MACHINE LEARNING I NATURAL LANGUAGE PROCESSING W PROGRAMOWANIU

Piotr Wróblewski

Podręcznik z ćwiczeniami w Pythonie

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

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

ISBN: 978-83-289-1580-0

Printed in Poland.

Spis treści

ROZDZIAŁ 6.

Szybka analiza danych przy użyciu Matplotlib

Powtórka z Matplotlib • Rozkład normalny • Popularne miary statystyczne • Interakcja NumPy i Pandas z Matplotlib • Narysować oznacza zrozumieć!

W wielu zadaniach programistycznych wizualizacja danych stanowi uwiecznienie żmudnej pracy wykonywanej przez program, w nauczaniu maszynowym jest często wymagana jeszcze przed zbudowaniem modeli ML. Dlatego każdy, kto poważnie podchodzi do nauki ML, prędzej czy później będzie potrzebował wykresów graficznych, które pomogą lepiej zrozumieć dane, wykryć anomalie, wstępnie zidentyfikować pewne korelacje itp.

W celu analizy informacji, jakie skrywają w sobie „surowe" serie danych, można podejść do tematu „matematycznie", tj. wykorzystać biblioteki, które nam to ułatwią, pomagając w oczyszczaniu danych i umożliwiając wszechstronne analizy statystyczne. Bezcenne są tu mechanizmy oferowane przez NumPy i Pandas (zob. rozdział 4.). Czasem to nie wystarcza i może nam coś ważnego umknąć — i tu właśnie mogą pomóc typowe wizualizacje stosowane w analizie danych: wykresy liniowe, punktowe, słupkowe, histogramy i wiele innych. Nawet takie proste realizacje często pozwalają wykryć zaburzenia, anomalie, nieoczywiste korelacje — czasem na pierwszy rzut oka.

O tym właśnie opowiem w bieżącym rozdziale i przekonasz się, że nie są to trudne zagadnienia. Do budowy wykresów użyjemy biblioteki Matplotlib, co wymaga jej zaimportowania do środowiska Pythona (`pip3 install matplotlib`, w Windowsie użyj polecenia `pip`) lub dodania do projektu PyCharma. Oprócz Matplotlib analitycy danych często używają biblioteki seaborn (*https://seaborn.pydata.org*), ale jej przestudiowanie pozostawiam już Tobie — w książce użyję jej kilka razy i zalecam wykonanie instalacji (`pip3 install seaborn`, w systemie Windows `pip`).

Matplotlib oferuje bogaty zestaw klas pozwalających na tworzenie wizualizacji w Pythonie, i to nie tylko statycznych, lecz także dynamicznych, a nawet interaktywnych, jeśli połączymy ją z narzędziami interaktywnymi, takimi jak IPython Widgets. Umożliwi to tworzenie interaktywnych wykresów, które użytkownik może modyfikować, np. dokładając klasyczne formy interakcji, takie jak suwaki, pola tekstowe czy przyciski, aby zmieniać parametry wykresów w czasie rzeczywistym, podobnie jak np. w PowerBI.

Podczas instalacji Matplotlib wgrywane są dodatkowe moduły, których ona wymaga (*pyparsing*, *numpy*, *kiwisolver*, *fonttools*, *cycler*, *contourpy*). NumPy już znasz z poprzednich rozdziałów, jest to specjalistyczna biblioteka ułatwiająca obsługę dużych, wielowymiarowych tablic i macierzy, które mogą stanowić źródła danych dla wykresów tworzonych z użyciem Matplotlib. Autorami biblioteki są John D. Hunter i Michael Droettboom. Gdy już zapoznasz się z omówieniem zawartym w tym rozdziale, koniecznie odwiedź stronę zawierającą źródłowe informacje referencyjne opisujące wszystkie metody zawarte w tej bibliotece: *https://matplotlib.org*.

Jeśli używasz systemu Windows i podczas testowania przykładowych programów uruchomienie nie powiedzie się, np. otrzymasz komunikat podobny do `from matplotlib._path import ImportError: DLL load failed while importing _path: Nie można odnaleźć określonego modułu`, to zainstaluj dodatek o nazwie Microsoft Visual C++ Redistributable. Zawiera on biblioteki, które są wymagane do uruchomienia aplikacji w języku C++ skompilowanych przy użyciu programu Visual Studio 2015. Dodatek jest dostępny pod adresem: *https://www.microsoft.com/pl-PL/download/details.aspx?id=48145*. W czasie pisania tej książki dla systemu Windows 11 Pro nie zaobserwowałem już tego błędu, ale podaję te wskazówki na wszelki wypadek.

Wykresy punktowe i liniowe

Nawet bez wczytywania się w obszerną dokumentację[19] można szybko i bez wysiłku utworzyć prosty program rysujący wykresy. Załóżmy, że interesuje nas stworzenie wykresu funkcji, dla której znamy dziewięć wartości (pary *x*, *y*), które w kodzie Pythona są umieszczone w listach o nazwach `osX` i `osY`. Taki zestaw można zwizualizować, pokazując wykres punktowy (inna nazwa to wykres rozrzutu, ang. *scatter plot*), lub też jako wykres liniowy, uznając, że punkty danych są częścią wykresu liniowego.

Skrypt, który tworzy taki wykres, jest wyjątkowo zwięzły (*matplot0.py*):

```
from matplotlib import pyplot as plt
osX =[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
osY =[5, 5.6,  8,  9, 11, 20, 14, 12, 10, 9.5]
plt.plot(osX, osY)   # Wykres liniowy
# Dodanie pogrubionych kropek w punktach danych
plt.scatter(osX, osY, color='grey', s=50)   # Wykres punktowy, s — rozmiar kropki
plt.savefig('wykresik.pdf')
plt.show()
```

Mimo że skrypt zajmuje raptem kilka linijek, wykonuje sporo pracy:

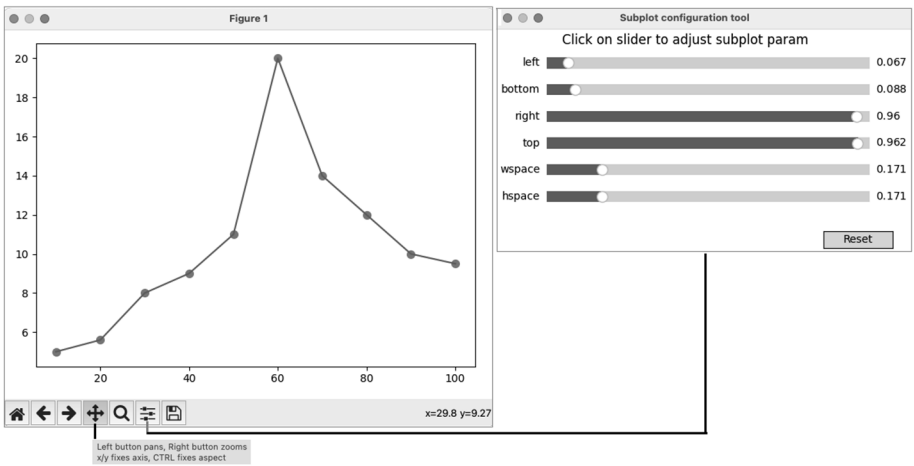
- Wyświetla miniprogram graficzny zawierający wykres i kontrolki pozwalające nim operować (skalowanie, nawigacja w zakresie osi *X*, *Y*).

[19] Określenie „obszerną” to eufemizm: podręcznik użytkownika zajmuje 4917 stron w PDF-ie dla wersji 3.8.3 i ciągle rośnie z kolejnymi aktualizacjami.

- W ramach prezentu pożegnalnego na dysku w katalogu roboczym zostanie zapisany plik o nazwie *wykresik.pdf*, będący kopią wykresu w jego układzie pierwotnym.

Dane liczbowe na osi *X* celowo dobrałem tak, aby były posortowane i w miarę w równych odstępach, by uzyskać ładny wykres linii ciągłej (funkcja `plot()`), gdyż linia jest rysowana punkt po punkcie i chciałem uniknąć efektu cofania kreski. Na linię są nałożone też kropki punktów danych (funkcja `scatter()`).

Rysunek 6.1 pokazuje okienko, które zobaczysz na ekranie po uruchomieniu skryptu. Jest to wersja macOS-a, natomiast dla Windowsa lub w Linuksie styl okna będzie nieco inny.



RYSUNEK 6.1. Wykres wygenerowany przez moduł pyplot (Matplotlib)

Okno oferuje kilka przycisków nawigacyjnych:

- — powrót do układu pierwotnego (reset);
- strzałki i — powrót do poprzedniego / przywrócenie cofniętego widoku;
- — skalowanie lub przesunięcie z użyciem myszy (np. lewy klawisz przesuwa wykres, prawy skaluje);
- — zaznacz obszar z przyciśniętym lewym klawiszem myszy i puść, aby wejść w tryb zoom obszaru;
- — marginesy i pozycja wykresu względem okna (zob. osobne okno *Subplot configuration tool*);
- — zapis grafiki na dysk w formacie PNG.

Wykonaj kilka ćwiczeń, gdyż przeznaczenie niektórych kontrolek może nie być zbyt intuicyjne. Przycisk resetu (symbol domku) pozwala wrócić do pierwotnego układu wykresu przed ewentualnymi zmianami skali lub przesunięciami, które użytkownik wykonał za pomocą myszy po wciśnięciu przycisku pozwalającego na skalowanie wykresu (zoom) lub zmianę widocznego obszaru w ramach osi *X*, *Y*.

Wspierane funkcje skalowania lub przesunięć:

- Możesz chwytać wykres myszą i przesuwać w lewo i w prawo, np. aby przejść poniżej osi *X*, w stronę ujemnych wartości *Y*.
- Wciśnij klawisz *Ctrl* i przytrzymując wciśnięty *prawy klawisz* myszy, przesuwaj ją w górę (powiększanie) lub w dół (pomniejszanie).
- Wciśnij klawisz *Ctrl* i przytrzymując wciśnięty *lewy przycisk* myszy, przesuwaj ją w górę i w dół (przesuwanie się na osi *X*) lub w lewo i prawo (przesuwanie się na osi *Y*).

Ponowne wciśnięcie przycisku pozwala powrócić do klasycznego ekranu i zachowanego stanu wykresu.

Modyfikacje wyglądu wykresu

Prosty wykres z domyślnymi ustawieniami rzadko spełnia nasze oczekiwania i może okazać się nieczytelny. Modyfikacje wyglądu wykresu wprowadzamy jako dodatkowe parametry metody rysującej wykres, którym u nas jest linia oferowana przez funkcję `plot()`, wzbogacona kropkami punktów danych. Dla takiego prostego wykresu dwuwymiarowego można łatwo zmodyfikować kolor, grubość oraz styl linii, można też zmienić sposób wyświetlania punktów danych.

W tym celu funkcję `plot()` wzbogacamy o parametry:

- `marker` — styl punktu danych;
- `ms` — rozmiar markera (ang. *marker size*);
- `linestyle` — styl linii;
- `color` — kolor;
- `linewidth` — grubość linii.

Tabela 6.1 zawiera *listę kolorów* dostępnych w ramach podstawowej palety RGB. Oprócz takich kodów można podawać też kolory, używając notacji kodowej RGB w konwencji `'#rrggbb'` (np. `'#67e210'` pozwoli uzyskać piękny, wiosenny odcień zieleni).

TABELA 6.1. Kody kolorów palety podstawowej rozróżniane przez funkcję wykreślającą plot

CECHA	KOLOR	CECHA	KOLOR
`'m'`	Purpurowy (ang. *purple*), magenta	`'r'`	Czerwony (ang. *red*)
`'y'`	Żółty (ang. *yellow*)	`'g'`	Zielony (ang. *green*)
`'k'`	Czarny (ang. *black*)	`'b'`	Niebieski (ang. *blue*)
`'w'`	Biały (ang. *white*)	`'c'`	Niebieskozielony (ang. *cyan*)

Tabela 6.2 zawiera listę popularnych kodów pozwalających zmodyfikować *styl kreskowania*, co jest bardzo przydatne podczas rysowania wykresów wielokrotnych i ułatwia rozróżnianie, która linia odpowiada określonej serii danych.

TABELA 6.2. Typy kreskowania w Matplotlib

KOD	STYL
`'-'`	____ (linia ciągła)
`':'`	..... (linia kropkowana)
`'--'`	– – – (linia kreskowana)
`'-.'`	– . – . (linia kropkowano-kreskowana)

Tabela 6.3 zawiera kilka wybranych kodów, których można użyć do oznaczenia punktu danych tworzącego wykres.

TABELA 6.3. Wybrane style markerów

KOD	STYL	KOD	STYL
`'o'`	Kółko	`'P'`	Duży pogrubiony plus
`'s'`	Kwadracik	`'s'`	Gwiazdka
`'*'`	Gwiazdka	`'D'`	Diament równomierny
`'x'`	Symbol X	`'1'`	Drzewko
`'X'`	Symbol X wypełniony	`'p'`	Pięciobok
`'+'`	Plus	`'H'`	Sześciobok
`'v'`	Trójkącik w dół	`'^'`	Trójkącik w górę
`'<'`	Trójkącik w lewo	`'>'`	Trójkącik w prawo

Popatrz na przykładowe zastosowanie wybranej kombinacji tych parametrów. Użyjemy tu nieco zmodyfikowanego zbioru danych, który nie jest posortowany „parami". Dlatego w celu uzyskania ładnego wykresu, narysowanego jednym pociągnięciem kreski „od lewej do prawej", przerzucimy się na tablice NumPy, które posortujemy przed użyciem (*matplot1.py*):

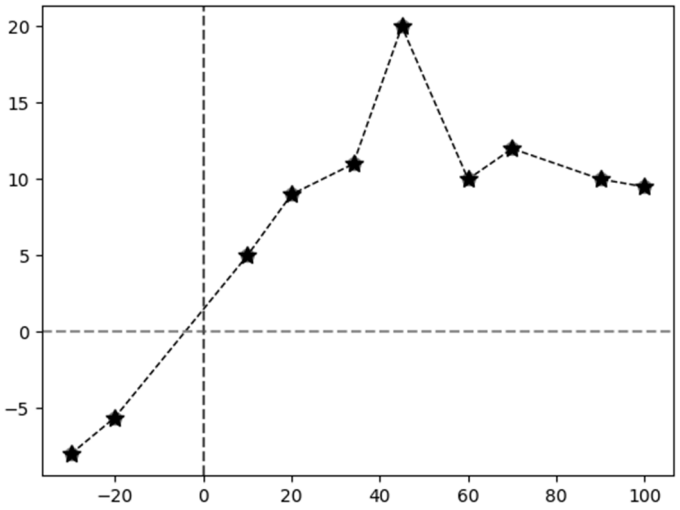
```
import numpy as np
from matplotlib import pyplot as plt
# Tablice par wartości do wykresu, nieposortowane
osX = np.array([10, -20,  -30, 20, 34, 45, 60, 70, 90, 100])
osY = np.array([5,  -5.6,  -8,  9, 11, 20, 10, 12, 10, 9.5])
posortowane_indeksy = np.argsort(osX) # Sortowanie indeksów
# Sortowanie osX i osY za pomocą posortowanych indeksów:
x_sort = osX[posortowane_indeksy]
y_sort = osY[posortowane_indeksy]
plt.scatter(x_sort, y_sort, color='grey', s=50)   # s — rozmiar kropki
plt.plot(x_sort,  y_sort,
         marker='*',          # Styl markera (tu: symbol gwiazdki)
         linestyle='--',      # Styl linii (tu: kreskowana)
         color='k',           # Kod koloru (tu: czarny)
         ms=10,               # Rozmiar markera
         linewidth = '1')     # Grubość linii
```

```
plt.axhline(0, color='c', linestyle='--') # Dodatkowa linia pozioma dla Y = 0
plt.axvline(0, color='g', linestyle='--') # Dodatkowa linia pionowa dla X = 0
plt.show()
```

Wykres dodatkowo wzbogaciłem liniami: poziomą (dla Y = 0) i pionową (dla X = 0). Taki zabieg ułatwi odczyt wartości.

Rysunek 6.2 pokazuje wynik uruchomienia programu.

RYSUNEK 6.2. Style linii i markery w akcji



Wykresy wielokrotne

Wiele wykresów komponuje się w celu porównania kilku serii danych, co oczywiście jest wspierane przez Matplotlib. Możliwość dołożenia nowego wykresu do poprzedniego polega na *kolejnym wywołaniu funkcji kreślącej* `plot()`. Warto w tym momencie dodać kilka ozdobników, których użycie pokażę na prostym przykładzie:

- własny tytuł wykresu,
- etykiety osi,
- legenda,
- znormalizowanie wartości na osi *X* (zamiast wypisywać wszystkie liczby, utworzymy pięć etykiet w zakresie od wartości *min* do *max*).

A zatem do dzieła (*matplot2.py*)!

```
from matplotlib import pyplot as plt
osX=[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
osY =[5, 5.6, 8, 9,  11, 20, 14, 12, 10, 9.5]
osY2=[2, 3,   6, 9,  12, 13, 16, 14, 12, 12 ]
seria1,=plt.plot(osX, osY,  marker='*', linestyle='--', color='k',
                 ms=10, linewidth = '1', label='Seria 1.') # Seria 1.
seria2,=plt.plot(osX, osY2, marker='o', linestyle='-',  color='k',
                 ms=10, linewidth = '1', label='Seria 2.') # Seria 2.
```

```
plt.legend(handles=[seria1, seria2])
plt.title("Pomiary napięcia")
plt.xlabel("[t] - czas")
plt.ylabel("[V] - napięcie")
# Znormalizowana lista z pięcioma wartościami od min(osX) do max(osX)
lista_do_etykiet_X = np.linspace(min(osX), max(osX), 5) # (*)
# Zaokrąglanie do liczb całkowitych:
lista_do_etykiet_X = np.around(lista_do_etykiet_X).astype(int)
plt.xticks(lista_do_etykiet_X) # Normalizacja wartości na osi X
plt.show()
```

Dość specyficzna składnia użyta powyżej (przecinek po nazwie zmiennej `seria1`) wynika z konstrukcji klasy `pyplot` w Matplotlib i wymagań wobec określonych typów danych używanych do przekazywania wartości koniecznych do zbudowania tekstu legendy. Chcemy bowiem zwrócić typ `tuple` zamiast pojedynczej wartości. Ponieważ `plt.plot` zwraca pojedynczą zmienną dla linii wykresu, to dodając przecinek po nazwie zmiennej, uzyskamy tuplę z jednym elementem, co jest konwencją często stosowaną w Pythonie. W naszym kodzie bowiem `plt.legend()` oczekuje argumentów w formie tupli.

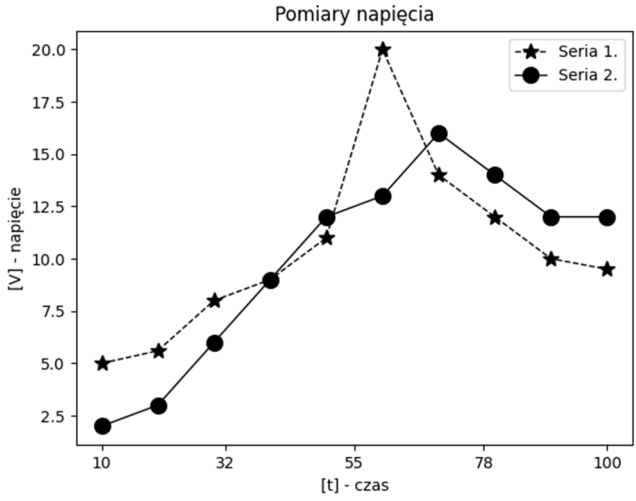
Wiersz oznaczony (*) zastępuje dziesięć wartości występujących na osi *X*:

```
osX=[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

ich mniejszą liczbą (pięć), na dodatek są one rozłożone równomiernie, zaczynając od wartości najmniejszej (10) i kończąc na największej (100), co czyni wykres bardziej czytelnym. Ta sztuczka może się też przydać do opisowej zmiany jednostek. Ponadto wartości są zaokrąglone do liczb całkowitych.

Możliwość modyfikacji etykiet na osiach jest bardzo użyteczna. Na przykład gdyby wartości osi *X* były znacznikami daty i czasu, to zamiast nich można by dać czytelne etykiety zakresu, np. nazwy dni tygodnia, kwartały.

Dla naszych danych efekt końcowy jest bardzo czytelny i miejmy nadzieję, że (jak by to powiedział ktoś z młodszego pokolenia Polaków) *aesthetic* (rysunek 6.3).



RYSUNEK 6.3. Wykresy wielokrotne

Popularne miary statystyczne

Zanim przystąpisz do jakiejkolwiek próby analizy danych, koniecznie odśwież sobie poniższe pojęcia. Nie są one zbyt skomplikowane, ale warto je dobrze rozumieć.

- **Mediana** to środkowy punkt całego zakresu danych, gdybyśmy je posortowali w kolejności.
- **Średnia arytmetyczna** to suma wszystkich wartości podzielona przez ich liczbę.
- **Moda** nie ma nic wspólnego z pokazami w Paryżu; w statystyce to po prostu liczba najczęściej występująca w zbiorze danych.

Przykład dla zbioru 13 liczb [5, 6, 10, **12**, **12**, **14**, **14**, 18, 20, 22, 25, 30, 40]:

Mediana: środkową wartością jest liczba 14. W przypadku parzystej liczby wartości mediana byłaby średnią arytmetyczną dwóch wartości w środku, np. dla serii 10, 12, 14, 20 byłoby to (12+14):2 = 13.

Średnia arytmetyczna: (5+6+10+12+12+14+14+18+20+22+25+30+40):13 = 19.

Moda: 12 i 14.

- **Wariancja** to średnia arytmetyczna kwadratów różnic między każdą wartością a średnią.
- **Odchylenie standardowe (σ)** to pierwiastek kwadratowy z wariancji. Odchylenie standardowe (ang. *standard deviation*) jest popularną miarą rozproszenia danych wokół średniej. Oznacza ono średnią odległość punktów danych od średniej arytmetycznej całego zestawu danych. Im większe odchylenie standardowe, tym większe rozproszenie danych, a im mniejsze odchylenie standardowe, tym dane są bardziej skoncentrowane wokół średniej. Liczymy je w następujący sposób: dysponując ciągiem *n* wartości i umawiając się, że x_i to poszczególne *wartości*, a μ to *średnia arytmetyczna*, otrzymamy urokliwy wzorek:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}}$$

Hm, a miało być mało wzorów… Ale ten jest naprawdę prosty i warto go znać, chyba nawet bardziej niż $E = mc^2$ (Albercie, wybacz mi tę profanację!).

Oto przykład obliczeniowy:

Zbiór: [4, 10, 12, 14, 20, 30].

Średnia: (4 + 10 + 12 + 14 + 20 + 30) : 6 = 15.

Wariancja: $(4-15)^2 + (10-15)^2 + (12-15)^2 + (14-15)^2 + (20-15)^2 + (30-15)^2 = 121 + 25 + 9 + 1 + 25 + 225 = \frac{406}{6} \approx 67{,}67$.

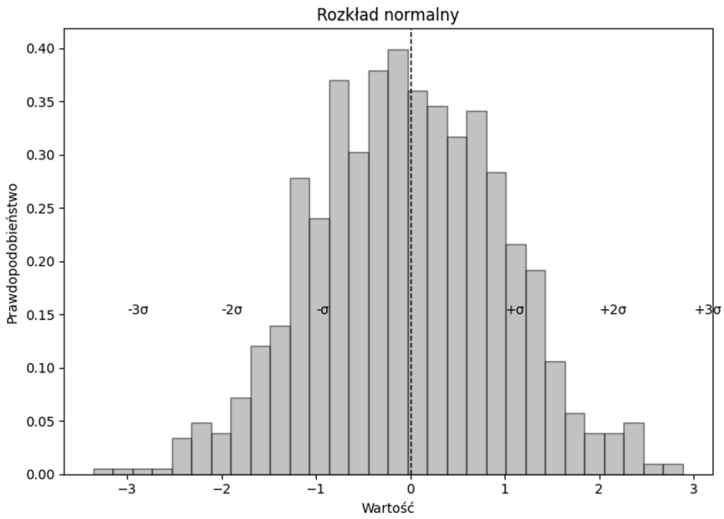
Odchylenie standardowe dla powyższych danych to pierwiastek kwadratowy z wariancji, czyli ok. 8,23.

Rozkład normalny warto znać!

Z premedytacją unikam w tej książce nadmiaru aparatu matematycznego i używania pojęć o zacięciu naukowym, ale termin **rozkład normalny** przewija się często w ML i powinniśmy go dobrze rozumieć.

Otóż rozkład normalny, zwany też rozkładem Gaussa, jest jednym z najważniejszych rozkładów prawdopodobieństwa w statystyce i analizie danych. Na wykresie charakteryzuje się cechami podsumowanymi na rysunku 6.4.

RYSUNEK 6.4.
Co to jest rozkład normalny?



W takim rozkładzie większość obserwacji skupiona jest wokół średniej, a rozkład danych jest symetryczny względem średniej (kształt zbliżony do dzwonu, z gładko opadającymi skrzydłami po obu stronach). Jest on zbudowany po wcześniejszym określeniu dwóch parametrów zbioru (zob. poprzedni podpunkt):

- Średnia (μ, *mikro*) — gdzie znajduje się szczyt krzywej?
- Odchylenie standardowe (σ, *mała sigma*) — jak „zbity" lub „rozciągnięty" jest rozkład wokół średniej?

Co ciekawe, mimo że jest to model matematyczny, zaobserwowano, że pasuje on też do wielu zjawisk występujących w naturze lub w danych empirycznych. Metryki tego rozkładu określają, że np. 68% obserwacji znajduje się w obrębie jednego odchylenia standardowego od średniej, 95% mieści się w dwóch odchyleniach standardowych, a 99,7% mieści się w trzech odchyleniach standardowych. Do tej interesującej właściwości jeszcze powrócimy.

Popularne wykresy analizy danych

Matplotlib oferuje nie tylko wykresy liniowe lub punktowe. Podczas pracy z dużymi tablicami, liczącymi setki lub tysiące wierszy, zachodzi często potrzeba przeprowadzenia nieskomplikowanych analiz rozkładu danych, co wymaga np. wyliczania sum cząstkowych wyników zebranych w danej kategorii. Analityka danych zainteresują też wizualizacje rozkładów statystycznych. Zapoznajmy się więc ze sposobami tworzenia kilku charakterystycznych typów wykresów, które doskonale nadają się do takich celów (słupkowe, histogramy i inne).

Wykresy słupkowe

W tym punkcie omówię sposób prezentowania danych w formie wykresu słupkowego (ang. *bar chart*). Analizując ten przykład, nauczysz się następujących elementów:

- używania innych serii danych niż liczbowe;
- definiowania wykresu słupkowego i ustalania stylu słupków;
- normalizowania wartości na osi *Y*.

W przypadku wykresu słupkowego do każdej kategorii musi być przypisana jedna wartość lub zestaw wartości (seria danych) — taki właśnie przykład pokażę dalej. Będziemy używali pięciu kategorii i trzech serii danych, każda z nich będzie zawierała pięć wartości, które będą reprezentowane przez słupki w każdej z pięciu kategorii. Oprócz kolorów, w bibliotece Matplotlib dostępne są różne desenie, które można wykorzystać do wypełnienia słupków poprzez ustawienie parametru `hatch` (tabela 6.4).

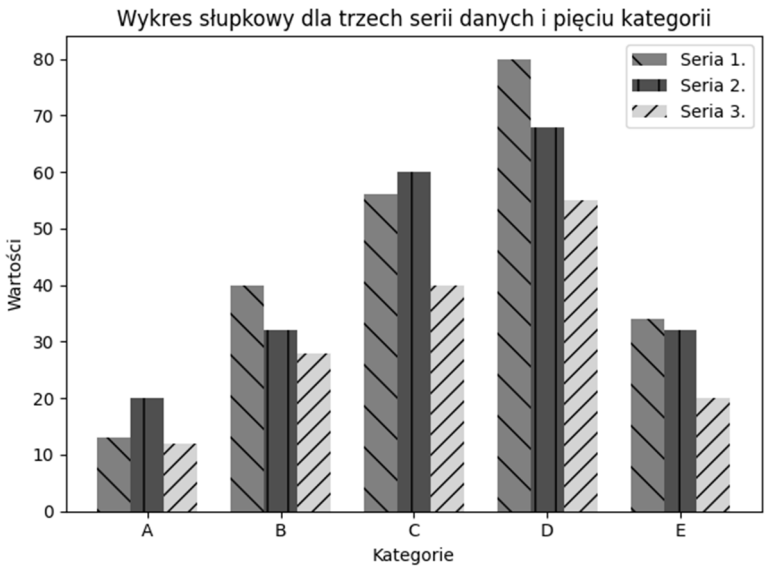
TABELA 6.4. Style deseni dostępne w Matplotlib

KOD	STYL
`'/'`	Linie ukośne
`'\'`	Linie ukośne w przeciwnym kierunku
`'|'`	Linie pionowe
`'-'`	Linie poziome
`'+'`	Krzyżyki
`'x'`	Krzyżyki skośne
`'o'`	Kropki
`'.'`	Małe kropki
`'*'`	Gwiazdki
`'0'`	Duże kropki
`'//'`	Linie podwójne ukośne
`'\\\|'`	Linie podwójne pionowe

Spójrzmy na przykładowy program. Użyta w kodzie funkcja `tight_layout()` automatycznie dostosowuje rozmiar elementów na wykresie, tak aby zapewnić, że nie nachodzą one na siebie, i aby całość była czytelna. Pomaga to uniknąć sytuacji, w której elementy (tytuły, etykiety osi itp.) są przycięte lub nachodzą na siebie, co może pogorszyć czytelność wykresu (*matplot3.py*).

```
import matplotlib.pyplot as plt
# Dane do wykresu
kategorie = ['A', 'B', 'C', 'D', 'E']
seria_1 = [13, 40, 56, 80, 34]
seria_2 = [20, 32, 60, 68, 32]
seria_3 = [12, 28, 40, 55, 20]
szer = 0.25  # Szerokość słupka
# Słupki dla pierwszej serii danych
plt.bar([x - szer for x in range(len(kategorie))],seria_1, width=szer, label='Seria 1.',
↪hatch='\\', color='grey')
# Słupki dla drugiej serii danych
plt.bar(range(len(kategorie)), seria_2, width=szer, label='Seria 2.', hatch='|',
↪color='green')
# Słupki dla trzeciej serii danych
plt.bar([x + szer for x in range(len(kategorie))], seria_3, width=szer, label='Seria 3.',
↪hatch='//', color='pink')
# Dodanie tytułu i etykiet osi
plt.title('Wykres słupkowy dla trzech serii danych i pięciu kategorii')
plt.xlabel('Kategorie')
plt.ylabel('Wartości')
plt.xticks(range(len(kategorie)), kategorie) # Ustawienie etykiet na osi X (*)
plt.legend()
plt.tight_layout()
plt.show() # Wyświetlenie wykresu
```

Efekt końcowy jest bardzo czytelny (rysunek 6.5).



RYSUNEK 6.5. Wykres słupkowy

Jak łatwo zauważyć, funkcję umownej osi *X* pełni tu zbiór etykiet, czyli nazwy kategorii testów, a wyniki na osi *Y* są wyśrodkowane względem tych etykiet.

Jeśli uruchomisz program bez wiersza oznaczonego gwiazdką, to na osi pionowej zostaną wyświetlone liczby 0, 1, 2, 3 i 4 (indeksy kategorii). Stąd zastosowana normalizacja etykiet na tej osi (metoda `xticks()`).

Histogramy

Histogram dzieli zakres wartości danych na kilka równych przedziałów i przedstawia *liczbę obserwacji* należących do każdego przedziału za pomocą słupków. Histogramy zbliżają nas jeszcze bardziej do analizy statystycznej, pomagając zrozumieć dystrybucję (rozkład) danych według ich wartości. O ile w przypadku małych zbiorów danych łatwo to zobaczyć, analizując surowe dane, gdzie widać, które wartości są bardziej popularne, a które występują rzadziej, o tyle w przypadku tabeli z np. tysiącem wartości ocena w trybie „na oko" mogłaby prowadzić do błędów i konieczności wizyty u optyka!

Podsumowując, w histogramie:

- oś *X* reprezentuje zakres wartości danych podzielony na przedziały (tzw. kubełki, ang. *bin*);
- oś *Y* reprezentuje liczbę obserwacji w każdym przedziale.

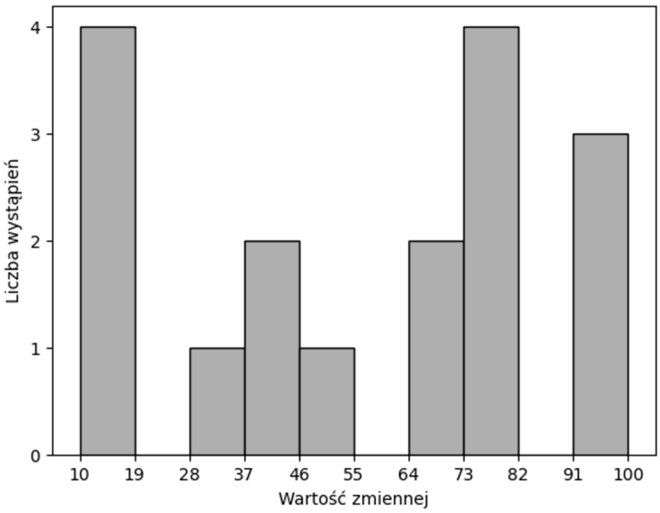
Histogramy są intuicyjnie łatwe do przyswojenia i bardzo przydatne do zrozumienia rozkładu danych oraz występujących w nich wzorców (kształt, tendencje do grupowania, obecność wartości odstających — ang. *outlier*). Histogramy dobrze uwypuklają wszelkie asymetrie rozkładu.

Popatrzmy zatem, jak tworzy się wykres typu histogram przy użyciu biblioteki Matplotlib. Załóżmy, że mamy zebranych 17 wyników dość podobnych do siebie i chcemy utworzyć histogram, który je przenalizuje i pokaże, które z nich występują częściej, a które rzadziej — niech wykres rozrzuci te podsumowania na dziesięć umownych „kubełków" (*matplot4.py*).

```
from matplotlib import pyplot as plt
import numpy as np
t=np.array([10,10,10,10,30,40,40,50,70,70,80,80,80,80,100,100, 100])
# Tworzenie histogramu i pobranie informacji o krotnościach
counts,bins,_=plt.hist(t, bins=10, color='orange', edgecolor='black')
# Obliczanie środków kubełków
bin_centers = 0.5 * (bins[:-1] + bins[1:])
# Ustawienie etykiet na osi X na granicach kubełków
plt.xticks(bins)
# Ustalanie zakresu etykiet na osi Y zgodnie z rzeczywistymi krotnościami
plt.yticks(np.arange(0, max(counts) + 1, step=1))
plt.xlabel('Wartość zmiennej')
plt.ylabel('Liczba wystąpień')
plt.show() # Wyświetlenie histogramu
```

Efekt końcowy pokazuje rysunek 6.6.

RYSUNEK 6.6.
Histogram ułatwia analizę rozkładu wartości danych w serii



Jak łatwo zauważyć, funkcję umownej osi *X* pełni tu z góry ustalona liczba „kubełków", czyli zakresów wartości, natomiast na osi *Y* możemy odczytać liczbę wartości zliczonych dla bieżącego kubełka (np. 4 wartości dla pierwszego zakresu, 3 dla ostatniego).

Wykres skrzypcowy

Wykres skrzypcowy (ang. *violin plot*) jest podobny do histogramu, ale daje dodatkowe informacje o gęstości rozkładu danych, czyli stanowi połączenie informacji, które można odczytać z histogramu i z wykresu pudełkowego. Wykres ten pokazuje rozkład danych w sposób zbliżony do histogramu i gęstość prawdopodobieństwa w różnych przedziałach wartości.

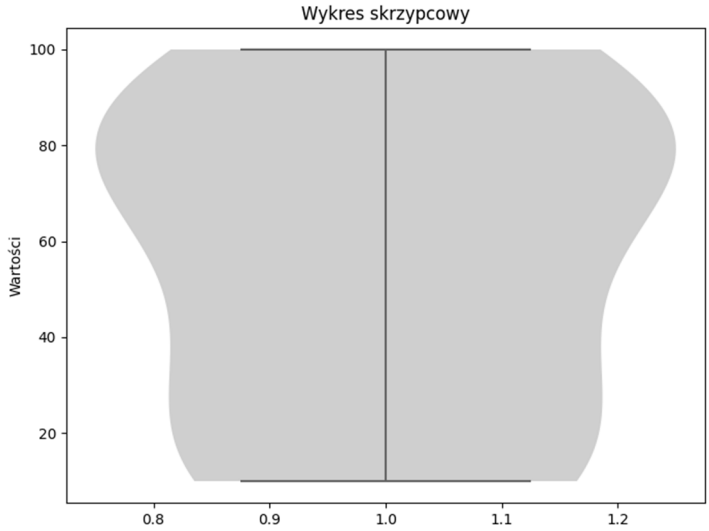
Wynikowy wykres skrzypcowy pokaże rozkład wartości danych zbioru wejściowego i można go wyświetlić pionowo lub poziomo. Każdy „skrzypek" reprezentuje rozkład wartości w danym przedziale. Użyjmy teraz nieco innego zbioru danych o odmiennym rozkładzie (*matplot5.py*):

```
import numpy as np
import matplotlib.pyplot as plt
t=np.array([-10,-5,0,10,12,15,17,60,65,70,80,80,80,80,85,86,90])
plt.violinplot(t, vert=True)
plt.ylabel('Wartości')
plt.title('Wykres skrzypcowy')
plt.show()
```

Popatrz, co wyświetli program po uruchomieniu (rysunek 6.7).

Kreski na początku i końcu obszaru skrzypcowego oznaczają minimalną i maksymalną wartość w tym przedziale danych.

RYSUNEK 6.7.
Wykres skrzypcowy



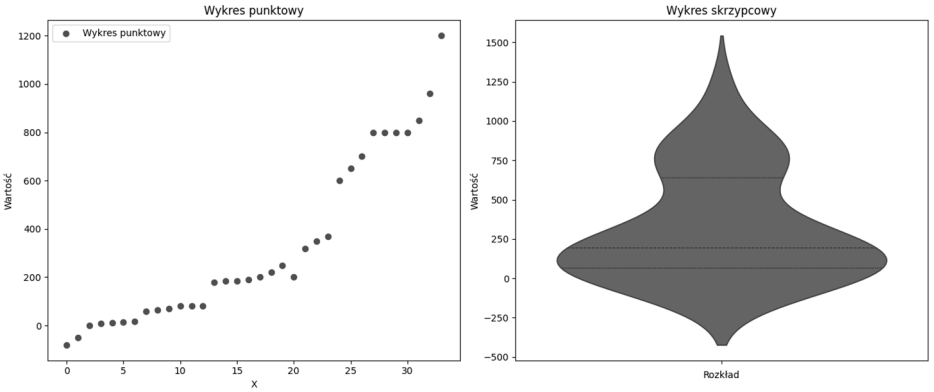
W układzie pionowym (`vert=True`) oś Y reprezentuje wartości zmiennych w analizowanym zbiorze danych, a nie gęstość rozkładu wartości. Im wyższy punkt na osi Y w danym obszarze, tym wyższe są wartości danych w tym przedziale. Natomiast szerokość obszaru dla danej wartości na osi Y pokazuje na osi X, jak gęsto są skupione wartości danych wokół tej wartości. U nas występują dwa główne skupiska wartości. Pierwsze skupisko, które jest szersze, występuje w okolicach wartości 80, drugie, mniejsze skupisko znajduje się nieco niżej i również jest szerokie, co oznacza gęste skupienie danych wokół tej niższej wartości. Zawężenie skrzypiec w okolicach 40 sugeruje mniejszą gęstość danych w tym przedziale.

Wykres skrzypcowy nadaje się do wizualizacji tzw. zbiorów wielomodalnych, w których występują grupy danych i dla każdej z nich osobno można zidentyfikować parametry rozkładu i najczęściej występujące wartości.

Przykład zawarłem w pliku *matplot6.py* (musisz mieć zainstalowaną bibliotekę `seaborn`):

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
t = np.array([-80, -50, 0, 10, 12, 15, 17, 60, 65, 70, 80, 80, 80, 180, 185, 186, 190,
200, 220, 250, 200, 320, 350, 370, 600, 650, 700, 800, 800, 800, 800, 850, 960, 1200])
fig, axes = plt.subplots(1, 2, figsize=(14, 6)) # Podwykresy
axes[0].scatter(np.arange(len(t)), t, color='green', label='Wykres punktowy')
axes[0].set_title('Wykres punktowy')
axes[0].set_xlabel('X')
axes[0].set_ylabel('Wartość')
axes[0].legend()
sns.violinplot(ax=axes[1], data=t, inner='quartile') # Wykres skrzypcowy w seaborn
axes[1].set_title('Wykres skrzypcowy')
axes[1].set_xlabel('Rozkład')
axes[1].set_ylabel('Wartość')
plt.tight_layout() # Wyświetlenie wykresów
plt.show()
```

Efekt zobaczymy na rysunku 6.8, na którym wyświetliłem obok siebie wykresy punktowy i skrzypcowy dla danych zawartych w skrypcie. Wykrycie w takim zestawie, zwłaszcza jeśli jest bardzo liczny, zbioru bi- lub N-modalnego jest łatwiejsze na wykresie skrzypcowym (wystarczy zidentyfikować dwa „pagórki" danych).



RYSUNEK 6.8. Połączenie wykresu punktowego ze skrzypcowym wzbogaca możliwość analizy zbiorów danych

Wykres pudełkowy

Wykres pudełkowy (ang. *boxplot*) wygląda na wyższą szkołę jazdy, z nieujawnionej jeszcze przez serwisy streamingowe serii „Szybcy i wściekli analitycy danych", ale po przyjrzeniu się okazuje się możliwy do opanowania także przez laików.

Spójrz na kod z przykładu *matplot7.py*:

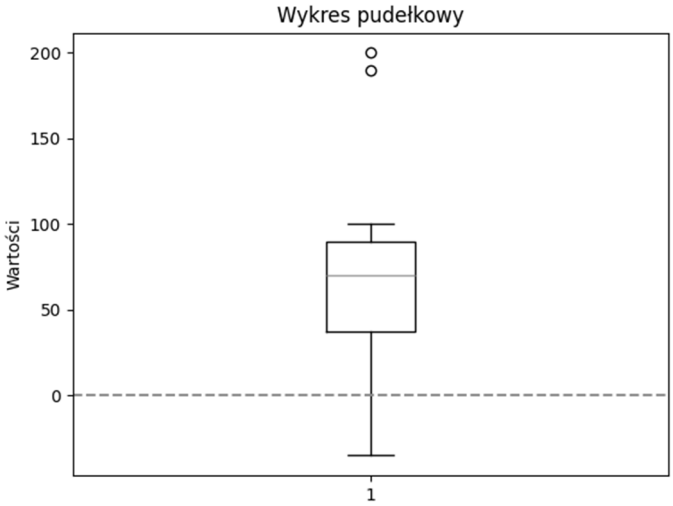
```
import matplotlib.pyplot as plt
import numpy as np
# Przykładowe dane
t = np.array([-35, -30, 10, 10, 35, 40, 40, 50, 70, 70, 80, 80, 80, 80, 100, 100, 100,
190, 200])
# Tworzenie wykresu pudełkowego
plt.boxplot(t)
# Dodanie tytułu i etykiety osi
plt.title('Wykres pudełkowy')
plt.ylabel('Wartości')
plt.axhline(0, color='c', linestyle='--') # Linia pozioma dla Y = 0
plt.show() # Wyświetlenie wykresu
```

Ten nieskomplikowany skrypt wyświetli wykres pokazany na rysunku 6.9.

Jak go zinterpretować?

- Prostokątna skrzynka (pudełko) oraz linie (wąsy) reprezentują rozkład wartości obserwacji w zbiorze danych.
- Dolna krawędź pudełka reprezentuje pierwszy kwartyl (Q1), czyli 25. percentyl danych. (Percentyl to miara pozycji w rozkładzie statystycznym, która określa, jaki procent danych znajduje się poniżej danej wartości).

RYSUNEK 6.9.
Wykres pudełkowy



- Środkowa linia pudełka reprezentuje medianę (drugi kwartyl, Q2), czyli 50. percentyl danych (50% danych znajduje się poniżej wartości mediany).
- Górna krawędź pudełka reprezentuje trzeci kwartyl (Q3), czyli 75. percentyl danych.
- Wysokość pudełka to zakres między pierwszym a trzecim kwartylem (ang. *interquartile range*, IQR), czyli odległość między dolną a górną krawędzią pudełka.
- Dolny wąs reprezentuje dolną granicę danych (niebędących outlierami), zazwyczaj 1,5 * IQR poniżej pierwszego kwartyla.
- Górny wąs reprezentuje górną granicę danych (niebędących outlierami), zazwyczaj 1,5 * IQR powyżej trzeciego kwartyla.
- Wartości poza granicami wąsów są potencjalnymi outlierami.
- Punkty znajdujące się poza granicami wąsów są uważane za odstające (outliery) i mogą wskazywać na wartości nietypowe lub błędy pomiarowe w danych.

Uff… Ale pomijając już pierwszy szok poznawczy, na pewno zgodzisz się, że ten wykres w interesujący sposób prezentuje rozkład danych i ułatwia analizę pod kątem anomalii rozkładu (zbyt duże koncentracje) oraz identyfikację wartości odstających, tzw. *outlierów*. To właśnie nietypowe, odstające wartości mogą łatwo popsuć jakość modelu ML (np. regresji).

Mapa cieplna

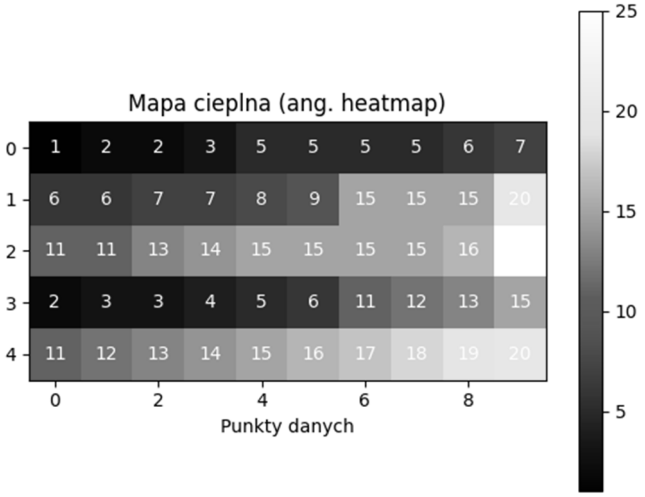
Wykres zwany mapą cieplną (ang. *heatmap*) wykorzystywany jest do prezentacji macierzy danych wizualizujących wartości numeryczne za pomocą kolorów. Jest to przydatne do analizy korelacji między zmiennymi. Każdy wiersz w macierzy danych odpowiada jednej serii danych. Każdy element wiersza reprezentuje wartość w danej serii w konkretnym punkcie danych.

Popatrzmy na przykład (*matplot8.py*):

```
import numpy as np
import matplotlib.pyplot as plt
# Przykładowe serie po 10 wartości
data = np.array(
    [[1,  2,  2,   3, 5,  5,   5,  5,  6,  7],
     [6,  6,  7,   7, 8,  9,  15, 15, 15, 20],
     [11, 11, 13, 14, 15, 15, 15, 15, 16, 25],
     [2,  3,   3,  4, 5,   6, 11, 12, 13, 15],
     [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
    ])
# Tworzenie mapy cieplnej
plt.imshow(data, cmap='hot', interpolation='nearest')
# Dodanie deseni do poszczególnych punktów danych
for i in range(data.shape[0]):
    for j in range(data.shape[1]):
        plt.text(j, i, str(data[i, j]), ha='center', va='center', color='white')
# Dodanie kolorowej skali
plt.colorbar()
plt.xlabel('Punkty danych')
plt.title('Mapa cieplna (ang. heatmap)')
plt.show() # Wyświetlenie mapy cieplnej
```

Nasz skrypt wyświetli wykres pokazany na rysunku 6.10.

RYSUNEK 6.10.
Mapa cieplna



Widać, że dla pierwszej serii danych wartości są niskie (ciemne kolory), a dla ostatniej są wyraźnie cieplejsze. Na wykresie, oprócz samej kolorowej mapy cieplnej, dodałem też desenie do każdej jej komórki, wyświetlając wartość danej liczbowej w środku każdej komórki (nie zawsze się to sprawdza i w swoich programach możesz pominąć ten element szablonu kodu). Desenie są białe, co pomaga je czytelnie wyświetlić na tle kolorowego tła mapy cieplnej.

Podręcznik Matplotlib na bezludną wyspę?

Tak jak wspomniałem, dokumentacja Matplotlib jest olbrzymia i nie da się jej powielić w sposób racjonalny. Dlatego aby wytłumaczyć filozofię używania tej biblioteki, postanowiłem zaznajomić Cię z kilkoma schematami tworzenia i upiększania wykresów. Autorzy oferują też podręcznik PDF, który znajdziesz pod adresem *https://matplotlib.org/Matplotlib.pdf* — załaduj na dysk dokument zajmujący już niemal 5000 stron (to nie jest żart).

Oczywiście nie musisz go drukować w celu zabrania na bezludną wyspę, jeśli takie masz plany życiowe. Wystarczy, że wyspa znajduje się w zasięgu internetu — wówczas wystarczy zapisać w przeglądarce zakładkę *https://matplotlib.org* i sięgać tam w razie potrzeby do wersji bieżącej (odsyłacze *Tutorials*, *User guide*). Alternatywnie zajrzyj na stronę *https://matplotlib.org/stable/users/index* i przeanalizuj gotowe wzorcowe skrypty budujące wykresy różnych typów. Do czego gorąco zachęcam, bo efekty mogą się okazać spektakularne!

Skorowidz

G

H

I

K

L

M

Wejdź na nowy poziom programowania z ML i NLP

Sztuczna inteligencja stale się rozwija. Właściwie codziennie słyszymy o jej rosnących możliwościach, nowych osiągnięciach i przyszłości, jaką nam przyniesie. Jednak w tej książce skupiamy się nie na przyszłości, a na teraźniejszości i praktycznym obliczu AI — na usługach, które świadczy już dziś. Większość najciekawszych zastosowań sztucznej inteligencji bazuje na ML (uczenie maszynowe, ang. *machine learning*), NLP (przetwarzanie języka naturalnego, ang. *natural language processing*) i architekturze RAG (ang. *retrieval augmented generation*) zwiększającej możliwości tzw. dużych modeli językowych (LLM, ang. *large language model*). Stanowią one podwaliny budowy systemów AI, bez których te systemy często wcale nie mogłyby powstać.

Do niedawna ML i NLP pozostawały domeną badaczy i specjalistów — znajdowały się poza zasięgiem praktyków programowania. Aktualnie jest inaczej, szybkie komputery, pojemne pamięci RAM i zaawansowane procesory pozwalają stosować te technologie w codziennej pracy programisty. Szczególnie programisty języka Python, do którego są one niemal „naturalnie" przypisane. Mało tego, od kodujących w Pythonie coraz częściej wręcz wymaga się umiejętności znajomości obszaru AI.

Tym bardziej warto sięgnąć po ten podręcznik z ćwiczeniami, dzięki któremu między innymi:

- **Dowiesz się, jak używać Pythona do rozwiązywania problemów AI**
- **Poznasz tajniki analizy tekstów, analizy sentymentu**
- **Zrozumiesz, jak skutecznie używać algorytmów klasyfikacji, regresji i grupowania do rozwiązywania problemów biznesowych**
- **Pokonwersujesz z ChatGPT — i to bez wchodzenia na stronę internetową tego serwisu**

Helion

helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-1580-0

9 788328 915800

Cena: 79,00 zł