 196
kumulacyjna funkcja gęstości, CDF, 79
kwadraty reszt, 137
kwantyl, 70

L

liczba Eulera, e, 24
liczby
  całkowite, 12
  całkowite nieujemne, 12
  naturalne, 12

niewymierne, 12
rzeczywiste, 13
wymierne, 12
zespolone i urojone, 13
linia
logarytmu szansy, 184
regresji
dopasowywanie, 135
LOC, level of confidence, 87
logarytm, 23
naturalny, 28
szansy, 182
wiarygodności dopasowania, 186, 187
wiarygodności, 187
logarytmiczne dodawanie, 177
logistyczna funkcja aktywacji, 170, 172, 182, 206
logit, 182, 183
losowość, 150

Ł

łączenie
dwóch przekształceń, 117
wektorów, 105

M

macierz
błędów, 192, 195
diagonalna, 123
jednostkowa, 122
korelacji, 157
kwadratowa, 122
odwrotna, 122, 124, 140
rzadka, 123
trójkątna, 123
wag, 209
macierze
mnożenie, 117
mnożenie przez wektor, 113
rozkład, 141
transponowane, 114
wartości własne, 127
wektor własny, 127
margines błędu, 88
mediana, 69
metoda gradientu
prostego, 142, 178
stochastycznego, 149
w SymPy, 145
znajdowanie minimum paraboli, 143
stochastycznego, 217
MLE, maximum likelihood estimation, 176
mnożenie
macierzy, 117
macierzy przez wektor, 113
moda, 71

N

nadmierne dopasowanie, 148
naiwny klasyfikator Bayesa, 51
niedomiar zmiennoprzecinkowy, 177
nierównowaga klas, 197
NoSQL, 232
notatniki Jupytera, 235
NumPy, 126
macierze wag, 209
regresja liniowa, 150
rozwiązywanie układu równań, 126
sieć jednokierunkowa, 208
wektory biasów, 209

O

obciążenie, 65, 134
obliczanie
błędu standardowego estymacji, 159
całek, 40
dokładności, 212
dominanty, 71
krytycznej wartości z, 88
logarytmu wiarygodności dopasowania, 186
mediany, 70
odchylenia standardowego, 73, 74
pochodnej w punkcie, 30
pochodnych, 34–37, 213–216
pola obszaru pod funkcją, 38
poziomu ufności, 89
prawdopodobieństwa, 58, 91
prawdopodobieństwa łącznego, 176
przedziału przewidywania, 161
reszt, 136
sumy kwadratów, 137
średniej, 68
średniej ważonej, 69
wariancji, 72
wartości krytycznej, 155

obliczanie
  wartości p, 190
  współczynnika korelacji, 152, 153
obrót, 113, 115
odchylenie standardowe, 73
  w populacji, 71
  w próbie, 74
odsetki
  „ciągłe”, 27
  składane, 25
operator @, 117, 118

P

Pandas, 135, 232
  sprawdzanie współczynnika korelacji, 152
  tworzenie macierzy korelacji, 157
PDF, probability density function, 78
perceptron wielowarstwowy, 220
p-hacking, 96
pochodna, 30
  cząstkowa, 33, 213
  względem x, 32
podstawa, 20
pole pod krzywą, 196
populacja, 65
  odchylenie standardowe, 71
  wariancja, 71
potęgowanie, 20
powłoka liniowa, 109
poziom ufności, LOC, 87
PPF, 81
prawdopodobieństwo, 43, 45
  alternatywne, 47, 51
  łączne, 46, 51, 177
  warunkowe, 48, 49
precyzja, 195
prognoza, 133, 175
programowanie, 233
propagacja
  w przód, 208
  wsteczna, 213, 220
próba, 65
  odchylenie standardowe, 74
  wariancja, 74
przedział
  przewidywania, 159
  ufności, 87, 160
przekształcenia liniowe, 110, 112, 115, 119
  rodzaje ruchu, 112

R

regresja
  dopasowywanie krzywej logistycznej, 171, 174
  funkcja logistyczna, 172
  grzbietowa, 149
  lasso, 149
  liniowa, 131, *Patrz także* dopasowywanie linii regresji
    przy użyciu SciPy, 134
    przy użyciu SymPy, 146
  logistyczna, 169
    logarytm szansy, 182
    z wieloma zmiennymi, 179
    ze sprawdzaniem krzyżowym, 192
  obliczanie wartości p, 189, 190
  podział danych, 162
  prognozowanie, 133
  przedziały przewidywania, 159
  reszty, 136
  sprawdzanie z podziałem losowym, 166
  wielokrotna, 167
  współczynnik r-kwadrat, 164, 185, 189
  znajdowanie najlepiej dopasowanej linii, 138
reguła
  iloczynowa, 20
  łańcuchowa, 35
  sumy, 48
reszty, 136
RGB, 202
r-kwadrat, 164, 185, 189
ROC, receiver operating characteristic, 196
rozkład
  macierzy, 127, 141
  normalny, 75
    standardowy, 82
    własności, 78
  prawdopodobieństwa
    beta, 54
    dwumianowy, 52
  Studenta, t, 97, 155
  wzdłuż linii, 160
  χ2, 190
rozrzut danych, 72
rozwiązywanie układu równań, 126

równanie
  w formie zamkniętej, 139
  wartości własnej, 128
równoważenie klas, 198
różniczkowanie automatyczne, 217

S

scikit-learn, 153, 162
  opcja stratify, 198
  sieć neuronowa, 220
SciPy
  dopasowanie linii regresji, 135
  macierz błędów, 194
  regresja liniowa, 134
  regresja logistyczna, 174
sieć neuronowa, 200
  działanie, 201
  funkcje aktywacji, 204, 210
  jednokierunkowa, 208
  metoda gradientu stochastycznego, 217
  ograniczenia, 221
  propagacja w przód, 208, 210
  propagacja wsteczna, 213
  tworzenie, 201
  używanie scikit-learn, 220
  wizualizacja, 210
  ze stochastycznym zejściem po gradiencie, 217
sigma, Σ, 19
skalar, 106
skalowanie, 113, 119
  wektorów, 106
sprawdzanie krzyżowe, 165, 191
  z podziałem losowym, 166
  z wyłączaniem jednego elementu, 166
SQL, Structured Query Language, 231
standardowy rozkład normalny, 82
standaryzacja Z, 82
statystyka, 45, 62
  opisowa, 64, 68
stopnie swobody, 98, 155, 190
suma
  kwadratów, 137, 138
  kwadratów błędów, 158
sumowanie, 19
SymPy, 17, 29
  całkowanie, 40
  kreślenie funkcji, 17
  logistyczna funkcja aktywacji, 206
  metoda gradientu prostego, 145
  regresja liniowa, 146
  rozwiązywanie układu równań, 126
  sumowanie, 20
  upraszczanie wyrażeń, 22
  wyznaczanie funkcji pochodnej, 32
szansa, 183
sztuczna inteligencja, 223

Ś

ścinanie, 113, 115, 120
średnia
  populacji, 68
  próby, 68
  ważona, 69

T

tempo nauki, 142
teoria liczb, 12
test
  dwustronny, 93
  jednostronny, 91
testowanie
  hipotez, 90
  istotności, 156
transpozycja macierzy, 114
twierdzenie Bayesa, 49, 195
typy obciążenia danych, 66

U

uczenie
  głębokie, 200
  maszynowe, 68, 132
  ograniczenia, 221
  trening, 139
ufność, 160
układ
  równań, 124
  współrzędnych, 16
ukryte IT, 248

W

wagi, 211
wariancja, 72, 148
  w populacji, 71
  w próbie, 74
wartości p, 89, 189
wektory, 101
  bazowe, 110, 116
  biasów, 209
  dodawanie, 105
  inwersja, 113
  liniowo niezależne, 109
  liniowo zależne, 109
  łączenie, 105
  obrót, 113
  skalowanie, 106, 113
  ścinanie, 113
  trójwymiarowe, 104
  własne macierzy, 128
  zmienianie kierunku, 108
wiarygodność, 44
wielokrotna regresja liniowa, 167
wizualizacja danych, 236
własności rozkładu normalnego, 78
właściwości potęg i logarytmów, 24
wnioskowanie statystyczne, 64, 84
współczynnik
  determinacji, 157
  korelacji, 151
  r-kwadrat, 164, 185, 189
  zmienności, 84
wyjście aktywowane, 206
wykładnik, 20
wykres
  funkcji
    kwadratowej, 18
    liniowej, 16
    ReLU, 205
    wykładniczej, 33
  regresji logistycznej, 175
  trójwymiarowy funkcji, 18
wyniki Z, 82
wyznacznik, 119

Z

zbiór
  bimodalny, 71
  danych MNIST, 208, 221
zdarzenia
  niewykluczające się, 47
  wzajemnie się wykluczające, 47
zmienna wyjściowa, 15
zmienne, 14
  wejściowe, 15

Zrozum matematykę i efektywnie używaj danych!

Rosnąca dostępność danych powoduje, że data science i uczenie maszynowe znajdują bardzo szerokie zastosowanie. Równocześnie wiele osób pomija analizy matematyczne przed rozpoczęciem przetwarzania danych. A to wiąże się z ryzykiem popełnienia istotnych błędów już na etapie projektowania danego systemu. Dopiero dogłębne zrozumienie niektórych koncepcji matematycznych i umiejętność ich praktycznego zastosowania sprawia, że kandydat na analityka danych ma szansę osiągnąć poziom profesjonalisty.

To książka przeznaczona dla osób, które chcą dobrze zrozumieć matematyczne podstawy nauki o danych i nauczyć się stosowania niektórych koncepcji w praktyce. Wyjaśniono tu takie zagadnienia jak rachunek różniczkowy i całkowy, rachunek prawdopodobieństwa, algebra liniowa i statystyka, pokazano także, w jaki sposób posługiwać się nimi w regresji liniowej, regresji logistycznej i w tworzeniu sieci neuronowych. Poszczególne tematy zostały omówione zrozumiale, przystępnie, bez naukowego żargonu, za to z licznymi praktycznymi przykładami, co dodatkowo ułatwia przyswojenie koncepcji i prawideł matematyki. Opanowanie zawartej tu wiedzy pozwala uniknąć wielu kosztownych błędów projektowych i trafniej wybierać optymalne rozwiązania!

Dzięki książce nauczysz się:

- używać kodu Pythona i jego bibliotek do eksplorowania koncepcji matematycznych
- posługiwać się regresją liniową i regresją logistyczną
- opisywać dane metodami statystycznymi i testować hipotezy
- manipulować wektorami i macierzami
- łączyć wiedzę matematyczną z użyciem modeli regresji
- unikać typowych błędów w stosowaniu matematyki w data science

Thomas Nield jest instruktorem w O'Reilly Media i wykładowcą na Uniwersytecie Południowej Kalifornii. Prowadzi zajęcia z analizy danych, uczenia maszynowego, optymalizacji matematycznej i praktycznego zastosowania sztucznej inteligencji. Autor kilku cenionych książek. Założyciel firmy Yawman Flight, która opracowuje uniwersalne systemy sterowania ręcznego do symulatorów lotu i bezzałogowych statków powietrznych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-013-0
9 788383 220130
Cena: 69,00 zł