O'REILLY®

Uczenie maszynowe z użyciem Scikit-Learn i PyTorch

Koncepcje, narzędzia i techniki umożliwiające konstruowanie inteligentnych systemów

Helion

Aurélien Géron

Tytuł oryginału: Hands-On Machine Learning with Scikit-Learn and PyTorch: Concepts, Tools, and Techniques to Build Intelligent Systems

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-289-3772-7

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Nasz pierwszy projekt uczenia maszynowego

W tym rozdziale stworzymy od początku do końca przykładowy projekt uczenia maszynowego — będziemy udawać, że jesteśmy świeżo zatrudnionymi analitykami danych w agencji handlu nieruchomościami. Projekt ten jest fikcyjny; jego zadaniem jest ukazanie poszczególnych etapów przygotowywania projektu uczenia maszynowego, a nie przekazywanie wiedzy na temat obrotu nieruchomościami. Aby nam się udało, musimy wykonać następujące czynności:

1. Przeanalizować całokształt czekającego nas zadania.
2. Uzyskać dane.
3. Przeanalizować i zwizualizować dane w celu rozpoznania wzorców i dodatkowych informacji.
4. Przygotować dane pod względem algorytmów uczenia maszynowego.
5. Wybrać i wyuczyć model.
6. Dostroić model.
7. Zaprezentować rozwiązanie.
8. Uruchomić, monitorować i utrzymywać system.

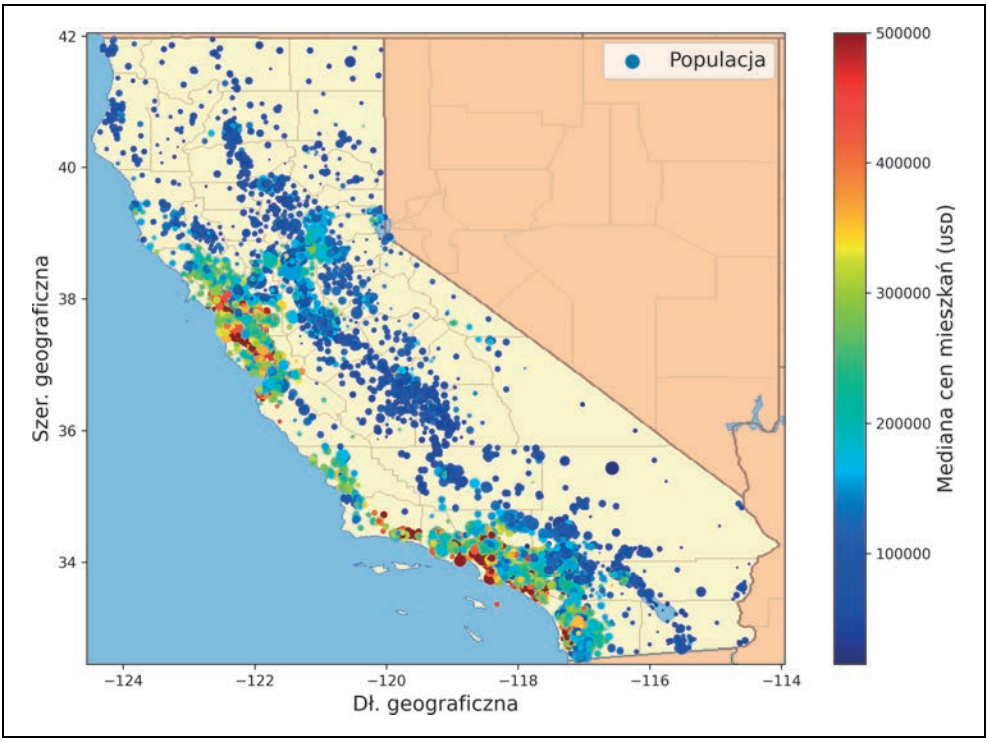
Praca z rzeczywistymi danymi

Podczas poznawania zagadnień uczenia maszynowego najlepiej jest eksperymentować na rzeczywistych, a nie sztucznych danych. Na szczęście są dostępne tysiące otwartych zbiorów danych, opisujące chyba wszelkie możliwe dziedziny życia. Poniżej wymieniam kilka popularnych repozytoriów otwartych danych, z których możesz skorzystać:

- wyszukiwarka Google Datasets Search (*https://datasetsearch.research.google.com*),
- zestawy danych Hugging Face (*https://huggingface.co/docs/datasets*),
- OpenML.org (*https://openml.org*),
- Kaggle.com (*https://kaggle.com/datasets*),

- repozytorium uczenia maszynowego UC Irvine (*https://archive.ics.uci.edu*),
- kolekcja dużych zestawów sieciowych Uniwersytetu Stanforda (*https://snap.stanford.edu/data*),
- zestawy danych na platformie Amazon AWS (*https://registry.opendata.aws/*),
- otwarte dane rządu USA (*https://data.gov*),
- DataPortals.org (*https://dataportals.org/*),
- lista zestawów danych do uczenia maszynowego na Wikipedii (*https://homl.info/9*).

Na użytek tego rozdziału wykorzystamy zestaw danych California Housing Prices, stanowiący część repozytorium StatLib[1] (rysunek 2.1). Dane te zostały opracowane na podstawie spisu ludności Kalifornii z 1990 roku. Nie są one już zbyt aktualne (w tamtym czasie można było nabyć dom w Bay Area za całkiem rozsądne pieniądze), ale pod wieloma względami mają one charakter edukacyjny, dlatego będziemy udawać, że są całkiem świeże. W celach dydaktycznych dodałem również atrybut kategorialny i usunąłem kilka cech.



Rysunek 2.1. Ceny domów w Kalifornii

[1] Pierwotnie zbiór tych danych pojawił się w artykule R. Kelleya Pace'a i Ronalda Barry'ego *Sparse Spatial Autoregressions*, „Statistics & Probability Letters 33", nr 3 (1997), s. 291 – 297.

Przeanalizuj całokształt projektu

Witaj w Korporacji Inteligentne Nieruchomości! Twoim pierwszym zadaniem jest wykorzystanie danych spisu ludności Kalifornii do stworzenia modelu cen mieszkań w tym stanie. Dane te zawierają takie informacje jak wielkość populacji, mediana dochodów, mediana cen mieszkań — wszystko rozdzielone pomiędzy poszczególne grupy bloków Kalifornii. Grupami bloków nazywamy najmniejsze jednostki terytorialne, dla których są publikowane próbki danych ze spisu ludności (zazwyczaj na typową grupę bloków składa się od 600 do 3000 osób). Dla uproszczenia będę nazywał je „dystryktami".

Twój model powinien się uczyć z tych danych i przy użyciu pozostałych wskaźników być w stanie przewidywać medianę cen mieszkań w dowolnym dystrykcie.

Skoro jesteśmy poukładanymi analitykami danych, naszą pierwszą czynnością powinno być przygotowanie listy kontrolnej projektu. Możesz zacząć od tej dostępnej w materiałach dodatkowych na stronie *https://ftp.helion.pl/przyklady/umaszs.zip*, *p*owinna się ona nadawać do większości projektów uczenia maszynowego, nie zapomnij jednak dostosować jej do własnych potrzeb. W niniejszym rozdziale zajmiemy się wieloma elementami umieszczonymi na tej liście, ale też pominiemy niektóre, gdyż albo nie wymagają objaśnienia, albo zostaną omówione w następnych rozdziałach.

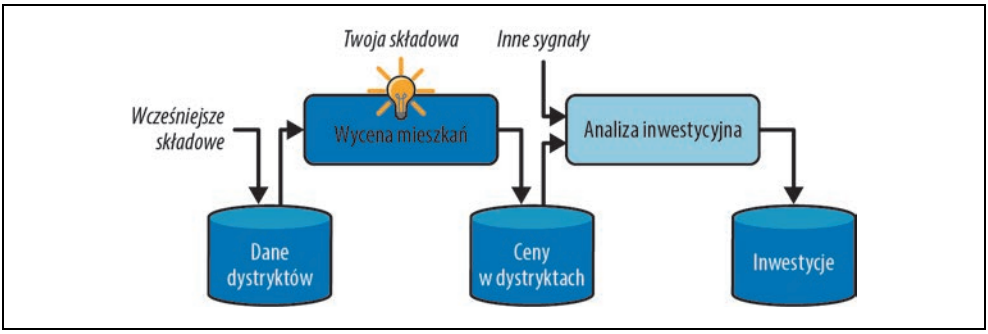
Określ zakres problemu

Pierwszym pytaniem, jakie należałoby zadać przełożonemu, byłoby: „Jaki jest dokładny cel biznesowy?"; utworzenie modelu prawdopodobnie nie jest ostatecznym celem samym w sobie. W jaki sposób firma zamierza skorzystać i zarobić na tym modelu? Znajomość celu jest istotna, ponieważ pomaga określić zakres problemu, dobrać odpowiednie algorytmy oraz miarę wydajności służącą do oceny wydajności modelu, a także pozwala oszacować wysiłek potrzebny do zoptymalizowania działania modelu.

Twój szef odpowiada, że wynik tego modelu (prognoza mediany cen mieszkań w danym dystrykcie) będzie kluczowy dla określenia, czy warto inwestować na danym obszarze. Dokładniej mówiąc, wynik modelu zostanie przekazany do innego systemu uczenia maszynowego (rysunek 2.2) wraz z innymi sygnałami[2]. Dlatego ważne jest, aby nasz model przewidywania cen mieszkań był jak najbardziej dokładny.

Następnie należałoby zapytać szefa, jak wygląda obecne rozwiązanie (jeśli w ogóle jakieś istnieje). Często możesz uzyskać w ten sposób dostęp do referencyjnej wydajności, a także podpowiedzi ułatwiające rozwiązanie problemu. Twój szef odpowiada, że obecnie ceny domów są szacowane ręcznie przez zespół ekspertów: gromadzą oni aktualne informacje o dystryktach, a gdy nie są w stanie wyliczyć mediany cen mieszkań, korzystają ze skomplikowanych reguł.

[2] Zgodnie z teorią informacji, którą Claude Shannon, zdefiniował w firmie Bell Labs w celu poprawy jakości telekomunikacji, **sygnałem** nazywamy informację przekazywaną systemowi uczenia maszynowego. Zgodnie z tą teorią zależy nam na uzyskaniu wysokiej wartości stosunku sygnału do szumu.



Rysunek 2.2. Potok uczenia maszynowego używany w branży nieruchomości

Jest to bardzo kosztowna i czasochłonna czynność, a uzyskane tą metodą oszacowania nie są wcale takie dokładne; często w sytuacjach, gdy analitykom udaje się wyliczyć medianę cen mieszkań, okazuje się, że pomylili się w oszacowaniach nawet o ponad 30%. Dlatego zarząd firmy uważa, że warto byłoby wyuczyć model przewidujący medianę cen mieszkań w danym dystrykcie za pomocą innych danych opisujących dany dystrykt. Znakomitym źródłem potrzebnych informacji okazuje się spis ludności, ponieważ możemy w nim znaleźć medianę cen mieszkań wyliczoną dla tysięcy dystryktów, a także wiele innych danych.

Potoki

Potokiem (ang. *pipeline*) danych nazywamy sekwencję **składowych** przetwarzanych danych. Potoki są bardzo popularne w systemach uczenia maszynowego, ponieważ manipulujemy w nich olbrzymią ilością danych oraz przeprowadzamy na nich wiele przekształceń.

Składowe są zazwyczaj przetwarzane asynchronicznie. Każda składowa pobiera znaczną ilość danych, przetwarza je i umieszcza wyniki w innym magazynie informacji. Po pewnym czasie następna składowa potoku pobiera te przetworzone wyniki i generuje z nich kolejne. Każda składowa jest w znacznej mierze uniezależniona od pozostałych; jedynym interfejsem pomiędzy poszczególnymi składowymi jest magazyn danych. W ten sposób uzyskujemy prosty do zrozumienia system (za pomocą grafów przepływu danych), a poszczególne zespoły mogą się skoncentrować na własnych składowych. Poza tym awaria danej składowej nie wpływa na pracę następnych składowych (przynajmniej przez pewien czas), gdyż mogą one korzystać z ostatniego prawidłowego wyniku wygenerowanego przez uszkodzoną składową. Dzięki temu taka architektura jest dość odporna.

Z drugiej strony przy braku odpowiednich mechanizmów monitorowania uszkodzona składowa może przez dłuższy czas niezauważona dostarczać nieprawidłowe wyniki. Dane stają się nieaktualne i spada ogólna wydajność systemu.

Po uzyskaniu tych informacji możesz się zająć projektowaniem systemu. Najpierw musisz określić rodzaj nadzorowania uczenia modelu: czy realizowane zadanie jest nienadzorowane, nadzorowane, półnadzorowane, samonadzorowane, czy może należy użyć uczenia przez wzmacnianie? Jest to zadanie klasyfikacji, regresji czy jeszcze jakieś inne? Powinniśmy korzystać z technik

uczenia przyrostowego czy wsadowego? Zanim przejdziemy dalej, przerwij na chwilę czytanie i postaraj się samodzielnie odpowiedzieć na te pytania.

Masz już odpowiedzi? Zastanówmy się. Z pewnością mamy tu do czynienia z klasycznym zadaniem uczenia nadzorowanego, ponieważ można wytrenować model za pomocą *oznakowanych* przykładów uczących (każdy przykład ma zdefiniowany od razu oczekiwany wynik, np. medianę cen mieszkań w dystrykcie). Jest to klasyczne zadanie regresyjne, ponieważ model ma przewidzieć jakąś wartość. Mówiąc dokładniej, stajemy przed problemem **regresji wielorakiej** (ang. *multiple regression*), gdyż do prognozowania wyniku nasz system wykorzysta wiele cech (populację dystryktu, medianę dochodów itd.). Możemy także traktować ten problem jako zadanie **regresji jednoczynnikowej** (ang. *univariate regression*), ponieważ dla każdego dystryktu staramy się przewidzieć tylko pojedynczą wartość. Gdybyśmy próbowali prognozować wiele wartości dla każdego dystryktu, mielibyśmy do czynienia z **regresją wieloczynnikową** (ang. *multivariate regression*). Na koniec zwróćmy uwagę, że dane nie będą dostarczane w ciągły sposób, nie będziemy musieli dynamicznie dostosowywać systemu do zmieniających się danych, a same dane są wystarczająco małe, aby zmieścić się w pamięci, dlatego powinna nam wystarczyć zwykła, wsadowa metoda uczenia.

Gdyby dane miały jednak ogromne rozmiary, mógłbyś rozdzielić czynności uczenia wsadowego pomiędzy kilka serwerów (przy użyciu techniki MapReduce) albo zastąpić je po prostu mechanizmem uczenia przyrostowego.

Wybierz wskaźnik wydajności

Kolejnym etapem jest dobór wskaźnika wydajności. W przypadku zagadnień regresyjnych klasyczną miarą wydajności jest **pierwiastek błędu średniokwadratowego** (ang. *Root Mean Square Error* — RMSE). Dowiadujemy się dzięki niemu, w jakim stopniu model myli się w przewidywaniach — wraz ze wzrostem wartości błędu rośnie również waga tego wskaźnika. Równanie 2.1 przedstawia wzór matematyczny używany do wyliczania błędu RMSE.

Równanie 2.1. Pierwiastek błędu średniokwadratowego (RMSE)

$$\mathrm{RMSE}(\boldsymbol{X}, \mathbf{y}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2}$$

Notacje

Powyższy wzór zawiera kilka najpowszechniejszych notacji używanych w uczeniu maszynowym, z których będę korzystać w dalszej części książki:

- Wartość *m* określa liczbę elementów zbioru uczącego, wobec których będziemy mierzyć błąd RMSE,
  - przykładowo jeśli chcesz obliczyć błąd RMSE dla zbioru walidacyjnego 2000 dystryktów, to *m* = 2000.

- Wartość $\boldsymbol{x}^{(i)}$ stanowi wektor wartości wszystkich cech (z pominięciem etykiet) *i*-tego przykładu, natomiast $y^{(i)}$ stanowi etykietę tejże próbki (czyli pożądaną wartość wyniku dla danego przykładu). **y** to wektor zawierający etykiety wszystkich przykładów w zestawie danych.
  - na przykład jeśli pierwszy dystrykt w zbiorze danych znajduje się na długości geograficznej –118,29°W i szerokości geograficznej 33,91°N, a do tego wiemy, że jego populacja wynosi 1416 mieszkańców przy średnim dochodzie 38 372 dolarów, natomiast mediana ceny mieszkań na tym obszarze wynosi 156 400 dolarów (na razie ignorujemy pozostałe cechy), to możemy zapisać te dane w następujący sposób:

$$\boldsymbol{x}^{(1)} = \begin{pmatrix} -118{,}29 \\ 33{,}91 \\ 1416 \\ 38372 \end{pmatrix}$$

i

$y^{(1)} = 156400$.

- Wartość $\boldsymbol{X}$ oznacza macierz zawierającą wartości wszystkich cech (z pominięciem etykiet) wszystkich przykładów składających się na zbiór danych. Dla każdego przykładu przeznaczony jest jeden wiersz, natomiast *i*-ty wiersz stanowi transpozycję wektora $\boldsymbol{x}^{(i)}$; zwróć uwagę na zapis[3] $(\boldsymbol{x}^{(i)})^T$,
  - na przykład zgodnie z podanym powyżej opisem pierwszego dystryktu macierz $\boldsymbol{X}$ będzie wyglądała następująco:

$$X = \begin{pmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \vdots \\ \left(x^{(1999)}\right)^T \\ \left(x^{(2000)}\right)^T \end{pmatrix} = \begin{pmatrix} -118{,}29 & 33{,}91 & 1416 & 38372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- Parametr *h* jest funkcją prognozującą naszego systemu (bywa ona zwana także **hipotezą**). Po wprowadzeniu wartości wektora cech $\boldsymbol{x}^{(i)}$ do systemu zostaje wygenerowana prognozowana wartość $\hat{y}^{(i)} = h(\boldsymbol{x}^{(i)})$ dla tego przykładu (symbol $\hat{y}$ nazywamy „igrekiem z daszkiem”),
  - przykładowo jeśli Twój system wyliczy, że mediana cen mieszkań w pierwszym dystrykcie wynosi 158 400 dolarów, to $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158400$. Wartość błędu predykcyjnego dla tego dystryktu to $\hat{y}^{(1)} - y^{(1)} = 2000$.
- Zapis $RMSE(\boldsymbol{X},y,h)$ oznacza funkcję kosztu mierzoną dla zbioru uczącego za pomocą hipotezy *h*.

Wartości skalarne (np. *m* lub $y^{(i)}$), a także nazwy funkcji (np. *h*) zapisujemy małymi literami i kursywą, symbole pogrubione (np. $\boldsymbol{x}^{(i)}$) oznaczają wektory, natomiast duże, pogrubione litery (np. $\boldsymbol{X}$) są zarezerwowane dla macierzy.

[3] Przypominam, że operator transpozycji przekształca wiersze na kolumny i odwrotnie.

Pomimo że preferowanym wskaźnikiem wydajności w zadaniach regresyjnych jest pomiar błędu RMSE, w pewnych sytuacjach lepiej sprawdza się inna funkcja, zwłaszcza jeśli w zestawie danych znajduje się wiele przykładów odstających od reszty, gdyż RMSE jest dość wrażliwy na ich obecność. W takim przypadku warto byłoby wyliczyć **średni absolutny błąd** (ang. *Mean Absolute Error* — MAE; zwany jest także średnim odchyleniem bezwzględnym, ang. *Average Absolute Deviation*), zaprezentowany w równaniu 2.2:

Równanie 2.2. Średni absolutny błąd (MAE)

$$MAE(\boldsymbol{X}, \boldsymbol{y}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(x^{(i)}\right) - y^{(i)}\right|$$

Obydwie miary (RMSE i MAE) pozwalają na obliczanie odległości pomiędzy dwoma wektorami: wektorem prognoz i wektorem wartości docelowych. Dostępne są różne miary odległości (tzw. **normy**):

- Obliczenie pierwiastka błędu średniokwadratowego (RMSE) wiąże się z **normą euklidesową** — stanowi ona doskonale nam wszystkim znaną notację odległości. Zwana jest ona także **normą ℓ_2** i jest zapisywana w postaci wyrażenia $\|\cdot\|_2$ (lub po prostu $\|\cdot\|$).
- Obliczenie średniego absolutnego błędu (MAE) wiąże się z **normą ℓ_1**, zapisywaną jako wyrażenie $\|\cdot\|_1$. Jest ona nazywana również **normą taksówkową**, **miejską** oraz **Manhattan**, ponieważ mierzy ona odległość pomiędzy dwoma punktami w mieście, jedynie poruszając się wzdłuż ortogonalnych bloków miasta.
- W ogólniejszym ujęciu **normę ℓ_k** wektora $\boldsymbol{v}$ zawierającego n elementów definiujemy następująco: $\|\boldsymbol{v}\|_k = (|v_1|^k + |v_2|^k + \cdots + |v_n|^k)^{\frac{1}{k}}$. Norma ℓ_0 podaje liczbę niezerowych elementów w danym wektorze, natomiast ℓ_∞ wyznacza w nim maksymalną wartość bezwzględną.

Im wyższy indeks normy, tym bardziej skupia się ona na dużych wartościach i ignoruje małe wartości. Z tego właśnie powodu wskaźnik RMSE jest bardziej wyczulony na elementy odstające niż miara MAE. Jednakże jeśli liczba tych elementów wykładniczo maleje (np. w przypadku krzywej dzwonowej), pomiary RMSE znakomicie się sprawdzają i generalnie jest zalecane używanie tej miary.

Sprawdź założenia

Przed przystąpieniem do tworzenia modelu zawsze warto sporządzić listę dotychczasowych założeń i je zweryfikować (przez Ciebie lub kogoś innego) — może Ci to pomóc już na wczesnym etapie w wychwyceniu jakichś problemów. Na przykład prognozowane przez Twój system ceny w poszczególnych dystryktach będą przekazywane do następnej składowej potoku, a Ty zakładasz, że będą one tam używane w zdefiniowanym przez nas formacie. A jeśli okaże się, że ceny te będą kategoryzowane (np. „tanie", „średnie" i „drogie") i dopiero tego typu wartości będą wykorzystywane? W takim wypadku uzyskanie dokładnej ceny mija się zupełnie z celem; Twój system musi jedynie prawidłowo przewidywać kategorie. Skoro tak, zagadnienie,

którym się zajmujesz, powinno zostać oznaczone jako problem klasyfikacyjny, a nie regresyjny. Raczej nie chciałbyś tego odkryć dopiero po kilkumiesięcznej pracy nad systemem regresyjnym.

Na szczęście po rozmowie z szefostwem i pozostałymi zespołami masz pewność, że potrzebne będą rzeczywiste ceny, a nie kategorie. Znakomicie! Wszystko przygotowane, masz zielone światło i możesz się zabrać za pisanie kodu!

Zdobądź dane

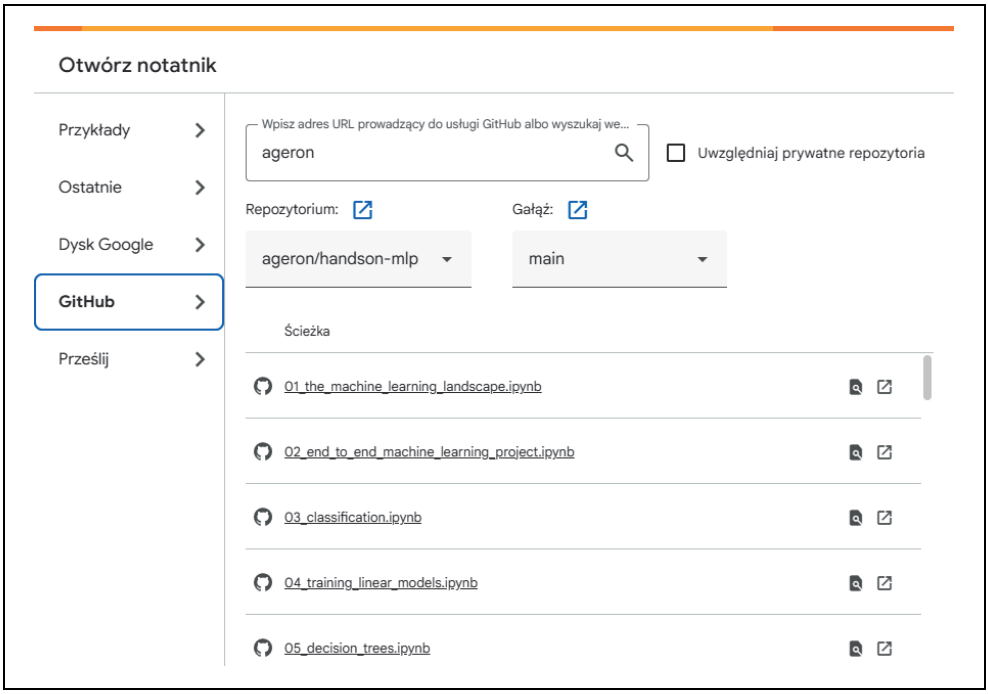
Czas pobrudzić sobie ręce. Bez wahania uruchom laptopa i w miarę zapoznawania się z treścią książki samodzielnie sprawdzaj przykładowy kod. Jak już wspomniałem w przedmowie, cały przykładowy kod w tej książce jest objęty licencją otwartego oprogramowania i dostępny w internecie (oryginał w języku angielskim pod adresem *https://github.com/ageron/handsonml3*, wersja polska dostępna na stronie *https://ftp.helion.pl/przyklady/umaszs.zip*) jako notatniki Jupyter, czyli interaktywne dokumenty zawierające tekst, obrazy i wykonywalne fragmenty kodu (w naszym przypadku napisane w języku Python). Zakładam w tej książce, że uruchamiasz te notatniki w bezpłatnym serwisie Google Colab, umożliwiającym uruchomienie każdego notatnika Jupyter bezpośrednio w internecie, bez konieczności instalowania czegokolwiek na własnym komputerze. Jeżeli chcesz skorzystać z innej platformy internetowej (np. Kaggle) lub zainstalować potrzebne narzędzia lokalnie na własnym komputerze, znajdziesz odpowiednie instrukcje we wspomnianych powyżej materiałach.

Uruchom przykładowy kod w serwisie Google Colab

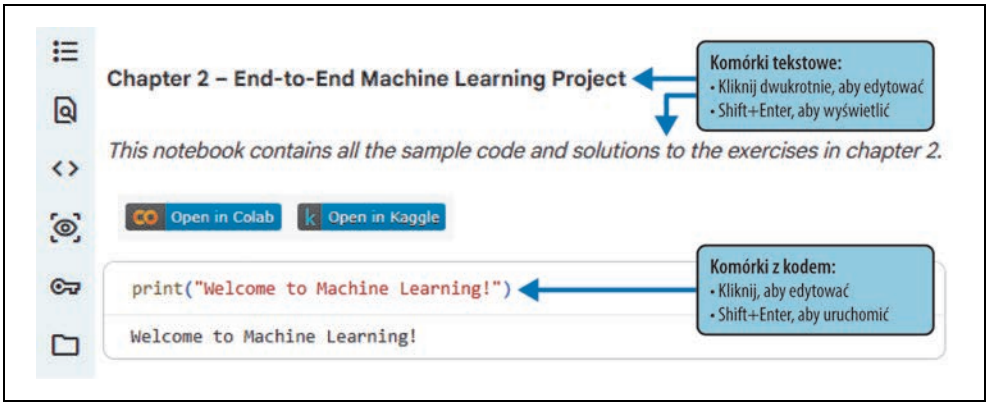
Otwórz najpierw przeglądarkę i odwiedź stronę *https://homl.info/colab-p*: przejdziesz do serwisu Google Colab, gdzie zostanie wyświetlona lista notatników Jupyter stworzonych na potrzeby tej książki (rysunek 2.3). Dla każdego rozdziału został przygotowany osobny notatnik, a do tego znajdziesz kilka dodatkowych notatników poświęconych bibliotekom *NumPy*, *Matplotlib*, *Pandas*, algebrze liniowej oraz rachunkowi różniczkowemu. Na przykład jeśli klikniesz nazwę *02_end_to_end_machine_learning_project.ipynb*, w serwisie Google Colab zostanie otwarty notatnik Jupyter powiązany z niniejszym rozdziałem (rysunek 2.4).

Notatnik Jupyter składa się z listy komórek. Każda komórka zawiera wykonywalny kod lub dane tekstowe. Spróbuj kliknąć dwukrotnie pierwszą komórkę tekstową (zawierającą zdanie *Welcome to Machine Learning!*). Wejdziesz w tryb edycji komórki. Zwróć uwagę, że w notatnikach Jupyter formatowanie tekstu jest realizowane za pomocą składni Markdown (np. ****pogrubienie****, **kursywa**, `# Tytuł`, *[url](tekst odnośnika)* itd.). Spróbuj zmodyfikować tekst, a następnie wciśnij kombinację klawiszy *Shift+Enter*, aby zobaczyć wynik.

Następnie stwórz nową komórkę tekstową, wybierając w menu *Wstaw/Komórka* z kodem. Ewentualnie możesz kliknąć przycisk + *Kod* w pasku narzędzi lub najechać kursorem myszy na spód komórki, dopóki nie pojawią się przyciski + *Kod* oraz + *Tekst*, po czym kliknij + *Kod*. W nowej komórce wpisz jakiś kod Pythona, na przykład `print("Witaj, świecie")`, po czym uruchom ten kod za pomocą kombinacji klawiszy *Shift+Enter* (lub kliknij przycisk ▷ po lewej stronie komórki).



Rysunek 2.3. Lista notatników w serwisie Google Colab



Rysunek 2.4. Twój notatnik w serwisie Google Colab

Jeśli nie jesteś zalogowany na swoim koncie Google, zostaniesz teraz o to poproszony (jeśli nie masz konta Google, będziesz musiał je utworzyć). Gdy już będziesz zalogowany, w momencie próby uruchomienia kodu zostaniesz ostrzeżony o tym, że firma Google nie jest autorem tego notatnika. Nikczemna osoba mogłaby stworzyć notatnik, za pomocą którego próbowałaby wyłudzić od Ciebie dane uwierzytelniające, aby uzyskać dostęp do Twoich danych poufnych, dlatego przed uruchomieniem kodu upewnij się, że autor jest wiarygodny (albo uważnie sprawdź każdą komórkę z kodem). Przy założeniu, że mi ufasz (lub planujesz przeanalizować każdą komórkę z kodem), możesz teraz kliknąć *Uruchom mimo to*.

Serwis Colab wyznaczy Ci następnie **środowisko wykonawcze** (ang. *runtime*): jest to bezpłatna wirtualna maszyna zlokalizowana na serwerach firmy Google, zawierająca zestaw narzędzi i bibliotek Pythona, w tym niemal wszystkich wymaganych do uruchomienia kodu z większości rozdziałów (w niektórych rozdziałach będziesz musiał uruchomić polecenie instalujące dodatkowe biblioteki). Zajmie to kilka sekund. Następnie serwis Colab automatycznie połączy się z tym środowiskiem wykonawczym i wykorzysta je do wykonania kodu zawartego w komórce. Co ważne, kod ten jest uruchamiany w środowisku wykonawczym, a nie na Twoim komputerze. Wynik kodu zostanie wyświetlony pod komórką. Gratulacje! Właśnie uruchomiłeś kod Pythona w serwisie Colab!

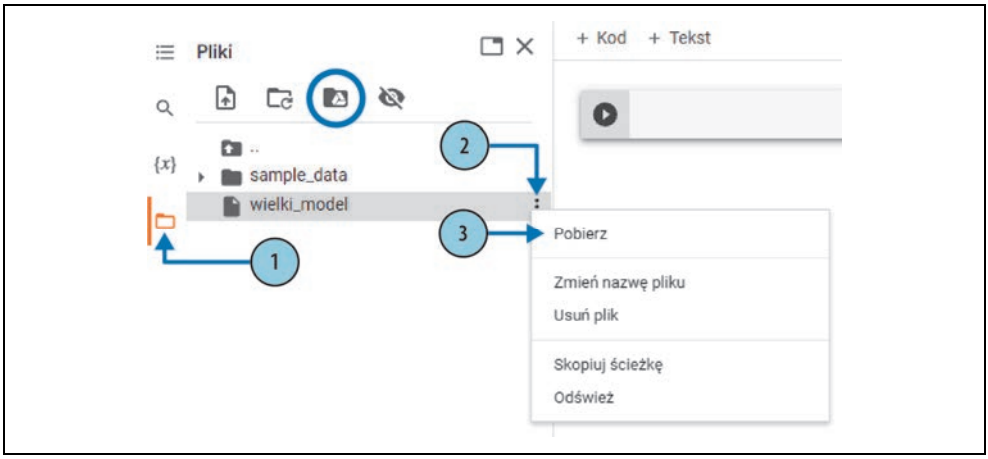
Aby wstawić nową komórkę, możesz również wcisnąć kombinację klawiszy *Ctrl+M* (lub *Cmd+M* w systemie macOS), a następnie klawisz *A* (aby wstawić komórkę powyżej aktywnej komórki) lub *B* (aby wstawić ją poniżej aktywnej komórki). Dostępnych jest wiele innych skrótów klawiaturowych: możesz je przeglądać i edytować po wciśnięciu kombinacji klawiszy *Ctrl+M* (lub *Cmd+M*), a następnie klawisza *H*. Jeżeli postanowisz otworzyć notatnik w serwisie Kaggle lub na własnym komputerze w aplikacji JupyterLab bądź innym interfejsie, takim jak Visual Studio Code z rozszerzeniem Jupyter, zauważysz pewne drobne różnice — środowiska wykonawcze są nazywane **jądrami** (ang. *kernel*), interfejs użytkownika i skróty klawiaturowe są nieco odmienne itd. — ale przełączanie się między środowiskami Jupyter nie jest zbyt skomplikowane.

Zapisz zmiany w kodzie i w danych

Możesz wprowadzać zmiany w notatniku uruchomionym w serwisie Colab i będą one przechowywane, dopóki będzie otwarta zakładka z tym notatnikiem. Jednak po zamknięciu tej zakładki wszystkie zmiany zostaną utracone. Aby tego uniknąć, zapisz kopię notatnika na Dysku Google (*Plik/Zapisz kopię na Dysku*). Ewentualnie możesz pobrać notatnik na swój komputer, wybierając *Plik/Pobierz/Pobierz plik IPYNB*. Możesz później odwiedzić adres *https://colab.research.google.com/* i otworzyć ponownie notatnik (zarówno przechowywany na Dysku Google, jak i na własnym dysku twardym).

Serwis Google Colab służy jedynie do interaktywnego użytkowania: możesz uruchamiać zawartość notatników i dowolnie modyfikować kod, ale nie możesz pozostawiać uruchomionych notatników przez długi czas bez nadzoru, gdyż środowisko wykonawcze zostanie zamknięte i utracisz wszelkie przechowywane w nim dane.

Jeżeli notatnik generuje dane, na których Ci zależy, pobierz je przed zakończeniem działania środowiska wykonawczego. W tym celu kliknij ikonę *Pliki* (zob. etap 1. na rysunku 2.5), znajdź plik, który chcesz pobrać, kliknij trzy ułożone pionowo kropki obok jego nazwy (etap 2.) i wybierz opcję *Pobierz* (etap 3.). Ewentualnie możesz zamontować swój Dysk Google w środowisku wykonawczym i umożliwić notatnikowi bezpośredni odczyt i zapis plików na tym Dysku, jak gdyby był katalogiem lokalnym. W tym celu kliknij ikonę *Pliki* (etap 1.), następnie ikonę Dysku Google (zaznaczona kółkiem na rysunku 2.5) i wykonuj pojawiające się na ekranie instrukcje.



Rysunek 2.5. Pobieranie pliku ze środowiska wykonawczego Google Colab (etapy od 1. do 3.) lub montowanie Dysku Google (zaznaczone kółkiem)

Domyślnie ścieżka zamontowanego Dysku Google to */content/drive/MyDrive*. Jeśli chcesz stworzyć kopię zapasową pliku z danymi, wystarczy skopiować go do tego katalogu za pomocą polecenia `!cp [.keeptogether]#/content/wielki_model /content/drive/MyDrive`. Każde polecenie rozpoczynające się wykrzyknikiem traktowane jest jako polecenie powłoki, a nie kod Pythona: polecenie `cp` jest poleceniem powłoki Linuksa kopiującym plik z jednej ścieżki do drugiej. Zwróć uwagę, że środowiska wykonawcze serwisu Colab są obsługiwane przez system Linux (a dokładniej dystrybucję Ubuntu).

Zalety i wady interaktywności

Notatniki Jupyter są interaktywne i jest to dla nas wspaniała wiadomość: możesz uruchamiać kolejno każdą komórkę, przerwać w dowolnym momencie, wstawić komórkę, zmodyfikować kod, cofnąć się i uruchomić ponownie wcześniejszą komórkę itd. Szczerze zachęcam do korzystania z tych możliwości. Jeżeli będziesz tylko uruchamiał komórki jedną po drugiej bez ingerowania w ich zawartość, nie będziesz w stanie uczyć się szybko. Jednak taka swoboda ma swoją cenę: bardzo łatwo uruchamiać komórki w niewłaściwej kolejności albo pominąć jakąś przypadkiem. W takiej sytuacji istnieje duże prawdopodobieństwo, że kod w późniejszych komórkach nie będzie działał. Na przykład pierwsza komórka w każdym z towarzyszących tej książce notatników zawiera kod konfiguracyjny (np. importowane biblioteki), dlatego musisz koniecznie ją uruchomić, gdyż w przeciwnym razie występujący po niej kod nie będzie działał.

Jeżeli kiedykolwiek natrafisz na „dziwny" błąd, spróbuj uruchomić ponownie środowisko wykonawcze (menu *Środowisko wykonawcze/Uruchom ponownie środowisko wykonawcze*), a następnie uruchom ponownie wszystkie komórki od samego początku notatnika. Operacja ta często rozwiązuje problem. Jeśli nie, to prawdopodobnie jedna z wprowadzonych przez Ciebie zmian „zepsuła" notatnik: wystarczy cofnąć te zmiany i notatnik powinien działać normalnie. Jeśli wciąż pojawia się jakiś problem, zgłoś opis problemu w serwisie GitHub.

Kod w książce a kod w notatnikach Jupyter

Od czasu do czasu możesz zauważyć drobne różnice między kodem zawartym w książce a umieszczonym w notatnikach. Może to wynikać z kilku powodów:

- Biblioteka mogła zostać nieznacznie zmieniona w momencie czytania treści książki, a może wbrew moim jak największym staraniom wkradł się gdzieś błąd. Niestety, nie mogę w magiczny sposób poprawić zawartości w Twoim egzemplarzu książki (chyba że czytasz jej wersję elektroniczną i pobrałeś jej najnowsze wydanie), ale *mogę* naprawić notatniki. Jeśli więc natrafisz na błąd po przepisaniu kodu z książki, sprawdź poprawiony kod w notatnikach: będę się starał usuwać z nich błędy na bieżąco i aktualizować o najnowsze wersje bibliotek.
- Notatniki zawierają dodatkowy kod upiększający rysunki (dodający etykiety, definiujący rozmiary czcionek itd.) oraz zapisujący je w wysokiej rozdzielczości na potrzeby tej książki. Jeśli chcesz, możesz spokojnie zignorować ten dodatkowy kod.

Zoptymalizowałem kod z myślą o prostocie i czytelności: starałem się, aby był możliwie liniowy i jednowymiarowy, przez co zdefiniowałem bardzo mało funkcji i klas. Celem było sprawienie, aby uruchamiany kod zawsze znajdował się na widoku, a nie był zagnieżdżony w kilku warstwach abstrakcji, przez które musiałbyś się przedzierać. Dzięki temu łatwiej jest również modyfikować ten kod. Dla zachowania prostoty występuje tu ograniczona obsługa błędów, a ja umieściłem niektóre z mniej powszechnych instrukcji importowanych klas tam, gdzie są potrzebne (a nie na początku pliku, jak jest zalecane przez poradnik stylu programowania PEP 8 Python). Wbrew pozorom Twój kod w środowisku produkcyjnym nie będzie się od tego drastycznie różnił: będzie jedynie nieco bardziej modułowy oraz będzie zawierał dodatkowe testy oraz rozbudowaną obsługę błędów.

W porządku! Skoro czujesz się już komfortowo w serwisie Colab, jesteś gotów do pobrania danych.

Pobierz dane

W typowym środowisku dane są najczęściej umieszczone w relacyjnej bazie danych lub innym powszechnie używanym magazynie danych i rozrzucone pomiędzy wiele tabel/dokumentów/plików. Aby uzyskać do nich dostęp, należy najpierw uzyskać autoryzację do ich przeglądania lub jakieś inne poświadczenia, a następnie zaznajomić się z używanym schematem danych[4]. W naszym projekcie jest jednak znacznie prościej: pobierzemy jedynie archiwum (*housing.tgz*) zawierające plik CSV (ang. *comma-separated values* — wartości rozdzielone przecinkami) o nazwie *housing.csv*, w którym mieszczą się wszystkie interesujące nas dane.

Zamiast samodzielnie pobierać i wypakowywać dane, zazwyczaj lepiej jest napisać zajmującą się tym funkcję. Jest to przydatne zwłaszcza w sytuacjach, gdy dane są regularnie aktualizowane: możesz stworzyć maleńki skrypt uruchamiany za każdym razem, gdy będziesz potrzebować najświeższych danych (ewentualnie możesz zaplanować wykonywanie tej czynności

[4] Musisz również brać pod uwagę ograniczenia prawne, np. prywatne pola, których nie należy nigdy kopiować do niezbyt bezpiecznych magazynów danych.

w regularnych odstępach czasu). Automatyzacja tego procesu ułatwia również instalację zbioru danych na wielu komputerach.

Funkcja pobierająca i wczytująca dane przedstawia się następująco:

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_housing_data():
    tarball_path = Path("datasets/housing.tgz")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "https://github.com/ageron/data/raw/main/housing.tgz"
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets", filter="data")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing_full = load_housing_data()
```

Jeśli pojawi się błąd SSL *CERTIFICATE_VERIFY_FAILED* w systemie macOS, prawdopodobnie musisz zainstalować pakiet *certifi*, zgodnie z instrukcjami dostępnymi na stronie *https://homl.info/sslerror*.

Po wywołaniu funkcji `fetch_housing_data()` jest wyszukiwany plik *datasets/housing.tgz*. Jeżeli nie zostanie znaleziony, zostanie utworzony katalog *datasets* wewnątrz bieżącego katalogu (domyślnie */content* w serwisie Colab). Pobierany jest plik *housing.tgz* z repozytorium GitHub *ageron/data* i zostaje wyodrębniona zawartość do katalogu *datasets*; zostanie utworzony katalog *datasets/housing* zawierający plik *housing.csv*. Na koniec funkcja ta wczytuje plik CSV do obiektu Pandas `DataFrame` przechowującego wszystkie dane, po czym go zwraca.

Jeśli korzystasz z Pythona 3.12 lub 3.13, powinieneś dodać argument `filter='data'` do metody `extractall()`. Ogranicza to możliwości algorytmu ekstrakcji i poprawia bezpieczeństwo (więcej szczegółów znajdziesz w dokumentacji).

Rzut oka na strukturę danych

Przyjrzyjmy się najpierw pierwszym pięciu wierszom naszych danych za pomocą metody `head()` obiektu `DataFrame` (rysunek 2.6).

Każdy wiersz reprezentuje jeden dystrykt. Dostępnych jest 10 atrybutów (nie wszystkie są widoczne na rysunku 2.6): `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value` i `ocean_proximity` (w języku polskim są to, odpowiednio: *Dł. geograficzna*, *Szer. geograficzna*, *Mediana wieku mieszkań*, *Całk. liczba pokoi*, *Całk. liczba sypialni*, *Populacja*, *Rodziny*, *Mediana dochodów*, *Mediana cen mieszkań* i *Odległość do oceanu*).

```
housing.head()
```

	longitude	latitude	housing_median_age	median_income	ocean_proximity	median_house_value
0	-122.23	37.88	41.0	8.3252	NEAR BAY	452600.0
1	-122.22	37.86	21.0	8.3014	NEAR BAY	358500.0
2	-122.24	37.85	52.0	7.2574	NEAR BAY	352100.0
3	-122.25	37.85	52.0	5.6431	NEAR BAY	341300.0
4	-122.25	37.85	52.0	3.8462	NEAR BAY	342200.0

Rysunek 2.6. Pięć pierwszych wierszy w naszym zbiorze danych

Dzięki metodzie `info()` możemy się zapoznać z krótkim opisem danych, zwłaszcza z całkowitą liczbą wierszy, typem każdego atrybutu oraz liczbą wartości niezerowych:

```
>>> housing_full.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):


#       Column                Non-Null Count     Dtype
---     ------                --------------     -----
0       longitude             20640 non-null     float64
1       latitude              20640 non-null     float64
2       housing_median_age    20640 non-null     float64
3       total_rooms           20640 non-null     float64
4       total_bedrooms        20433 non-null     float64
5       population            20640 non-null     float64
6       households            20640 non-null     float64
7       median_income         20640 non-null     float64
8       median_house_value    20640 non-null     float64
9       ocean_proximity       20640 non-null     object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Gdy w tej książce przykład zawiera kombinację kodu i wyników (tak jak powyżej), jest formatowany jak w interpreterze Pythona, co pozwala zachować większą czytelność: wiersze kodu są oznaczone symbolami zachęty >>> (lub ... w przypadku bloków kodu objętych wcięciem), natomiast wyniki nie mają żadnego prefiksu.

Na zbiór danych składa się 20 640 przykładów, co oznacza, że jest on dość mały jak na standardy uczenia maszynowego, ale nadaje się idealnie dla początkujących analityków. Zwróć uwagę, że atrybut `total_bedrooms` zawiera zaledwie 20 433 wartości niezerowe, co oznacza, że cecha ta nie została zdefiniowana dla 207 dystryktów. Będziesz się musiał zająć tym później.

Z wyjątkiem `ocean_proximity` wszystkie pozostałe atrybuty mają wartości numeryczne. Typem wartości w cesze `ocean_proximity` jest `object`, co oznacza, że można tu przechowywać dowolny obiekt języka Python. Skoro jednak wczytaliśmy dane z pliku CSV, to wiemy, że musi to być wartość tekstowa. Po przyjrzeniu się pierwszym pięciu wierszom zbioru danych okazuje się,

że wartości w kolumnie `ocean_proximity` są powtarzalne, dzięki czemu możemy wywnioskować, że mamy najprawdopodobniej do czynienia z atrybutem kategorialnym. Możemy sprawdzić, jakie kategorie są dostępne oraz jaki jest rozkład poszczególnych dystryktów do każdej z nich; posłużymy się w tym celu metodą `value_counts()`:

```
>>> housing_full["ocean_proximity"].value_counts()
ocean_proximity
<1H OCEAN 9136
INLAND 6551
NEAR OCEAN 2658
NEAR BAY 2290
ISLAND 5
Name: count, dtype: int64
```

Przyjrzyjmy się innym polom. Metoda `describe()` generuje podsumowanie atrybutów numerycznych (rysunek 2.7).

`housing.describe()`

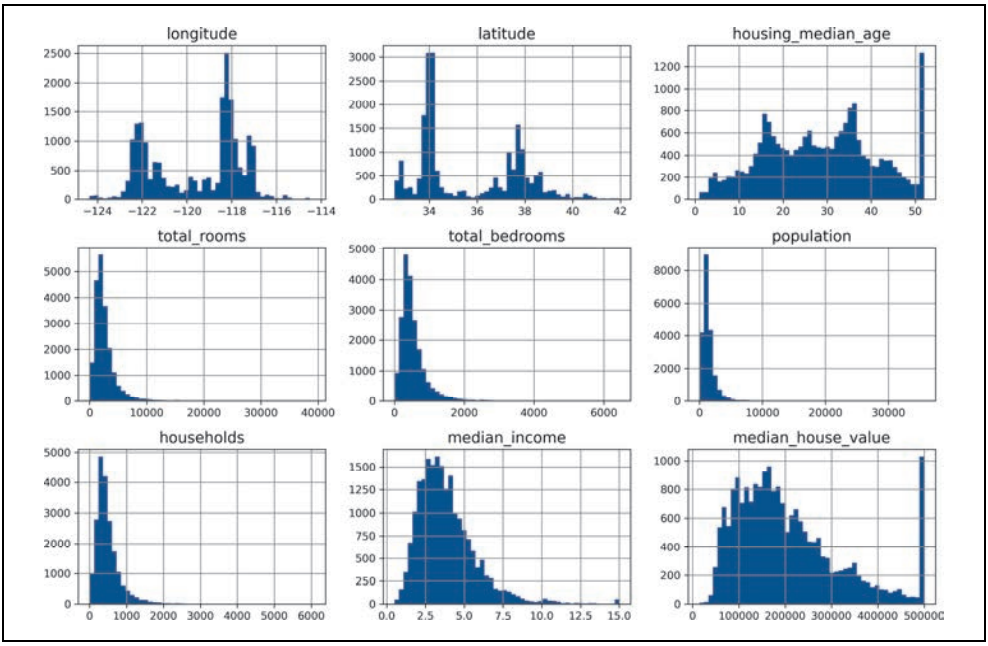
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	500001.000000

Rysunek 2.7. Podsumowanie wszystkich atrybutów numerycznych

Wartości `count` (liczba), `mean` (średnia), `min` i `max` raczej nie wymagają wyjaśnienia. Zwróć uwagę, że wartości zerowe są ignorowane (z tego powodu liczba elementów `total_bedrooms` wynosi tu 20 433, a nie 20 640). Wiersz `std` zawiera **odchylenie standardowe** (ang. *standard deviation*), określające „rozrzut" wartości[5]. Wartości 25%, 50% i 75% ukazują odpowiadające im **percentyle**: percentyl wskazuje wartość, poniżej której znajduje się określony odsetek obserwowanych przykładów. Przykładowo 25% dystryktów ma wartość atrybutu `housing_median_age` mniejszą od 18, z kolei 50% próbek nie przekracza wartości 29, a 75% procent nie osiągnęło wieku 37 lat. Parametry te są często nazywane, kolejno: 25. percentylem (lub pierwszym **kwartylem**), medianą oraz 75. percentylem (lub trzecim kwartylem).

[5] Odchylenie standardowe jest zwyczajowo oznaczane grecką literą σ (sigma) i w ujęciu matematycznym stanowi pierwiastek kwadratowy z **wariancji**, którą z kolei definiujemy jako średnią arytmetyczną kwadratów odchyleń od wartości średnich przykładów. Gdy wykres danej cechy przyjmuje dzwonowy kształt **rozkładu normalnego** (zwanego również **rozkładem Gaussa**), co jest bardzo często spotykane, znajduje zastosowanie tzw. **reguła 68-95 – 99,7**: mniej więcej 68% próbek znajduje się w odległości 1σ od wartości oczekiwanej, 95% w odległości 2σ, a 99,7% — w odległości 3σ.

Kolejnym szybkim sposobem przyjrzenia się analizowanym danym jest wygenerowanie histogramu dla każdego atrybutu numerycznego. Histogram przedstawia liczbę przykładów (w pionowej osi) znajdujących się w określonym przedziale wartości (oś pozioma). Możesz stworzyć histogram jednego atrybutu lub za pomocą metody `hist()` dokonać tego dla całego zbioru danych (co ukazano za pomocą poniższego listingu), co spowoduje narysowanie histogramu dla każdego atrybutu numerycznego (rysunek 2.8).



Rysunek 2.8. Histogram każdego atrybutu numerycznego

Liczbę przedziałów wartości można dostosować za pomocą argumentu `bins` (warto poeksperymentować z tą wartością, aby zobaczyć, jak wpływa na histogramy):

```
import matplotlib.pyplot as plt

housing_full.hist(bins=50, figsize=(12, 8))
plt.show()
```

Możesz wyczytać z tych histogramów pewne informacje:

- Przede wszystkim atrybut mediany dochodów nie przypomina danych podawanych w dolarach amerykańskich (USD). Po konsultacji z zespołem odpowiedzialnym za zebranie tych danych wiesz już, że dane te zostały przeskalowane i ograniczone do maksymalnej wartości 15 (w rzeczywistości 15,0001) dla wyższej mediany dochodów oraz do minimalnej wartości 0,5 (w rzeczywistości 0,4999) dla niższej mediany dochodów. Liczby reprezentują w przybliżeniu dziesiątki tysięcy dolarów (np. wartość 3 oznacza w rzeczywistości ok. 30 000 dolarów). Praca ze wstępnie przetworzonymi danymi stanowi normę w uczeniu maszynowym i niekoniecznie musi stanowić problem, należy jednak spróbować zrozumieć, jakim operacjom zostały one poddane.

- Ograniczeniu uległy również wartości median wieku oraz cen mieszkań. Ta druga informacja może nas szczególnie zmartwić, ponieważ stanowi ona nasz docelowy atrybut (etykietę). Algorytmy uczenia maszynowego mogą uznać, że ceny domów nigdy nie przekraczają górnej, ograniczonej wartości. Musisz się skontaktować z zespołem klienckim (zespołem wykorzystującym uzyskane przez Ciebie wyniki) i dowiedzieć się, czy ograniczenie to będzie stanowiło problem. Jeżeli powiedzą Ci, że potrzebne są im precyzyjne prognozy nawet powyżej wartości 500 000 dolarów, to pozostają Ci dwie możliwości:
  - uzyskać prawidłowe etykiety dla dystryktów mających obcięte górne wartości cen;
  - usunąć te dystrykty z zestawu uczącego (a także testowego, ponieważ system nie powinien być karany, jeżeli będzie przewidywał wartości przekraczające 500 000 dolarów).
- Każdy z tych atrybutów jest ukazany w odmiennych skalach, nieraz znacznie zróżnicowanych. Zajmiemy się tym zagadnieniem w dalszej części rozdziału, podczas omawiania skalowania cech.
- Wiele histogramów cechuje się **prawoskośnością** (ang. *skewed right*): rozciągają się one znacznie bardziej po prawej stronie mediany niż po lewej. Utrudnia to nieco niektórym algorytmom uczenia maszynowego rozpoznawanie wzorców. Spróbujesz później przekształcić te atrybuty w taki sposób, aby ich rozkład był bardziej symetryczny i dzwonowy.

Teraz już powinieneś wiedzieć co nieco na temat danych, którymi będziesz się zajmować.

Stwórz zbiór testowy

Zanim znowu zajrzysz do danych, musisz stworzyć z nich podzbiór testowy, odłożyć go i nigdy więcej do niego nie zerkać. Być może dziwnym pomysłem wydaje się dobrowolne odłożenie części danych na tym etapie. Przecież ledwie zajrzeliśmy do nich, aby poznać ich strukturę, i z pewnością moglibyśmy wyciągnąć na ich temat znacznie więcej wniosków przed wybraniem jakiegoś algorytmu, prawda? Owszem, z tym że Twój mózg zawiera niesamowity układ rozpoznawania wzorców, co oznacza także, że jest znacznie narażony na przetrenowanie: jeśli przyjrzysz się zbiorowi testowemu, możesz dostrzec jakiś pozornie interesujący wzorzec, który sprawi, że wybierzesz określony model uczenia maszynowego. Po oszacowaniu błędu uogólniania za pomocą tego zbioru uzyskane wyniki okażą się nadmiernie optymistyczne i uruchomisz system, którego wydajność będzie mniejsza od zakładanej. Zjawisko to jest nazywane **obciążeniem związanym z podglądaniem danych** (ang. *data snooping bias*).

Teoretycznie tworzenie zbioru testowego nie powinno być zbyt kłopotliwe: wystarczy wybrać losowo część przykładów (najczęściej 20% całego zbioru lub mniej, jeżeli zestaw danych jest bardzo duży) i je odłożyć.

```
import numpy as np

def shuffle_and_split_data(data, test_ratio, rng):
    shuffled_indices = rng.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Teraz możesz użyć powyższej funkcji w następujący sposób:

```
>>> rng = np.random.default_rng()  # domyślny generator liczb losowych
>>> train_set, test_set = shuffle_and_split_data(housing_full, 0.2, rng)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

To rozwiązanie działa, ale nie jest idealne: przy następnym uruchomieniu programu zostanie wygenerowany zupełnie inny zbiór testowy! Po pewnym czasie Ty (albo algorytm uczenia maszynowego) zobaczysz cały zbiór danych uczących, a tego właśnie chcemy uniknąć.

Jednym z rozwiązań jest zapisanie zestawu testowego po jego pierwszym utworzeniu i wczytywanie go przy kolejnych okazjach. Możesz również skorzystać z zarodka liczb losowych (na przykład przez przekazanie `seed=42` do funkcji `default_rng()`)[6], aby zawsze generował tę samą sekwencję liczb losowych przy każdym uruchomieniu programu.

Obydwa te rozwiązania jednak zawiodą po dostarczeniu nowej porcji zaktualizowanych danych. Aby dysponować stabilnym podziałem na przykłady uczące/testowe nawet po zaktualizowaniu zestawu danych, możemy dobierać przykłady stanowiące część zbioru testowego za pomocą ich identyfikatorów (przy założeniu, że próbki te zawierają niepowtarzalne i niezmienne identyfikatory). Możesz na przykład obliczyć hasz każdego identyfikatora i umieścić ten przykład w zestawie danych testowych, jeżeli wartość hasza jest mniejsza lub równa 20% maksymalnej wartości hasza. Dzięki temu jesteśmy w stanie zagwarantować stabilność zestawu testowego za każdym razem, gdy uruchamiamy program, nawet po odświeżeniu zbioru danych. Nowy zbiór testowy będzie zawierał 20% świeżych przykładów, ale nie pojawi się w nim żaden przykład występujący wcześniej w zbiorze uczącym.

Oto jedna z możliwych implementacji tego mechanizmu:

```
from zlib import crc32

def is_id_in_test_set(identifier, test_ratio):
    return crc32(np.int64(identifier)) < test_ratio * 2**32

def split_data_with_id_hash(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: is_id_in_test_set(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Niestety zbiór danych *Housing* nie zawiera kolumny z identyfikatorami. Najprostszym rozwiązaniem okazuje się wykorzystanie w tym celu indeksu wiersza.

```
housing_with_id = housing_full.reset_index() # Dodaje kolumnę `index`
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "index")
```

Jeśli używasz indeksu wiersza jako niepowtarzalnego identyfikatora, musisz się upewnić, że nowe informacje będą umieszczane na końcu zestawu danych i że żaden wiersz nie zostanie

[6] Zauważysz, że ludzie często korzystają z zarodka o wartości 42. Liczba ta nie jest w żaden sposób wyjątkowa, stanowi jedynie odpowiedź na życie, wszechświat i całą resztę.

nigdy usunięty. Jeśli nie jesteś w stanie tego zagwarantować, możesz spróbować wykorzystać najstabilniejsze cechy do stworzenia niepowtarzalnego identyfikatora. Przykładowo współrzędne geograficzne dystryktów raczej nie zmienią się przez kilka najbliższych milionów lat, dlatego możesz wygenerować z nich identyfikatory w zaprezentowany poniżej sposób[7]:

```
housing_with_id["id"] = housing_full["longitude"] * 1000 + housing_full["latitude"]
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "id")
```

Moduł *Scikit-Learn* zawiera kilka różnych funkcji rozdzielających zbiory danych na wiele podzbiorów. Najprostszą z nich jest `train_test_split()`, której działanie w znacznym stopniu przypomina zdefiniowaną przed chwilą funkcję `shuffle_and_split_data()`, zawiera jednak ona pewne dodatkowe elementy. Po pierwsze, mamy tu do czynienia z parametrem `random_state`, pozwalającym wybrać zarodek liczb losowych. Po drugie natomiast, możemy przekazywać wiele zbiorów danych mających taką samą liczbę wierszy — będą one rozdzielane pomiędzy takie same indeksy (jest to bardzo przydatne rozwiązanie, w przypadku gdy np. etykiety znajdują się w osobnym obiekcie `DataFrame`):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing_full, test_size=0.2, random_state=42)
```

Do tej pory rozważaliśmy metody całkowicie losowego próbkowania. Zazwyczaj takie rozwiązanie spisuje się dobrze w przypadku odpowiednio dużych zbiorów danych (zwłaszcza w odniesieniu do liczby atrybutów), bowiem w przeciwnym razie ryzykujemy wprowadzenie dużego obciążenia próbkowania. Gdy pracownicy urzędu statystycznego zamierzają przeprowadzić ankietę na tysiącu osób, nie są one wybierane losowo z książki telefonicznej. Muszą zostać tak dobrane, aby stanowiły reprezentatywny przykład całej populacji w kontekście zadawanych pytań. Na przykład według amerykańskiego Urzędu Spisu Ludności 51,6% obywateli w wieku uprawniającym do głosowania to kobiety, a 48,4% to mężczyźni, zatem prawidłowo przeprowadzona ankieta powinna uwzględniać te proporcje: 516 kobiet i 484 mężczyzn (przynajmniej wtedy, gdy odpowiedzi między obydwiema płciami mogą się w jakiś sposób różnić). Jest to tzw. **losowanie warstwowe** (ang. *stratified sampling*): populacja zostaje rozdzielona na jednorodne podgrupy zwane **warstwami** (ang. *strata*, l. poj. *stratum*) i z każdej z nich jest dobierana odpowiednia liczba przykładów zapewniająca prawidłowe odzwierciedlenie stanu populacji. Gdyby ankieterzy używali metod czysto losowego doboru przykładów, istniałoby ponad dziesięcioprocentowe prawdopodobieństwo uzyskania wypaczonego zestawu testowego zawierającego mniej niż 49% lub ponad 54% kobiet. W każdym przypadku wyniki ankiety cechowałyby się prawdopodobnie dość dużym błędem systematycznym próbkowania.

Załóżmy, że po rozmowie z pewnymi ekspertami okazuje się, że bardzo ważnym atrybutem pomagającym w prognozowaniu mediany cen mieszkań jest mediana dochodów. Możesz chcieć zagwarantować, żeby zbiór testowy wiernie reprezentował różne kategorie dochodów dla całego zbioru danych. Mediana dochodów stanowi atrybut ciągłych wartości numerycznych, dlatego najpierw należy stworzyć atrybuty kategorii dochodów. Przyjrzyjmy się uważniej

[7] W rzeczywistości podane współrzędne geograficzne nie są zbyt precyzyjne i w związku z tym wiele dystryktów otrzyma taką samą wartość identyfikatora oraz znajdzie się w tym samym zbiorze (uczącym bądź testowym). Ku naszemu utrapieniu wkrada się w ten sposób pewien błąd systematyczny próbkowania.

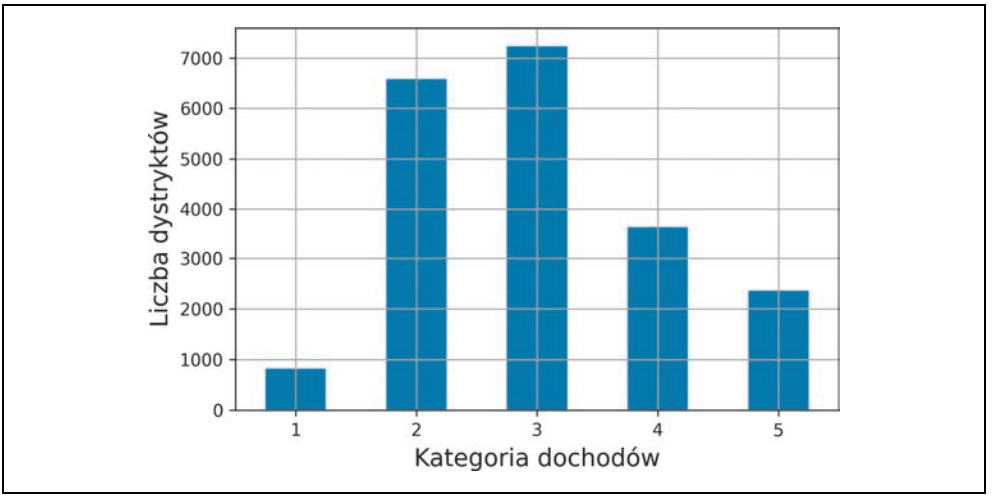
histogramowi mediany dochodów na rysunku 2.8: większość wartości mieści się w przedziale od 1,5 do 6 (tzn. 15 000 – 60 000 dolarów), jednak niektóre znacznie przekraczają 6. Ważne jest, aby każda warstwa zestawu danych zawierała wystarczającą liczbę przykładów, gdyż w przeciwnym wypadku oszacowanie znaczenia danej warstwy może być nieadekwatne do rzeczywistości. Oznacza to, że nie możemy tworzyć zbyt wielu warstw i każda z nich powinna być wystarczająco duża. Poniższy kod wykorzystuje funkcję `pd.cut()` do utworzenia atrybutu kategorii dochodów składającego się z pięciu kategorii (oznaczonych cyframi od 1 do 5); zakres pierwszej kategorii wynosi od 0 do 1,5 (tzn. poniżej 15 000 dolarów), drugiej kategorii od 1,5 do 3 itd.:

```
housing_full["income_cat"] = pd.cut(housing_full["median_income"],
                                    bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                                    labels=[1, 2, 3, 4, 5])
```

Uzyskane kategorie zostały pokazane na rysunku 2.9:

```
cat_counts = housing_full["income_cat"].value_counts().sort_index()
cat_counts.plot.bar(rot=0, grid=True)
plt.xlabel("Kategoria dochodów")
plt.ylabel("Liczba dystryktów")
plt.show()
```

Teraz możemy przeprowadzić próbkowanie warstwowe na podstawie kategorii dochodów. Biblioteka *Scikit-Learn* zawiera wiele klas rozdzielających w pakiecie *sklearn.model_selection*, implementujących odmienne strategie rozdzielania zestawu danych na zbiory uczący i testowy. Każda z tych klas zawiera metodę `split()` zwracającą iterator różnych podziałów tych samych danych.



Rysunek 2.9. Histogram kategorii dochodów

Mówiąc precyzyjniej, metoda `split()` generuje *indeksy* zbiorów uczącego i testowego, a nie same dane. Dysponowanie wieloma podziałami bywa przydatne, jeśli chcesz lepiej oszacować skuteczność modelu, o czym się przekonasz w czasie omawiania sprawdzianu krzyżowego

w dalszej części rozdziału. Na przykład poniższy kod generuje dziesięć różnych podziałów warstwowych tego samego zestawu danych:

```
from sklearn.model_selection import StratifiedShuffleSplit

splitter = StratifiedShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
strat_splits = []
for train_index, test_index in splitter.split(housing_full, housing_full["income_cat"]):
    strat_train_set_n = housing_full.iloc[train_index]
    strat_test_set_n = housing_full.iloc[test_index]
    strat_splits.append([strat_train_set_n, strat_test_set_n])
```

Na razie wystarczy na tylko pierwszy podział:

```
strat_train_set, strat_test_set = strat_splits[0]
```

Albo, skoro próbkowanie warstwowe jest tak powszechne, można łatwiej dokonać jednokrotnego podziału za pomocą funkcji `train_test_split()` z argumentem `stratify`:

```
strat_train_set, strat_test_set = train_test_split(
    housing_full, test_size=0.2, stratify=housing_full["income_cat"], random_state=42)
```

Sprawdźmy, czy mechanizm ten działa zgodnie z naszymi założeniami. Najpierw możemy zobaczyć proporcje kategorii dochodów w zestawie testowym:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
income_cat
3     0.350533
2     0.318798
4     0.176357
5     0.114341
1     0.039971
Name: count, dtype: float64
```

Za pomocą podobnego kodu możemy mierzyć proporcje kategorii przychodów w pełnym zestawie danych. Na rysunku 2.10 porównujemy te proporcje w całym zbiorze danych, w zbiorze testowym wygenerowanym za pomocą losowania warstwowego oraz w zestawie testowym utworzonym w całkowicie losowy sposób. Jak widać, zbiór testowy wygenerowany przy użyciu losowania warstwowego ma proporcje niemal identyczne jak uzyskane w pełnym zbiorze danych, podczas gdy wyniki uzyskane w całkowicie losowym zestawie danych wskazują wypaczenie proporcji.

Kategoria dochodów	Łącznie (%)	Warstwowe (%)	Losowe (%)	Błąd - warstwowe (%)	Błąd - losowe (%)
1	3.98	4.00	4.24	0.36	6.45
2	31.88	31.88	30.74	-0.02	-3.59
3	35.06	35.05	34.52	-0.01	-1.53
4	17.63	17.64	18.41	0.03	4.42
5	11.44	11.43	12.09	-0.08	5.63

Rysunek 2.10. Porównanie obciążenia próbkowania dla losowania warstwowego i losowego próbkowania

Nie będziesz już korzystał z kolumny `income_cat`, więc równie dobrze możesz ją usunąć, dzięki czemu dane zostaną przywrócone do pierwotnego stanu:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

Nie bez powodu poświęciliśmy tyle czasu na zagadnienie tworzenia zbioru testowego: jest to często bagatelizowany, ale jeden z najważniejszych elementów uczenia maszynowego. Do tego wiele z poruszonych tu koncepcji przyda nam się w dalszej części książki, podczas omawiania sprawdzianu krzyżowego. Przejdźmy teraz do następnego etapu: eksplorowania danych.

Odkrywaj i wizualizuj dane, aby zdobywać nowe informacje

Na razie zerknęliśmy jedynie na dane w celu ustalenia rodzaju informacji, z jakimi będziemy pracować. Teraz naszym celem jest zagłębienie się w te dane.

Upewnijmy się najpierw, że odłożyliśmy zbiór testowy i że zajmujemy się wyłącznie zestawem uczącym. Poza tym jeśli zbiór uczący ma bardzo duże rozmiary, warto wydzielić z niego zbiór eksploracyjny, aby przyspieszyć i ułatwić sobie manipulowanie danymi w fazie eksploracji. W tym przypadku zestaw uczący jest na tyle mały, że możemy korzystać ze wszystkich danych. Będziesz eksperymentował z różnymi przekształceniami pełnego zbioru uczącego, dlatego powinieneś stworzyć kopię oryginału, aby wrócić do niego po zakończeniu pracy:

```
housing = strat_train_set.copy()
```

Zwizualizuj dane geograficzne

Skoro zestaw danych zawiera dane geograficzne (długość i szerokość geograficzną), warto stworzyć wykres punktowy wszystkich dystryktów, aby zobaczyć ich rozmieszczenie w układzie współrzędnych (rysunek 2.11):

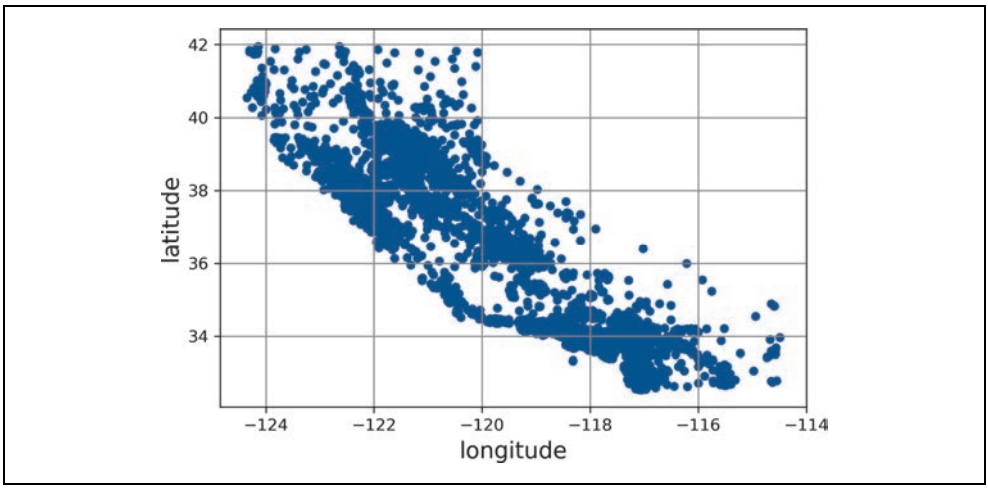
```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True)
plt.show()
```

Kształt tego wykresu rzeczywiście przypomina Kalifornię, jednak poza tym trudno doszukać się jakiegokolwiek wzoru. Po wyznaczeniu wartości `0,2` parametru `alpha` będzie nam o wiele łatwiej zwizualizować miejsca, w których występuje duże zagęszczenie punktów danych (rysunek 2.12):

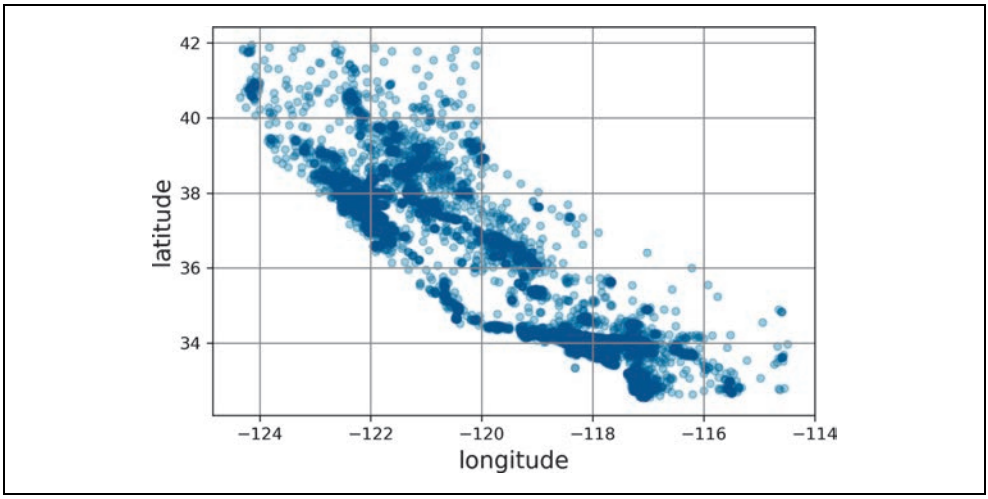
```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)
plt.show()
```

Teraz jest znacznie lepiej: wyraźnie widać obszary o dużym zagęszczeniu dystryktów, mianowicie rejon Bay Area oraz okolice miast Los Angeles i San Diego, a także długi pas gęsto zaludnionych obszarów w Dolinie Kalifornijskiej (zwłaszcza wokół miast Sacramento i Fresno).

Nasze mózgi doskonale radzą sobie z dostrzeganiem wzorów na danych przedstawionych w postaci graficznej, ale czasami trzeba sobie pomagać, modyfikując niektóre parametry wizualizacji.



Rysunek 2.11. Geograficzny wykres punktowy danych



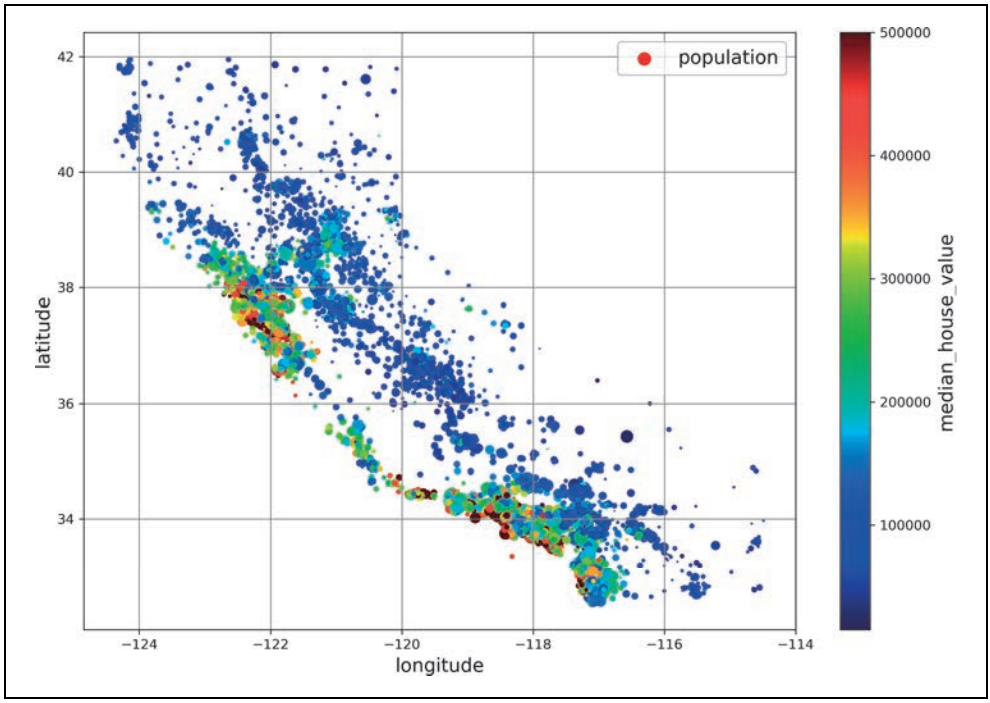
Rysunek 2.12. Lepsza wizualizacja, ukazująca obszary o dużym zagęszczeniu punktów danych

Przyjrzyj się następnie sytuacji z cenami mieszkań (rysunek 2.13). Promień każdego widocznego kółka symbolizuje populację dystryktu (opcja s), z kolei kolorami oznaczamy ceny mieszkań (opcja c). Tutaj skorzystasz z domyślnej mapy kolorów (opcja `cmap`) o nazwie `jet`, której zakres mieści się od niebieskiego (małe ceny) do czerwonego (duże ceny)[8]:

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,
             s=housing["population"] / 100, label="population",
             c="median_house_value", cmap="jet", colorbar=True,
             legend=True, sharex=False, figsize=(10, 7))
plt.show()
```

[8] Jeżeli czytasz tę książkę w wersji czarno-białej, zaznacz czerwonym pisakiem większość wybrzeża od regionu Bay Area aż do San Diego. Możesz również na żółto otoczyć okolice Sacramento.

Możemy z tego wykresu wyczytać, że ceny mieszkań są w dużej mierze uzależnione od położenia geograficznego (np. od odległości do oceanu) i zaludnienia, o czym już zapewne wiesz. Algorytm analizy skupień powinien pomóc w wykryciu największego skupiska, a także dodać nowe cechy, pozwalające określić odległość do centrów tego skupienia. Również przydatna może się okazać odległość do oceanu, chociaż w północnej Kalifornii w rejonie wybrzeża ceny domów nie są zbyt wysokie, zatem ta reguła nie jest taka prosta.



Rysunek 2.13. Ceny mieszkań w Kalifornii: na czerwono zostały zaznaczone wysokie ceny mieszkań, na niebiesko niskie ceny, a duże kółka symbolizują obszary o dużym zaludnieniu

Poszukaj korelacji

Nasz zbiór danych nie jest zbyt duży, dlatego możemy z łatwością wyliczyć **współczynnik korelacji liniowej** (zwany również **współczynnikiem korelacji Pearsona**) pomiędzy każdą parą wartości numerycznych za pomocą metody `corr()`:

```
corr_matrix = housing.corr(numeric_only=True)
```

Teraz możesz sprawdzić stopień korelacji każdego atrybutu z medianą cen mieszkań:

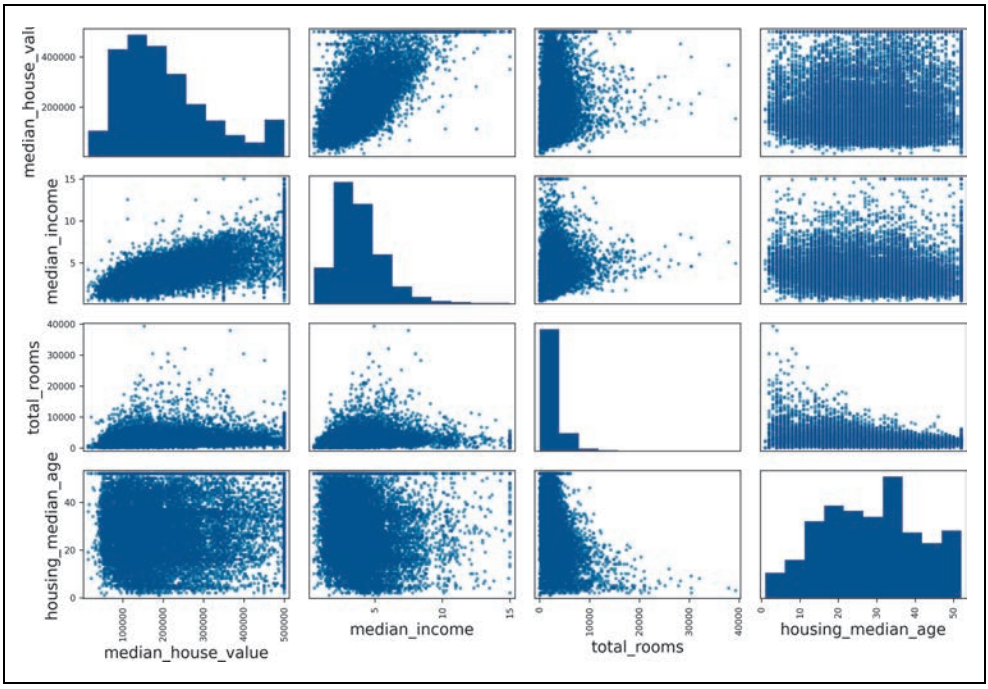
```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value     1.000000
median_income          0.688380
total_rooms            0.137455
housing_median_age     0.102175
households             0.071426
total_bedrooms         0.054635
```

```
population                  0.020153
longitude                  -0.050859
latitude                   -0.139584
Name: median_house_value, dtype: float64
```

Wartości współczynnika korelacji mieszczą się w zakresie pomiędzy –1 a 1. Wartości zbliżone do 1 wskazują silną korelację dodatnią; na przykład mediana cen mieszkań zazwyczaj rośnie wraz ze wzrostem mediany dochodów. Z kolei wartości zbliżone do –1 mówią nam, że istnieje silna korelacja ujemna; widzimy niewielką korelację ujemną pomiędzy szerokością geograficzną a medianą cen mieszkań.

Innym sposobem sprawdzenia korelacji pomiędzy atrybutami jest użycie funkcji `scatter_matrix` stanowiącej część modułu *Pandas*; generuje ona wykres każdego atrybutu numerycznego wobec pozostałych atrybutów numerycznych. Obecnie mamy do dyspozycji 9 atrybutów numerycznych, dlatego uzyskalibyśmy w sumie $9^2 = 81$ wykresów, które łącznie nie zmieściłyby się na jednej stronie, dlatego wybierasz kilka najbardziej obiecujących atrybutów, które wydają się skorelowane w największym stopniu z medianą cen mieszkań (rysunek 2.14):

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
plt.show()
```

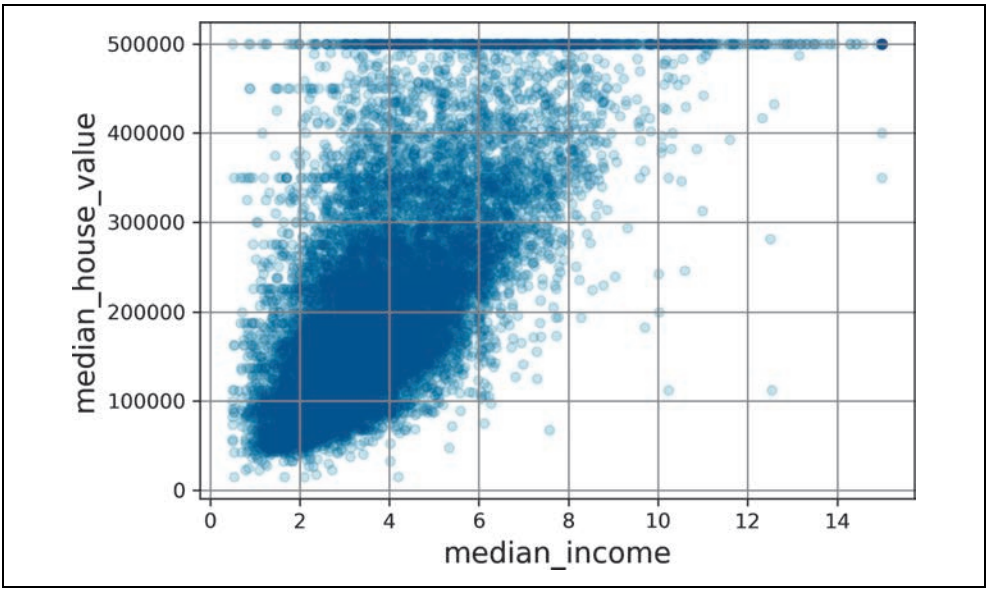


Rysunek 2.14. Zaprezentowany wykres rozproszenia pokazuje każdy atrybut numeryczny w stosunku do pozostałych atrybutów numerycznych, a także histogram wartości poszczególnych atrybutów na głównej przekątnej (biegnąca od lewego górnego do prawego dolnego rogu)

Główna przekątna byłaby wypełniona prostymi, gdyby moduł *Pandas* tworzył wykresy każdej zmiennej wobec samej siebie, co nie byłoby zbyt przydatne. Dlatego zamiast tego widzimy histogram każdego atrybutu (są dostępne również inne opcje; są one dobrze opisane w dokumentacji modułu *Pandas*).

Jeśli spojrzymy na wykresy korelacji, to można uznać, że najbardziej obiecującym atrybutem służącym do prognozowania mediany cen mieszkań jest mediana dochodów, dlatego przyglądasz się uważniej wykresowi korelacji (rysunek 2.15):

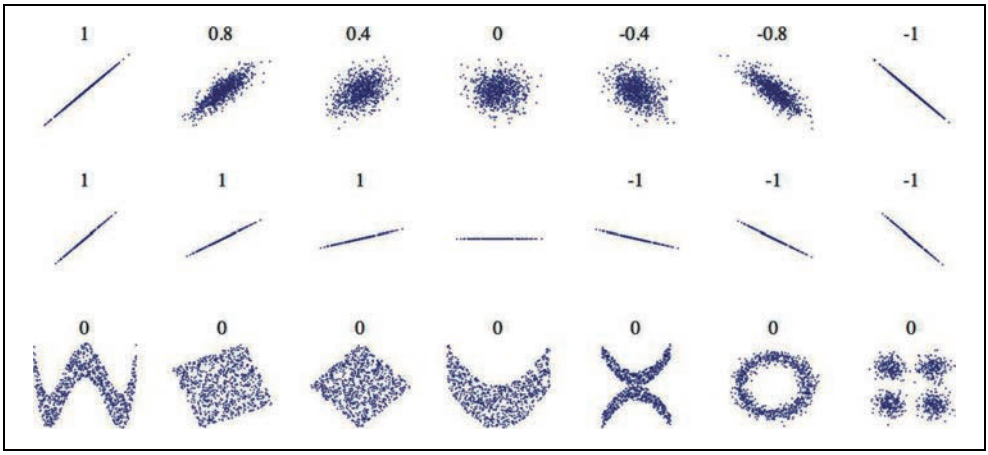
```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
                 alpha=0.1, grid=True)
plt.show()
```



Rysunek 2.15. Wykres mediany cen mieszkań w funkcji mediany dochodów

Z tego wykresu dowiadujemy się kilku rzeczy. Po pierwsze, korelacja pomiędzy tymi atrybutami jest całkiem silna; możemy bez trudu dostrzec tendencję wzrostową, chociaż dane są zaszumione. Po drugie, wspomniane już ograniczenie ceny jest wyraźnie widoczne jako pozioma linia w punkcie 500 000 dolarów. Możemy jednak zauważyć mniej oczywiste proste: poziomą w punkcie 450 000 dolarów, kolejną w okolicy 350 000 dolarów, być może jeszcze jedną mniej więcej w punkcie 280 000 dolarów, a pod nią kilka innych. Warto byłoby spróbować usunąć z danych odpowiedzialne za to dystrykty, aby uniemożliwić algorytmom naukę odtwarzania takich dziwactw.

Zwróć uwagę, że współczynnik korelacji mierzy jedynie zależność liniową („jeśli wartość x rośnie, to wartość y zazwyczaj rośnie/spada"). Może ona zupełnie nie uwzględniać zależności nieliniowej (np. „jeśli wartość x zbliża się do zera, to wartość y zazwyczaj rośnie"). Rysunek 2.16 ukazuje różnorodność zestawów danych wraz z ich współczynnikiem korelacji. Zwróć uwagę, że wszystkie wykresy w dolnym rzędzie mają współczynnik korelacji równy 0, pomimo że ich



Rysunek 2.16. Standardowy współczynnik korelacji dla różnych zbiorów danych (źródło: Wikipedia; rysunek stanowi własność publiczną)

osie nie są wzajemnie niezależne: widzimy tu przykład zależności nieliniowej. Ponadto w drugim rzędzie są ukazane przykłady wykresów, w których współczynnik korelacji wynosi 1 lub –1; jak widać, nie ma to żadnego związku z nachyleniem prostej. Przykładowo Twój wzrost w centymetrach ma współczynnik korelacji liniowej równy 1 dla Twojego wzrostu podanego w metrach lub nanometrach.

Eksperymentuj z kombinacjami atrybutów

Mam nadzieję, że dzięki informacjom zawartym w poprzednich punktach poznałeś już kilka sposobów eksplorowania danych i wydobywania z nich dodatkowej wiedzy. Wykryliśmy kilka artefaktów, które warto byłoby usunąć przed przesłaniem danych do algorytmu uczenia maszynowego, a do tego zauważyliśmy kilka interesujących korelacji pomiędzy atrybutami, zwłaszcza w związku z naszym docelowym atrybutem. Zwróciliśmy także uwagę, że niektóre atrybuty cechują się rozkładem prawoskośnym, dlatego wypadałoby je w jakiś sposób przekształcić (np. poprzez wyliczenie ich logarytmu lub pierwiastka kwadratowego). Oczywiście każdy projekt jest inny, ale ogólne koncepcje są dość podobne.

Ostatnią czynnością, jaką należałoby wykonać przed przygotowaniem danych do użytku algorytmów uczenia maszynowego, jest wypróbowanie różnych kombinacji atrybutów. Przykładowo całkowita liczba pomieszczeń w dystrykcie nie jest zbyt wartościowym atrybutem, jeśli nie znamy liczby przebywających tam rodzin. W rzeczywistości interesuje nas liczba pokojów przypadających na rodzinę. Również całkowita liczba sypialni sama w sobie nic nam nie mówi: prawdopodobnie należałoby ją porównać z całkowitą liczbą pomieszczeń. Inną ciekawą kombinacją atrybutów jest określenie zależności pomiędzy populacją a liczbą rodzin. Tworzysz te nowe atrybuty w następujący sposób:

```
housing["pokoje_na_rodzinę"] = housing["total_rooms"]/housing["households"]
housing["współczynnik_sypialni"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["liczba_osób_na_dom"]=housing["population"]/housing["households"]
```

Następnie przyglądasz się ponownie uzyskanej macierzy korelacji:

```
>>> corr_matrix = housing.corr(numeric_only=True)
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value      1.000000
median_income           0.688380
pokoje_na_rodzinę       0.143663
total_rooms             0.137455
housing_median_age      0.102175
households              0.071426
total_bedrooms          0.054635
population             -0.020153
liczba_osób_na_dom     -0.038224
longitude              -0.050859
latitude               -0.139584
współczynnik_sypialni  -0.256397
Name: median_house_value, dtype: float64
```

Całkiem nieźle! Nowy atrybut `współczynnik_sypialni` jest znacznie bardziej skorelowany z medianą cen mieszkań niż całkowita liczba pomieszczeń lub sypialni. Jest to wyraźna korelacja ujemna, co sugeruje, że domy o mniejszym stosunku liczby sypialni do liczby pomieszczeń okazują się droższe. Liczba pokojów przypadająca na rodzinę również dostarcza nam więcej informacji niż całkowita liczba pomieszczeń w dystrykcie — jest dość oczywiste, że wraz z powierzchnią domu rośnie jego cena.

Przy tworzeniu nowych złożonych cech upewnij się, że nie są one zbyt liniowo skorelowane z istniejącymi cechami: **współliniowość** może powodować problemy w niektórych modelach, takich jak regresja liniowa. W szczególności unikaj prostych ważonych sum istniejących cech.

Ten etap eksploracji danych nie musi być bardzo gruntowny; mamy dzięki niemu dobrze wejść w projekt i szybko uzyskać informacje pomagające w stworzeniu pierwszego odpowiednio dobrego prototypu. Jest to jednak wielokrotnie powtarzany proces: po stworzeniu i uruchomieniu projektu możesz przeanalizować uzyskiwane za jego pomocą wyniki i korzystając ze zdobytych w ten sposób informacji, powrócić do etapu eksploracji danych.

Przygotuj dane pod algorytmy uczenia maszynowego

Czas przygotować dane dla algorytmów uczenia maszynowego. Mamy kilka powodów, aby nie zajmować się tym własnoręcznie, lecz napisać funkcje odpowiedzialne za ten proces:

- W ten sposób będziemy w stanie z łatwością odtwarzać te transformacje na dowolnym zbiorze danych (np. przy okazji całkowicie świeżego zbioru danych).
- Możemy stopniowo powiększać bibliotekę funkcji przekształcających, które możemy wykorzystywać w następnych projektach.
- Możemy używać tych funkcji w już działającym systemie, aby przekształcać nowe dane przed przesłaniem ich do algorytmów uczenia maszynowego.
- Będziemy w stanie z łatwością wypróbowywać różne rodzaje przekształceń i sprawdzać, która kombinacja transformacji sprawuje się najlepiej.

Najpierw jednak wróć do pierwotnego zestawu uczącego (poprzez ponowne skopiowanie obiektu `strat_train_set`). Powinieneś także rozdzielić czynniki prognostyczne od etykiet, ponieważ nie chcesz dokonywać takich samych przekształceń na obydwu typach danych (zwróć uwagę, że funkcja `drop()` tworzy kopię danych i nie wpływa na zbiór `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Oczyść dane

Większość algorytmów uczenia maszynowego nie może działać, jeśli brakuje jakichś cech, dlatego musisz się tym zająć. Na przykład zauważyłeś wcześniej, że w atrybucie `total_bedrooms` brakuje pewnych wartości. Mamy trzy możliwości:

1. pozbyć się dystryktów zawierających brakujące dane;
2. pozbyć się całego atrybutu;
3. uzupełnić brakujące dane określoną wartością (zero, średnia, mediana itd.). Jest to tak zwana **imputacja** (ang. *imputation*).

Możemy tego łatwo dokonać za pomocą metod `dropna()`, `drop()` i `fillna()`, stanowiących część klasy `DataFrame`:

```
housing.dropna(subset=["total_bedrooms"], inplace=True) # opcja 1.

housing.drop("total_bedrooms", axis=1, inplace=True) # opcja 2.

median = housing["total_bedrooms"].median() # opcja 3.
housing["total_bedrooms"]=housing["total_bedrooms"].fillna(median)
```

Wybierasz trzecie rozwiązanie, ponieważ jest najmniej destrukcyjne, jednak zamiast powyższego kodu skorzystasz z przydatnej klasy `SimpleImputer` stanowiącej część biblioteki *Scikit-Learn*. Dużą zaletą jest możliwość przechowywania wartości mediany każdej cechy: dzięki temu możliwe będzie wstawianie brakujących wartości nie tylko w zbiorze uczącym, ale także walidacyjnym, testowym oraz we wszelkich nowych danych wprowadzanych do modelu. Żeby z niej skorzystać, musisz najpierw stworzyć wystąpienie klasy `SimpleImputer`, w którym zaznaczamy, że chcemy zastąpić brakujące wartości każdego atrybutu medianą tego atrybutu:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

Mediana może być wyliczana jedynie z wartości numerycznych, musisz zatem stworzyć kopię zbioru danych wyłącznie za pomocą atrybutów numerycznych (co oznacza pominięcie atrybutu `ocean_proximity`):

```
housing_num = housing.select_dtypes(include=[np.number])
```

Możemy teraz dopasować wystąpienie klasy `Imputer` do danych uczących za pomocą metody `fit()`:

```
imputer.fit(housing_num)
```

Klasa `imputer` po prostu obliczyła medianę atrybutu i zachowała wyniki w zmiennej `statistics_`. Jedynie w atrybucie `total_bedrooms` brakuje niektórych wartości, nie wiesz jednak, czy po uruchomieniu ostatecznej wersji systemu będą dostępne wszystkie wartości w nowych zestawach danych, dlatego najbezpieczniejszym rozwiązaniem jest zastosowanie klasy `imputer` wobec wszystkich atrybutów numerycznych:

```
>>> imputer.statistics_
array([-118.51  ,   34.26  ,   29.  , 2125.  ,  434.  , 1167.  ,  408.  ,    3.5835])
>>> housing_num.median().values
array([-118.51  ,   34.26  ,   29.  , 2125.  ,  434.  , 1167.  ,  408.  ,    3.5835])
```

Możesz teraz wykorzystać tak „wytrenowaną" klasę `imputer` do przekształcenia zbioru uczącego, zastępując brakujące wartości obliczonymi medianami:

```
X = imputer.transform(housing_num)
```

Można również zastąpić brakujące wartości wartością średnią (`strategy="mean"`) lub najczęściej występującą wartością (`strategy="most_frequent"`) albo wartością stałą (`strategy="constant", fill_value=…`). Dwie ostatnie strategie obsługują wartości nienumeryczne.

Istnieją jeszcze bardziej zaawansowane imputery w pakiecie *sklearn.impute* (obydwa przeznaczone wyłącznie dla cech numerycznych):

- `KNNImputer` zastępuje każdą brakującą wartość średnią z wartości *k*-najbliższych sąsiadów dla tej cechy. Odległość jest obliczana na podstawie wszystkich dostępnych cech.
- `IterativeImputer` trenuje model regresyjny dla każdej cechy po to, aby przewidywać brakujące wartości na podstawie wszystkich pozostałych cech. Następnie trenuje model za pomocą zaktualizowanych danych i powtarza ten proces kilka razy, poprawiając w każdej iteracji modele oraz zastępowane wartości.

Konstrukcja modułu Scikit-Learn

Moduł *Scikit-Learn* wyróżnia się wyjątkowo dobrze przemyślaną konstrukcją. Oto główne założenia projektowe[9] (*https://homl.info/11*):

- **Jednolitość** — wszystkie obiekty korzystają z jednolitego, prostego interfejsu:
  - **Estymatory** (**funkcje oszacowujące**). Każdy obiekt zdolny do szacowania pewnych parametrów na podstawie zbioru danych jest zwany **estymatorem** (np. klasa `SimpleImputer` jest funkcją oszacowującą). Sama operacja szacowania jest wykonywana przez metodę `fit()`, zaś jej parametrem jest zbiór danych lub dwa zestawy danych w przypadku algorytmów uczenia nadzorowanego; drugi zbiór zawiera etykiety. Wszelkie inne parametry wpływające na przebieg operacji szacowania są uznawane za hiperparametry (np. parametr `strategy` klasy `SimpleImputer`) — muszą być one wyznaczane jako zmienne wystąpienia (generalnie w postaci parametru konstruktora).

[9] Więcej informacji na temat założeń projektowych znajdziesz w dokumencie *API design for machine learning software: experiences from the scikit-learn project*, Lars Buitinck i in., arXiv preprint arXiv: 1309.0238: 2013.

- **Transformatory** (**funkcje transformujące** lub **przekształcające**). Niektóre estymatory (jak na przykład klasa `SimpleImputer`) są w stanie również przekształcać zbiór danych; są one zwane **transformatorami**. Także w tym przypadku ich interfejs nie jest skomplikowany: proces transformacji jest przeprowadzany za pomocą metody `transform()`, a jego parametr stanowi modyfikowany zbiór danych. W rezultacie zostaje zwrócony przekształcony zbiór danych. Operacja transformacji zależy przede wszystkim od wyuczonych parametrów (tak jak w przypadku klasy `Simple`
  `↪Imputer`). Wszystkie transformatory zawierają również metodę złożoną `fit_`
  `↪transform()`, która jest równoznaczna z wywołaniem metody `fit()`, a po niej `transform()` (czasami jednak ta metoda złożona jest zoptymalizowana i działa znacznie szybciej od jej elementów składowych).
- **Predyktory** (**funkcje prognostyczne**). Pewne estymatory są w stanie przewidywać wyniki na podstawie zbioru danych; są to tak zwane **predyktory**. Na przykład model `LinearRegression` z poprzedniego rozdziału stanowi predyktor: przewidzieliśmy za jego pomocą satysfakcję z życia, znając wartość PKB per capita danego kraju. Funkcja prognostyczna zawiera metodę `predict()` przyjmującą nowe zbiory danych i zwracającą zestaw powiązanych z nimi prognoz. Dodatkowo występuje tu także metoda `score()`, mierząca jakość prognoz na podstawie zbioru testowego (oraz powiązanych etykiet w przypadku algorytmów uczenia nadzorowanego)[10].

- **Inspekcja** — wszystkie hiperparametry estymatorów są bezpośrednio dostępne poprzez publiczne zmienne wystąpień (np. `imputer.strategy`); wszystkie wyuczone parametry funkcji oszacowującej są dostępne poprzez tego typu zmienne oznaczone na końcu podkreślnikiem (np. `imputer.statistics_`).
- **Nierozprzestrzenianie klas**. Zbiory danych nie są reprezentowane w postaci własnoręcznie przygotowanych klas, lecz jako macierze *NumPy* lub macierze rzadkie *SciPy*. Z kolei hiperparametry to standardowe ciągi znaków lub wartości liczbowe języka Python.
- **Kompozycja**. Istniejące elementy składowe są używane tak często, jak to możliwe. Przykładowo łatwo stworzyć estymator `Pipeline` z samodzielnie dobranej sekwencji funkcji transformujących zakończonych ostatnią funkcją oszacowującą, o czym przekonasz się już niebawem.
- **Rozsądne wartości domyślne**. Moduł *Scikit-Learn* zawiera przemyślane wartości domyślne dla większości parametrów, dzięki czemu możemy z łatwością tworzyć bazowy system roboczy.

Transformatory *Scikit-Learn* dają na wyjściu tablice *NumPy* (a czasami macierze rzadkie *SciPy*) nawet wtedy, gdy na wejściu są im dostarczane obiekty `DataFrame` modułu *Pandas*[11]. Dlatego wynikiem funkcji `imputer.transform(housing_num)` jest tablica *NumPy*: `X` nie zawiera

[10] Niektóre funkcje prognostyczne zawierają również metody mierzące pewność wyliczonych prognoz.

[11] Jeśli ustawisz `sklearn.set_config(transform_output="pandas")`, wszystkie transformatory będą zwracać obiekty `DataFrame` biblioteki *pandas*, gdy otrzymają `DataFrame` jako dane wejściowe: obiekty *pandas* na wejściu, obiekty *pandas* na wyjściu.

nazw kolumn ani indeksu. Na szczęście nie jest trudno umieścić `X` w obiekcie `DataFrame` i odzyskać nazwy kolumn oraz indeks z `housing_num`:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                                          index=housing_num.index)
```

Obsługa tekstu i atrybutów kategorialnych

Do tej pory zajmowaliśmy się wyłącznie atrybutami numerycznymi, ale Twoje dane mogą zawierać również atrybuty tekstowe. W używanym przez nas zestawie danych występuje tylko jeden atrybut tego typu: ocean_proximity. Przyjrzyjmy się wartościom jego pierwszych kilku przykładów:

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(8)
       ocean_proximity
13096         NEAR BAY
14973        <1H OCEAN
3785            INLAND
14689           INLAND
20507       NEAR OCEAN
1286            INLAND
18078        <1H OCEAN
4396          NEAR BAY
```

Nie jest to typowy tekst: istnieje tu ograniczona liczba wartości, z których każda reprezentuje atrybut kategorialny. Większość algorytmów uczenia maszynowego lepiej sobie radzi z liczbami, przekształćmy więc te kategorie z tekstu na wartości numeryczne. Możemy w tym celu użyć klasy `OrdinalEncoder`:

```
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

Kilka pierwszych zakodowanych wartości w `housing_cat_encoded` wygląda następująco:

```
>>> housing_cat_encoded[:8]
array([[3.],
       [0.],
       [1.],
       [1.],
       [4.],
       [1.],
       [0.],
       [3.]])
```

Możemy uzyskać listę kategorii za pomocą wystąpienia zmiennej `categories_`. Jest to lista zawierająca jednowymiarową tablicę kategorii dla każdego atrybutu kategorialnego (w tym przypadku mamy do czynienia z listą zawierającą jedną tablicę, ponieważ dysponujemy jednym atrybutem kategorialnym):

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Istnieje pewien problem z tym rozwiązaniem: algorytmy uczenia maszynowego będą uznawały, że dwie zbliżone wartości będą bardziej do siebie podobne niż do dalszych wartości. W pewnych przypadkach nie stanowi to problemu (np. przy uporządkowanych kategoriach, takich jak „zły", „przeciętny", „dobry" i „znakomity"), ale oczywiście nie dotyczy to kolumny `ocean_proximity` (np. kategorie 0 i 4 są wyraźnie bardziej do siebie podobne niż 0 i 1). Powszechnie stosowanym rozwiązaniem jest stworzenie jednego binarnego atrybutu dla każdej kategorii: jeden atrybut ma wartość 1, gdy kategorią jest `<1H OCEAN` (w przeciwnym wypadku otrzymuje wartość 0), inny atrybut uzyskuje wartość 1 dla kategorii `INLAND` (i 0 dla pozostałych kategorii) itd. Jest to tzw. **kodowanie „gorącojedynkowe"** (ang. *one-hot encoding*), ponieważ tylko jeden atrybut będzie „gorący" (będzie miał wartość 1), podczas gdy pozostałe będą „zimne" (wartość 0). Nowe atrybuty są czasami nazywane **atrybutami sztucznymi** (ang. *dummy attributes*). Moduł *Scikit-Learn* zawiera koder `OneHotEncoder`, konwertujący kategorialne wartości całkowite na wektory „gorącojedynkowe":

```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat.encoder.fit_transform(housing_cat)
```

Domyślnie wynikiem funkcji `OneHotEncoder` jest **macierz rzadka** (ang. *sparse matrix*) biblioteki *SciPy*, a nie tablica *NumPy*:

```
>>> housing_cat_1hot
<Compressed Sparse Row sparse matrix of dtype 'float64'
with 16512 stored elements and shape (16512, 5)>
```

Macierz rzadka jest bardzo skutecznym odwzorowaniem macierzy zawierających głównie zera. W istocie przechowywane są w niej jedynie wartości niezerowe oraz informacje o ich położeniu. Gdy atrybut kategorialny zawiera setki bądź tysiące kategorii, zakodowanie go w sposób „gorącojedynkowy" skutkuje powstaniem macierzy wypełnionej zerami, w której występuje tylko jedna jedynka na każdy wiersz. W tym przypadku macierz rzadka jest dokładnie tym, czego potrzebujesz: pozwala zaoszczędzić mnóstwo pamięci i przyspieszyć obliczenia. Przeważnie możesz korzystać z macierzy rzadkiej jak ze zwykłej macierzy dwuwymiarowej[12], a jeśli chcesz ją przekształcić w tablicę *NumPy* (gęstą), wystarczy wywołać metodę `toarray()`:

```
>>> housing_cat_1hot.toarray()
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.]], shape=(16512, 5))
```

Ewentualnie możesz wyznaczyć `sparse_output=False` podczas tworzenia klasy `OneHotEncoder`; w takim przypadku metoda `transform()` bezpośrednio zwróci zwykłą (gęstą) tablicę *NumPy*:

```
cat_encoder = OneHotEncoder(sparse_output=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat) # teraz jest to tablica gęsta
```

[12] Szczegóły znajdziesz w dokumentacji modułu SciPy.

Podobnie jak w przypadku klasy `OrdinalEncoder`, możemy uzyskać listę kategorii za pomocą wystąpienia zmiennej `categories_` kodera:

```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Biblioteka *Pandas* zawiera funkcję `get_dummies()`, która również przekształca każdą cechę kategorialną w odwzorowanie „gorącojedynkowe" zawierające po jednej cesze binarnej na kategorię:

```
>>> df_test = pd.DataFrame({"ocean_proximity": ["INLAND", "NEAR BAY"]})
>>> pd.get_dummies(df_test)
 ocean_proximity_INLAND      ocean_proximity_NEAR                    BAY
0                            True                                    False
1                            False                                   True
```

Wygląda to całkiem ładne i prosto, dlaczego więc nie użyć tego rozwiązania zamiast klasy `OneHotEncoder`? Zaletą klasy `OneHotEncoder` jest to, że zapamiętuje ona kategorie, na których została wytrenowana. Jest to bardzo ważne, ponieważ gdy Twój model znajdzie się w środowisku produkcyjnym, powinien otrzymywać dokładnie takie same cechy jak w fazie uczenia: nie należy żadnej dodawać ani pomijać. Spójrz, jakie wyniki generuje nasz wytrenowany `cat_encoder`, gdy spróbujemy przekształcić ten sam `df_test` (za pomocą metody `transform()`, a nie `fit_transform()`):

```
>>> cat_encoder.transform(df_test)
array([[0., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

Widzisz różnicę? Metoda `get_dummies()` dostrzegła tylko dwie kategorie, dlatego umieszcza na wyjściu dwie kolumny, natomiast klasa `OneHotEncoder` daje po jednej kolumnie na każdą wyuczoną kategorię, we właściwej kolejności. Do tego jeśli dostarczysz metodzie `get_dummies()` obiekt `DataFrame` zawierający nieznaną kategorię (np. `"<2H OCEAN"`), to radośnie wygeneruje dla niej kolumnę:

```
>>> df_test_unknown = pd.DataFrame({"ocean_proximity": ["<2H OCEAN", "ISLAND"]})
>>> pd.get_dummies(df_test_unknown)
       ocean_proximity_<2H OCEAN          ocean_proximity_ISLAND
0      True                               False
1      False                              True
```

Jednak klasa `OneHotEncoder` jest „mądrzejsza": wykryje nieznaną kategorię i spowoduje wyświetlenie komunikatu o wyjątku. Jeżeli chcesz, możesz wyznaczyć wartość `"ignore"` hiperparametru `handle_unknown`, co sprawi, że nieznana kategoria będzie oznaczana jako 0:

```
>>> cat_encoder.handle_unknown = "ignore"
>>> cat_encoder.transform(df_test_unknown)
array([[0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

Jeżeli atrybut kategorialny zawiera znaczną liczbę kategorii (np. kod kraju, zawód, gatunek), to w wyniku kodowania „gorącojedynkowego" będziemy otrzymywać znaczną liczbę cech wejściowych. Może to spowolnić proces uczenia i zmniejszyć skuteczność. W takim przypadku warto zastąpić wejściowe dane kategorialne przydatnymi cechami numerycznymi powiązanymi z kategoriami, np. możesz zastąpić cechę *ocean_proximity* odległością do oceanu (na drodze analogii kod kraju może zostać zastąpiony populacją państwa i PKB per capita). Ewentualnie możesz wykorzystać jeden z koderów umieszczonych w pakiecie *category_encoders* dostępnym w serwisie GitHub (*https://github.com/scikit-learn-contrib/category_encoders*). Jeśli zaś modelujesz sieci neuronowe, możesz zamiast każdej kategorii wprowadzić poznawalny, małowymiarowy wektor zwany **wektorem właściwościowym** (ang. *embedding*; zobacz rozdział 14.). Jest to przykład **uczenia się reprezentacji** (ang. *representation learning*; więcej przykładów znajdziesz w rozdziale 18.).

Podczas dopasowywania dowolnego estymatora *Scikit-Learn* za pomocą obiektu `DataFrame` estymator przechowuje nazwy kolumn w atrybucie `feature_names_in_`. Biblioteka *Scikit-Learn* sprawia następnie, że każdy następny obiekt `DataFrame` dostarczany do tego estymatora (np. do metody `transform()` czy `predict()`) zawiera te same nazwy kolumn. Transformatory zawierają także metodę `get_feature_names_out()`, której możesz użyć do stworzenia obiektu `DataFrame` na podstawie rezultatu tego transformatora:

```
>>> cat_encoder.feature_names_in_
array(['ocean_proximity'], dtype=object)
>>> cat_encoder.get_feature_names_out()
array(['ocean_proximity_<1H OCEAN', 'ocean_proximity_INLAND',
       'ocean_proximity_ISLAND', 'ocean_proximity_NEAR BAY',
       'ocean_proximity_NEAR OCEAN'], dtype=object)
>>> df_output = pd.DataFrame(cat_encoder.transform(df_test_unknown),
...                          columns=cat_encoder.get_feature_names_out(),
...                          index=df_test_unknown.index)
...
```

Ta funkcja pomaga uniknąć niedopasowania kolumn i jest również bardzo przydatna podczas usuwania błędów.

Skalowanie i przekształcanie cech

Jednym z najważniejszych przekształceń dokonywanych na danych jest **skalowanie cech** (ang. *feature scaling*). Większość algorytmów uczenia maszynowego słabo sobie radzi z atrybutami numerycznymi znajdującymi się w różnych zakresach skali. Dotyczy to również naszego zbioru danych: całkowita liczba pomieszczeń mieści się w zakresie od 6 do 39 320, z kolei wartości mediany dochodów to zakres zaledwie od 0 do 15. Bez skalowania większość modeli będzie faworyzowała ignorowanie mediany dochodów i koncentrowała się bardziej na liczbie pomieszczeń.

Istnieją dwa popularne rodzaje skalowania wszystkich atrybutów do jednego poziomu: **skalowanie min. – max.** (ang. *min-max scaling*) i **standaryzacja** (ang. *standarization*).

Podobnie jak w przypadku wszystkich estymatorów, należy dostosować funkcje skalujące wyłącznie do danych uczących: nigdy nie używaj metod `fit()` lub `fit_transform()` do innych zbiorów danych. Dopiero po wyuczeniu klasy skalującej możesz użyć metody `transform()` do przekształcenia dowolnego innego typu zbioru danych, w tym zbioru walidacyjnego, zbioru testowego oraz nowych danych. Zwróć uwagę, że chociaż wartości zestawu uczącego będą zawsze skalowane do określonego zakresu, to jeżeli nowe dane będą zawierać jakieś elementy odstające, to mogą one wylądować poza skalą. Jeżeli chcesz tego uniknąć, wyznacz wartość `True` hiperparametru `clip`.

Skalowanie min. – max. (zwane przez wiele osób **normalizacją**) jest najprostszym procesem: dla każdego atrybutu wartości są tak przesuwane i skalowane, że mieszczą się w zakresie pomiędzy 0 i 1. Odbywa się to poprzez odjęcie wartości minimalnej od wszystkich wartości i podzielenie wyników przez różnicę między wartością minimalną a maksymalną. W module *Scikit-Learn* służy do tego funkcja transformująca `MinMaxScaler`. Zawiera ona hiperparametr `feature_range`, pozwalający zmieniać zakres skali, jeśli z jakiegoś powodu nie odpowiada Ci domyślny zakres 0 – 1 (np. sieci neuronowe działają najlepiej z danymi o średniej wynoszącej zero, dlatego preferowany jest zakres od –1 do 1). Jest dość łatwo go użyć:

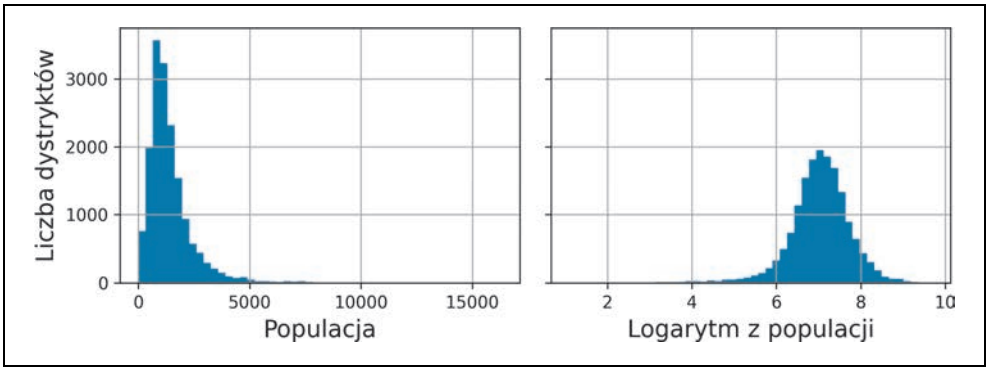
```
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

Mechanizm standaryzacji jest odmienny: najpierw od danej wartości jest odejmowana średnia (czyli średnia w standaryzowanych próbkach zawsze wynosi 0), a następnie wynik jest dzielony przez odchylenie standardowe (dzięki czemu odchylenie standardowe wartości standaryzowanych jest równe 1). W przeciwieństwie do skalowania min. – max. standaryzacja nie ogranicza skalowanych wartości do określonego zakresu. Z drugiej strony standaryzacja jest znacznie mniej wrażliwa na elementy odstające. Załóżmy na przykład, że przez pomyłkę mediana dochodów dla danego dystryktu jest równa 100 zamiast standardowego przedziału 0 – 15. Skalowanie metodą min. – max. do zakresu od 0 do 1 spowodowałoby odwzorowanie tego elementu odstającego w punkcie 1, przez co wszystkie pozostałe wartości zostałyby umieszczone w zakresie 0 – 0,15, natomiast nie miałoby to tak wielkiego wpływu na standaryzowane wyniki. Standaryzację w module *Scikit-Learn* uzyskujemy za pomocą transformatora `StandardScaler`:

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

Jeżeli chcesz skalować macierz rzadką bez konieczności uprzedniego jej przekształcenia w macierz gęstą, możesz użyć klasy `StandardScaler` z wartością `False` hiperparametru `with_mean`: dane zostaną podzielone wyłącznie przez odchylenie standardowe, z pominięciem odejmowania średniej (gdyż zniweczyłoby to rzadkość macierzy).

Gdy rozkład cechy jest **gruboogonowy** (ang. *heavy tail*; np. wartości znajdujące się daleko od średniej nie pojawiają się wyjątkowo rzadko), zarówno skalowanie metodą min. – max., jak i standaryzacja „ścisną" większość wartości w wąskim zakresie. W przypadku większości modeli uczenia maszynowego jest to niepożądane zjawisko, o czym przekonasz się w rozdziale 4. *Zanim* więc przeskalujesz cechę, powinieneś ją najpierw tak przekształcić, aby zredukować gruboogonowość i w miarę możliwości uzyskać w przybliżeniu symetryczny rozkład. Na przykład popularnym sposobem uzyskania takiego rezultatu w przypadku cech dodatnich mających rozkład gruboogonowy prawostronny jest zastąpienie wartości cechy ich pierwiastkami kwadratowymi (lub spotęgowanie cechy do potęgi między 0 a 1). Jeżeli cecha ma bardzo długi i gruby ogon, jak w przypadku **rozkładu potęgowego** (ang. *power law distribution*), to może pomóc zastąpienie wartości cechy ich logarytmami. Na przykład cecha `population` w przybliżeniu jest zgodna z prawem potęgowym: dystrykty zamieszkiwane przez 10 000 mieszkańców występują jedynie dziesięciokrotnie (a nie wykładniczo) rzadziej od dystryktów zamieszkiwanych przez 1000 mieszkańców. Na rysunku 2.17 widać, że cecha ta wygląda znacznie lepiej po obliczeniu jej logarytmu: zaczyna bardzo przypominać rozkład Gaussa (dzwonowy).



Rysunek 2.17. Przekształcanie cechy w sposób przybliżający jej rozkład do rozkładu Gaussa

Inną metodą obsługi cech gruboogonowych jest **kubełkowanie** (ang. *bucketizing*) cechy. Oznacza to podział rozkładu na zbliżonych rozmiarów przedziały i zastąpienie wartości każdej cechy indeksem przedziału, do którego ta wartość należy, podobnie jak robiliśmy to podczas tworzenia cechy `income_cat` (mimo że użyliśmy jej wyłącznie w próbkowaniu warstwowym). Mógłbyś na przykład zastąpić każdą wartość jej percentylem. Kubełkowanie przy użyciu przedziałów o równej szerokości skutkuje uzyskaniem cechy o rozkładzie niemal jednostajnym, dlatego nie trzeba jej dalej skalować; ewentualnie możesz podzielić ją przez liczbę przedziałów, aby wymusić wartości w przedziale od 0 do 1.

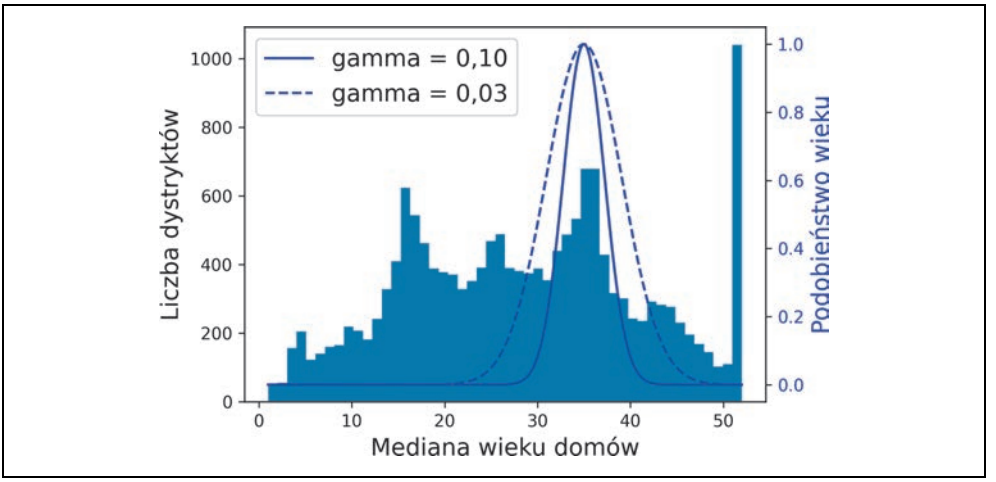
W przypadku cechy o rozkładzie wielomodalnym (czyli takiej, której rozkład ma co najmniej dwa wyraźne szczyty, zwane **modami**), na przykład `housing_median_age`, przydatne może być jej kubełkowanie, jednak w tym przypadku identyfikatory przedziałów będą traktowane jako kategorie, a nie wartości numeryczne. Oznacza to, że należy zakodować indeksy przedziałów, na przykład za pomocą klasy `OneHotEncoder` (dlatego zazwyczaj nie chcesz korzystać ze zbyt dużej liczby przedziałów). Rozwiązanie to umożliwi modelowi regresyjnemu łatwiej odkrywać

różne reguły dla poszczególnych zakresów danej wartości cechy. Na przykład być może domy zbudowane jakieś 35 lat temu miały niecodzienny kształt, odstający od ówczesnych trendów, dlatego są tańsze, niż wskazywałby na to ich wiek.

Kolejnym sposobem przekształcania rozkładów wielomodalnych jest dodawanie cechy dla każdego z modów (przynajmniej głównych), tak by odzwierciedlić podobieństwo między medianą wieku domów a danym modem. Wskaźnik podobieństwa jest zazwyczaj obliczany za pomocą **radialnej funkcji bazowej** (ang. *radial basis function* — RBF), czyli dowolnej funkcji, która zależy wyłącznie od odległości między wartością wejściową a stałym punktem. Najczęściej stosowaną radialną funkcją bazową jest gaussowska RBF, której wartość wynikowa maleje wykładniczo w miarę oddalania się wartości wejściowej od ustalonego punktu. Na przykład podobieństwo obliczone gaussowską RBF między wiekiem domu x a 35 jest uzyskiwane wzorem $exp(-\gamma(x-35)^2)$. Hiperparametr γ (gamma) określa szybkość rozkładu wskaźnika podobieństwa w miarę oddalania się x od wartości 35. Za pomocą funkcji `rbf_kernel()` z biblioteki *Scikit-Learn* możesz stworzyć nową cechę gaussowskiej RBF, mierzącą podobieństwo między medianą wieku domów a wartością 35:

```
from sklearn.metrics.pairwise import rbf_kernel

age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]], gamma=0.1)
```

Rysunek 2.18 ukazuje tę nową cechę w funkcji mediany wieku domów (linia ciągła). Widać także, jak wyglądałaby ta cecha po użyciu mniejszej wartości `gamma`. Zgodnie z wykresem nowa cecha podobieństwa wieku ma szczytową wartość w punkcie 35, czyli w okolicach maksimum rozkładu mediany wieku domów: jeżeli ta konkretna grupa wiekowa jest dobrze skorelowana z niższymi cenami, to istnieje spore prawdopodobieństwo, że ta cecha okaże się przydatna.



Rysunek 2.18. Cecha gaussowskiej RBF mierząca podobieństwo między medianą wieku domów a wartością 35

Do tej pory przyglądaliśmy się jedynie cechom wejściowym, ale czasami może być konieczne także przekształcanie cech docelowych. Na przykład jeśli rozkład cechy docelowej jest grubogonowy, być może zechcesz zastąpić ją jej logarytmem. Jeśli jednak to zrobisz, model regresyjny będzie teraz przewidywał **logarytm** mediany wartości domu, a nie samą medianę wartości domu. Będziesz musiał obliczyć wykładnik przewidywań modelu, aby otrzymać przewidywaną medianę wartości domu.

Na szczęście większość transformatorów *Scikit-Learn* zawiera metodę `inverse_transform()`, dzięki czemu obliczenie odwrotności przekształceń staje się łatwe. Na przykład poniższy kod prezentuje mechanizm skalowania etykiet za pomocą klasy `StandardScaler` (tak samo, jak robiliśmy to w przypadku danych wejściowych), a następnie wytrenowania prostego modelu regresji liniowej na przeskalowanych etykietach oraz użycia go do uzyskania przewidywań na podstawie nowych danych, które następnie przekształcamy z powrotem do pierwotnej skali za pomocą metody `inverse_transform()` wytrenowanej klasy skalującej. Zwróć uwagę, że przekształcamy etykiety z obiektu `Series` w `DataFrame`, ponieważ klasa `StandardScaler` oczekuje danych dwuwymiarowych. Poza tym dla uproszczenia trenujemy w tym przykładzie model jedynie na pojedynczej, nieprzetworzonej cesze wejściowej (medianie dochodów):

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5]  # udajemy, że są to nowe dane

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

Rozwiązanie to sprawdza się dobrze, ale prostszym i mniej podatnym na błędy podejściem jest użycie klasy `TransformedTargetRegressor` — pozwala uniknąć potencjalnych niedopasowań w skalowaniu. Musimy ją tylko skonstruować, podając model regresyjny i transformator etykiet, a następnie dopasowując ją do zestawu uczącego przy użyciu pierwotnych, nieskalowanych etykiet. Automatycznie wykorzysta transformator do skalowania etykiet i wytrenuje model regresji na otrzymanych skalowanych etykietach, tak samo jak robiliśmy to wcześniej. Następnie, gdy będziemy chcieli uzyskać przewidywania, wywoła metodę `predict()` modelu regresji i wykorzysta metodę `inverse_transform()` klasy skalującej w celu uzyskania predykcji:

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                   transformer=StandardScaler())
model.fit(housing[["median_income"]], housing_labels)
predictions = model.predict(some_new_data)
```

Niestandardowe transformatory

Mimo że moduł *Scikit-Learn* zawiera wiele przydatnych funkcji przekształcających, od czasu do czasu jesteśmy zmuszeni pisać własne transformatory wykonujące operacje niestandardowych przekształceń, oczyszczania danych lub łączenia określonych atrybutów.

W przypadku przekształceń niewymagających żadnego treningu wystarczy napisać funkcję przyjmującą na wejściu tablicę *NumPy* i dającą na wyjściu przekształconą tablicę. Na przykład, jak już wiemy z poprzedniego punktu, często warto jest przekształcać cechy o rozkładach gruboogonowych, zastępując je ich logarytmem (przy założeniu, że rozkład jest prawostronnie gruboogonowy, a cecha ma wartości dodatnie). Stwórzmy transformator logarytmiczny i zastosujmy go wobec cechy `population`:

```
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop = log_transformer.transform(housing[["population"]])
```

Argument `inverse_func` jest nieobowiązkowy. Pozwala określić funkcję odwrotnego przekształcenia, na przykład jeżeli zamierzasz użyć transformatora w klasie `TransformedTarget` ↪`Regressor`.

Twoja funkcja przekształcająca może przyjmować hiperparametry jako dodatkowe argumenty. Na przykład możemy stworzyć transformator obliczający ten sam wskaźnik podobieństwa (gaussowską RBF) jak wcześniej:

```
rbf_transformer = FunctionTransformer(rbf_kernel,
                                      kw_args=dict(Y=[[35.]], gamma=0.1))
age_simil_35 = rbf_transformer.transform(housing[["housing_median_age"]])
```

Zauważ, że nie ma funkcji odwrotnej dla jądra RBF, ponieważ zawsze istnieją dwie wartości w danej odległości od ustalonego punktu (z wyjątkiem odległości 0). Zwróć także uwagę, że metoda `rbf_kernel()` nie traktuje cech osobno. Jeżeli przekażesz jej tablicę zawierającą dwie cechy, zmierzy dwuwymiarową (euklidesową) odległość w celu określenia podobieństwa. Na przykład w poniższy sposób dodajemy cechę mierzącą podobieństwo geograficzne między każdym dystryktem a San Francisco:

```
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel,
                                     kw_args=dict(Y=[sf_coords], gamma=0.1))
sf_simil = sf_transformer.transform(housing[["latitude", "longitude"]])
```

Transformatory niestandardowe przydają się także do łączenia cech. Na przykład poniżej widać klasę `FunctionTransformer` obliczającą proporcje między cechami wejściowymi 0 i 1:

```
>>> ratio_transformer = FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])
>>> ratio_transformer.transform(np.array([[1., 2.], [3., 4.]]))
array([[0.5 ],
       [0.75]])
```

Klasa `FunctionTransformer` jest bardzo przydatna, ale co w przypadku, gdybyś chciał, aby można było trenować Twój transformator, który uczyłby się pewnych parametrów w metodzie `fit()`, po czym wykorzystywał je później w metodzie `transform()`? W tym celu musisz napisać niestandardową klasę.

Moduł *Scikit-Learn* polega na technice inferencji typów (ang. *duck typing*)[13], więc niestandardowe klasy transformatorów nie muszą dziedziczyć z jakiejś konkretnej klasy bazowej. Wystarczą im trzy metody: `fit()` (zwracająca wartość obiektu `self`), `transform()` oraz `fit_transform()`.

[13] W przypadku inferencji typów liczą się metody i działanie danego obiektu, a nie jego typ.

W dalszej części tego podrozdziału wyjaśniam, jak definiować niestandardowe klasy transformatorów. W szczególności zdefiniuję niestandardowy transformator, który grupuje dzielnice w dziesięć geograficznych skupień, a następnie mierzy odległość między każdą dzielnicą a środkiem każdego skupienia i dodaje dziesięć odpowiadających cech podobieństwa RBF do danych. Ponieważ definiowanie niestandardowych klas transformatorów jest nieco zaawansowane, możesz śmiało przejść do następnego podrozdziału i wrócić do tego tematu, gdy będzie to potrzebne.

Tę ostatnią metodę możemy uzyskać „za darmo", dodając bazową klasę `TransformerMixin`: domyślna implementacja będzie wywoływać jedynie metodę `fit()`, a następnie `transform()`. Jeśli dołączasz klasę bazową `BaseEstimator` (i unikasz zmiennych `*args` oraz `**kargs` w konstruktorze), uzyskasz także dostęp do dwóch dodatkowych metod (`get_params()` i `set_params()`). Przydadzą się one do automatycznego strojenia hiperparametrów.

Na przykład poniżej przedstawiam niestandardowy transformator zachowujący się w dużej mierze jak klasa `StandardScaler`:

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted

class StandardScalerClone(BaseEstimator, TransformerMixin):
    def __init__(self, with_mean = True): # Żadnych zmiennych *args ani **kwargs
        self.with_mean = with_mean

    def fit(self, X, y=None):  # y jest wymagana, mimo że jej nie używamy
        X = check_array(X)  # sprawdza, czy X jest tablicą ze skończonymi wartościami
                            # zmiennoprzecinkowymi
        self.mean_ = X.mean(axis=0)
        self.scale_ = X.std(axis=0)
        self.n_features_in_ = X.shape[1]  # każdy estymator przechowuje to w metodzie fit()
        return self  # zawsze zwraca self!

    def transform(self, X):
        check_is_fitted(self)  # wyszukuje poznane atrybuty (ze znacznikiem _)
        X = check_array(X)
        assert self.n_features_in_ == X.shape[1]
        if self.with_mean:
            X = X - self.mean_
        return X / self.scale_
```

Kilka uwag:

- Pakiet *sklearn.utils.validation* zawiera kilka funkcji, za pomocą których możemy walidować dane wejściowe. Dla uproszczenia będziemy pomijać takie testy w całej książce, ale kod używany w środowisku produkcyjnym powinien je zawierać.
- Potoki *Scikit-Learn* wymagają, aby metoda `fit()` zawierała dwa argumenty, `X` i `y`, dlatego potrzebujemy argumentu `y=None`, mimo że nie używamy `y`.
- Wszystkie estymatory *Scikit-Learn* wyznaczają `n_features_in_` wewnątrz metody `fit()` i sprawiają, że dane przekazywane do metod `transform()` lub `predict()` zawierają właśnie taką liczbę cech.

- Metoda `fit()` musi zwracać parametr `self`.
- Implementacja ta nie jest stuprocentowo pełna: wszystkie estymatory powinny wyznaczyć `feature_names_in_` w metodzie `fit()`, gdy jest im przekazywany obiekt `DataFrame`. Ponadto wszystkie transformatory powinny dostarczać metody `get_feature_names_out()` oraz `inverse_transform()`, gdy można odwrócić dane przekształcenie. Więcej szczegółów znajdziesz w ćwiczeniu na końcu tego rozdziału.

Niestandardowy transformator może (i często korzysta z tej możliwości) wykorzystywać w implementacji inne estymatory. Na przykład poniższy listing ukazuje niestandardowy transformator wykorzystujący klasę `KMeans` w metodzie `fit()` do wykrywania głównych skupień w danych, a następnie wprowadzający metodę `rbf_kernel()` do metody `transform()`, co pozwala zmierzyć podobieństwo każdej próbki do środka skupienia:

```
from sklearn.cluster import KMeans

class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self  # zawsze zwraca self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Podobieństwo {i} skupienia" for i in range(self.n_clusters)]
```

Możesz sprawdzić, czy Twój niestandardowy estymator jest zgodny z interfejsem Scikit-Learn, przekazując wystąpienie do metody `check_estimator()` z pakietu *sklearn.utils.estimator_checks*. Pełen opis API znajdziesz pod adresem *https://scikit-learn.org/stable/developers*.

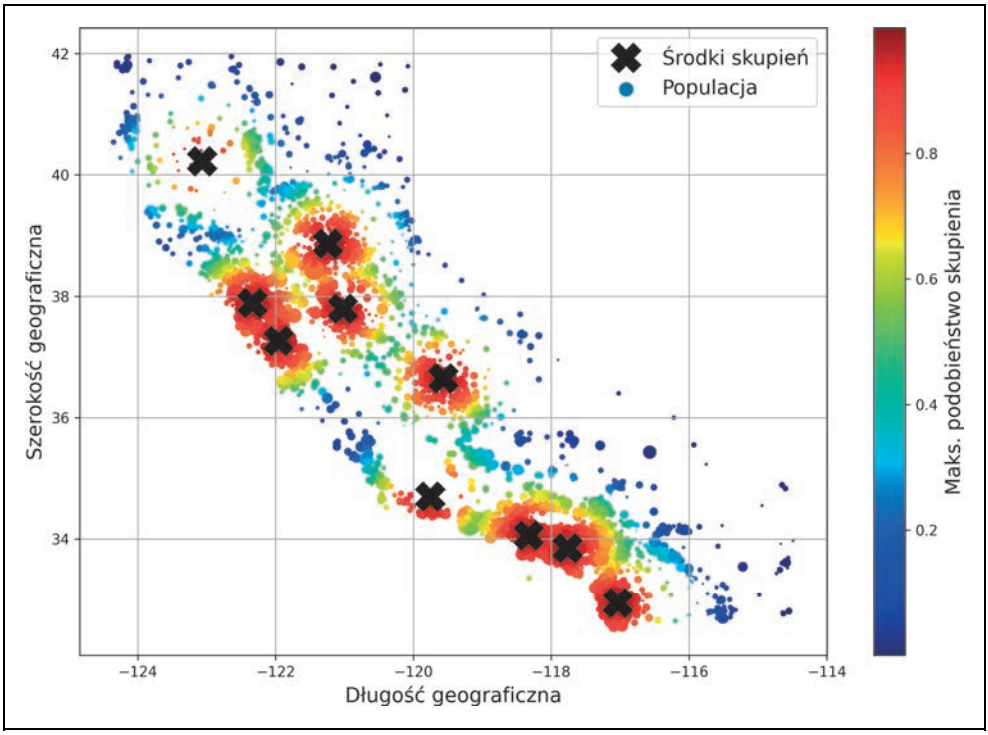
Jak dowiesz się w rozdziale 8., algorytm centroidów jest algorytmem analizy skupień wyszukującym zgrupowania danych. Można go na przykład wykorzystać do znalezienia najbardziej zaludnionych regionów w Kalifornii. Liczba skupień, których szuka algorytm centroidów, jest kontrolowana przez hiperparametr `n_clusters`. Metoda `fit()` klasy `KMeans` zawiera dodatkowy argument `sample_weight`, pozwalający użytkownikowi wyznaczać względne wagi próbek. Przykładowo, moglibyśmy przekazać medianę dochodów, gdybyśmy chcieli, aby skupienia częściej tworzyły się w rejonach zamożnych dzielnic. Po zakończeniu treningu środki skupień są dostępne w atrybucie `cluster_centers_`. Algorytm centroidów jest stochastyczny, co oznacza, że do wyszukiwania skupień wykorzystywana jest losowość, dlatego jeżeli chcesz otrzymywać odtwarzalne wyniki, musisz wyznaczyć parametr `random_state`. Jak widać, kod jest całkiem prosty mimo złożoności zadania. Skorzystajmy teraz z niestandardowego transformatora:

```
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
similarities = cluster_simil.fit_transform(housing[["latitude", "longitude"]])
```

Powyższy listing tworzy transformator `ClusterSimilarity`, wyznaczający dziesięć skupień. Następnie wywołuje metodę `fit_transform()` wraz ze współrzędnymi geograficznymi (szerokością i długością) każdego dystryktu ze zbioru uczącego (możesz spróbować określić wagę każdego dystryktu na podstawie jego mediany dochodów, aby sprawdzić jej wpływ na poszczególne skupienia). Transformator wykorzystuje algorytm centroidów do wykrywania skupień, a następnie mierzy podobieństwo metodą gaussowskiej RBF pomiędzy każdym dystryktem a wszystkimi dziesięcioma środkami skupień. W konsekwencji powstaje macierz zawierająca po jednym wierszu na każdy dystrykt i po jednej kolumnie na każde skupienie. Przyjrzyjmy się trzem pierwszym wierszom i zaokrąglijmy wartości do dwóch miejsc po przecinku:

```
>>> similarities[:3].round(2)
array([[0.46 , 0., 0.8 , 0. , 0. , 0., 0. , 0.98, 0. , 0. ],
       [0., 0.96 , 0., 0.03 , 0.04 , 0. , 0., 0. , 0.11, 0.35 ],
       [0.34 , 0., 0.45 , 0. , 0., 0., 0.01 , 0.73 , 0. , 0. ]])
```

Rysunek 2.19 ukazuje środki dziesięciu skupień wyznaczonych przez algorytm centroidów. Dystrykty są pokolorowane zgodnie z ich podobieństwem geograficznym do najbliższego środka skupienia. Zwróć uwagę, że większość skupień mieści się w gęsto zaludnionych obszarach.



Rysunek 2.19. Podobieństwo obliczone metodą gaussowskiej RBF do najbliższego środka skupienia

Potoki transformujące

Jak widać, należy przeprowadzać wiele operacji przekształcania we właściwej kolejności. Na szczęście możemy skorzystać z klasy `Pipeline`, pomagającej wyznaczyć odpowiednią sekwencję transformacji. Poniżej przedstawiamy niewielki potok stosowany wobec atrybutów numerycznych, który najpierw imputuje, a następnie skaluje cechy wejściowe:

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
      ('impute', SimpleImputer(strategy="median")),
      ('standardize', StandardScaler()),
])
```

Konstruktor `Pipeline` przyjmuje listę par nazwa/estymator (2-krotek) definiującą sekwencję operacji. Nazwy mogą być dowolne, pod warunkiem że są niepowtarzalne i nie zawierają podwójnego znaku podkreślenia (__). Przydadzą się one później podczas strojenia hiperparametrów. Wszystkie estymatory oprócz ostatniego (który może być dowolnym elementem: transformatorem, predyktorem czy jakimkolwiek innym estymatorem) muszą być funkcjami przekształcającymi (tj. muszą zawierać metodę `fit_transform()`).

Jeżeli w notatniku Jupyter wpiszesz `import sklearn` i uruchomisz `sklearn.set_config(display="diagram")`, to wszystkie estymatory *Scikit-Learn* będą wyświetlane w postaci interaktywnych diagramów. Jest to bardzo przydatne podczas wizualizowania potoków. Aby zwizualizować `num_pipeline`, uruchom komórkę z wstawioną `num_pipeline` w ostatniej linijce. Kliknięcie estymatora wyświetli dodatkowe szczegóły.

Jeżeli nie musisz nazywać transformatorów, możesz użyć wygodnej funkcji `make_pipeline()`; przyjmuje ona argumenty jako argumenty pozycyjne i tworzy obiekt `Pipeline` przy użyciu nazw klas transformatorów, małymi literami i bez podkreśleń (np. `"simpleimputer"`):

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

Jeżeli wiele transformatorów ma taką samą nazwę, na jej końcu zostaje dodany indeks (np. `"foo-1"`, `"foo-2"` itd.).

W momencie wywołania metody `fit()` następuje sekwencyjne wywołanie metody `fit_transform()` wobec wszystkich transformatorów, przekazanie wyniku każdego wywołania w postaci parametru do następnego wywołania aż do osiągnięcia ostatniego estymatora, dla którego zostaje wywołana metoda `fit()`.

Potok odsłania te same metody co ostatni estymator. W tym przykładzie ostatnią funkcją prognostyczną jest `StandardScaler`, będąca w istocie transformatorem, dlatego potok również zachowuje się jak transformator. Jeżeli wywołasz metodę `transform()` potoku, będzie ona sekwencyjnie realizować wszystkie przekształcenia wobec danych. Gdyby ostatni estymator był predyktorem, to cały potok miałby metodę `predict()`, a nie `transform()`. Jej wywołanie

spowodowałoby sekwencyjne zastosowanie wszystkich przekształceń wobec danych, a wynik zostałby przesłany do metody `predict()` predyktora.

Wywołajmy metodę `fit_transform()` potoku i sprawdźmy dwa pierwsze rzędy wyniku, zaokrąglone do dwóch miejsc po przecinku:

```
>>> housing_num_prepared = num_pipeline.fit_transform(housing_num)
>>> housing_num_prepared[:2].round(2)
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.6 , -0.7 ,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

Jak już wiesz, jeżeli chcesz odzyskać porządny obiekt `DataFrame`, możesz skorzystać z metody `get_feature_names_out()` potoku:

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared, columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index)
```

Potoki obsługują indeksowanie; na przykład `pipeline[1]` zwraca drugi estymator w potoku, a `pipeline[:-1]` zwraca obiekt `Pipeline` zawierający wszystkie estymatory oprócz ostatniego. Możesz także uzyskać dostęp do estymatorów za pomocą atrybutu `steps`, stanowiącego listę par nazwa/estymator, lub atrybutu słownikowego `named_steps`, odwzorowującego nazwy na estymatory. Na przykład `num_pipeline["simpleimputer"]` zwraca estymator o nazwie `"simpleimputer"`.

Do tej pory zajmowaliśmy się oddzielnie kolumnami kategorialnymi i numerycznymi. Byłoby wygodniej, gdybyśmy dysponowali pojedynczym transformatorem przetwarzającym wszystkie kolumny i dobierającym odpowiednie przekształcenia do poszczególnych typów kolumn. W tym celu możesz użyć klasy `ColumnTransformer`. Na przykład w poniższym listingu jest stosowana `num_pipeline` (zdefiniowana przed chwilą) do atrybutów numerycznych, a `cat_pipeline` do atrybutu kategorialnego:

```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```

Najpierw importujemy klasę `ColumnTransformer`, następnie definiujemy listy nazw kolumn numerycznych i nazw kolumn kategorialnych, po czym konstruujemy prosty potok dla atrybutów kategorialnych. Na koniec konstruujemy klasę `ColumnTransformer`. Jej konstruktor wymaga listy tripletów (3-krotek), gdzie każda krotka zawiera nazwę (niepowtarzalną i pozbawioną podwójnych znaków podkreślenia), transformator, a także listę nazw (lub indeksów) kolumn, wobec których powinien zostać użyty transformator.

Zamiast korzystać z transformatora, możesz wyznaczyć łańcuch znaków `"drop"`, jeżeli chcesz, aby kolumny zostały usunięte, lub `"passthrough"`, jeśli mają pozostać niezmienione. Domyślnie pozostałe kolumny (te, które nie zostały wymienione na liście) zostaną usunięte, ale możesz wyznaczyć hiperparametr `remainder` w dowolnym transformatorze (lub łańcuchu znaków `"passthrough"`), jeśli chcesz, aby były one przetwarzane inaczej.

Tworzenie listy nazw kolumn nie jest zbyt wygodne, dlatego moduł *Scikit-Learn* zawiera klasę `make_column_selector`, której możesz użyć do automatycznego wyboru wszystkich cech danego typu, na przykład numerycznych lub kategorialnych. Możesz przekazać selektor do `ColumnTransformer` zamiast nazw lub indeksów. Ponadto jeśli nie masz ochoty nadawać nazw transformatorom, możesz użyć metody `make_column_transformer()`, wyznaczającej samodzielnie nazwy, podobnie jak metoda `make_pipeline()`. Na przykład poniższy listing tworzy taką samą klasę `ColumnTransformer` jak wcześniej, z tym że transformatory zostają tu automatycznie nazwane `"pipeline-1"` i `"pipeline-2"` zamiast `"num"` i `"cat"`:

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)
```

Teraz możemy użyć klasy `ColumnTransformer` na zestawie danych *Housing*:

```
housing_prepared = preprocessing.fit_transform(housing)
```

Świetnie! Uzyskaliśmy potok przetwarzania wstępnego przyjmujący pełen zestaw danych i przetwarzający każdą kolumnę za pomocą odpowiednich transformatorów, a następnie łączący poziomo przekształcone kolumny (transformatory nie mogą zmieniać liczby wierszy). Po raz kolejny zwracana jest tablica *NumPy*, możesz jednak uzyskać nazwy kolumn za pomocą `preprocessing.get_feature_names_out()` i, tak jak poprzednio, umieścić dane w obiekcie `DataFrame`.

Klasa `OneHotEncoder` zwraca macierz rzadką, natomiast `num_pipeline` gęstą. W przypadku występowania takiej mieszanki macierzy rzadkich i gęstych klasa `ColumnTransformer` oszacowuje gęstość macierzy końcowej (tzn. współczynnik występowania komórek niezerowych) i zwraca macierz rzadką, jeżeli gęstość jest mniejsza od ustalonego progu (domyślnie `sparse_threshold=0.3`). W tym przykładzie zostaje zwrócona macierz gęsta.

Twój projekt zmierza w dobrym kierunku i już jesteś niemal gotowy, aby zacząć trenować modele! Chcesz teraz stworzyć pojedynczy potok, który będzie realizował wszystkie omówione do tej pory przekształcenia. Przypomnijmy sobie, co i dlaczego będzie robił potok:

- Brakujące wartości w cechach numerycznych będą wstawiane poprzez zastępowanie ich medianą, gdyż większość algorytmów uczenia maszynowego nie spodziewa się brakujących danych. W cechach kategorialnych brakujące wartości będą zastępowane najczęściej występującą kategorią.

- Cecha kategorialna będzie kodowana „gorącojedynkowo”, gdyż większość algorytmów uczenia maszynowego akceptuje na wejściu wyłącznie dane numeryczne.
- Zostanie obliczonych i dołączonych kilka cech współczynnikowych: `współczynnik_sypialni` (`bedrooms_ratio`), `pokoje_na_rodzinę` (`room_per_house`) i `liczba_osób_na_dom` (`people_per_house`). Mamy nadzieję, że będą lepiej skorelowane z medianą wartości domu, a więc pomogą modelom uczenia maszynowego.
- Zostanie dodanych także kilka cech podobieństwa skupień. Prawdopodobnie okażą się one bardziej przydatne od współrzędnych geograficznych.
- Cechy długoogonowe zostaną zastąpione ich logarytmami, gdyż większość modeli preferuje cechy o rozkładach mniej więcej jednostajnym lub Gaussa.
- Wszystkie cechy numeryczne będą standaryzowane, gdyż większość algorytmów uczenia maszynowego preferuje cechy o mniej więcej takiej samej skali.

Kod potoku realizującego powyższe założenia powinien wyglądać już znajomo:

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_name(function_transformer, feature_names_in):
    return ["ratio"]  # nazwy cech

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy="median"),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler())

log_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(np.log, feature_names_out="one-to-one"),
    StandardScaler())
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                                          StandardScaler())
preprocessing = ColumnTransformer([
        ("współczynnik_sypialni", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
        ("pokoje_na_rodzinę", ratio_pipeline(), ["total_rooms", "households"]),
        ("liczba_osób_na_dom", ratio_pipeline(), ["population", "households"]),
        ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
                               "households", "median_income"]),
        ("geo", cluster_simil, ["latitude", "longitude"]),
        ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
    ],
    remainder=default_num_pipeline)  # pozostaje jedna kolumna: housing_median_age
```

Jeżeli uruchomisz tę klasę `ColumnTransformer`, wykona ona wszystkie przekształcenia i wyświetli tablicę *NumPy* zawierającą 24 cechy:

```
>>> housing_prepared = preprocessing.fit_transform(housing)
>>> housing_prepared.shape
(16512, 24)
>>> preprocessing.get_feature_names_out()
```

```
array(['współczynnik_sypialni__ratio', 'pokoje_na_rodzinę__ratio',
'liczba_osób_na_dom__ratio',
       'log__total_bedrooms', 'log__total_rooms', 'log__population',
'log__households', 'log__median_income',
       'geo__Podobieństwo 0 skupienia', 'geo__Podobieństwo 1 skupienia',
'geo__Podobieństwo 2 skupienia',
       'geo__Podobieństwo 3 skupienia', 'geo__Podobieństwo 4 skupienia',
'geo__Podobieństwo 5 skupienia',
       'geo__Podobieństwo 6 skupienia', 'geo__Podobieństwo 7 skupienia',
'geo__Podobieństwo 8 skupienia',
       'geo__Podobieństwo 9 skupienia', 'cat__ocean_proximity_<1H OCEAN',
'cat__ocean_proximity_INLAND',
       'cat__ocean_proximity_ISLAND', 'cat__ocean_proximity_NEAR BAY',
'cat__ocean_proximity_NEAR OCEAN',
       'remainder__housing_median_age'], dtype=object)
```

Wybierz i wytrenuj model

Nareszcie! Określiliśmy ramy problemu, zdobyliśmy i przeanalizowaliśmy dane, przetestowaliśmy zestawy uczący i testowy, a także napisaliśmy potok wstępnego przetwarzania danych, automatycznie oczyszczającego i przygotowującego dane pod algorytmy uczenia maszynowego. Jesteśmy gotowi, aby wybrać i wytrenować model uczenia maszynowego.

Trenuj i oceń model za pomocą zbioru uczącego

Mam dobre wieści do przekazania: dzięki wszystkim wcześniejszym czynnościom dalsze etapy będą łatwe! Postanawiasz na początek wytrenować bardzo prosty model regresji liniowej:

```
from sklearn.linear_model import LinearRegression

lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)
```

I już! Masz teraz do dyspozycji działający model regresji liniowej. Wypróbowujesz go na zbiorze uczącym, sprawdzając pięć pierwszych przewidywań i porównując je z etykietami:

```
>>> housing_predictions = lin_reg.predict(housing)
>>> housing_predictions[:5].round(-2)  # –2 = zaokrąglone do najbliższych setek
array([246000., 372700., 135700., 91400., 330900.])
>>> housing_labels.iloc[:5].values
array([458300., 483800., 101700., 96100., 361800.])
```

Model działa, ale nie w każdym przypadku: pierwsza prognoza jest mylna (o ponad 200 000 dolarów!), natomiast pozostałe są nieco lepsze: dwie odstają o ok. 25%, a pozostałe dwie o niecałe 10%. Pamiętaj, że wybrałeś RMSE jako wskaźnik skuteczności, dlatego chcesz zmierzyć błąd RMSE tego modelu regresji dla całego zbioru uczącego za pomocą funkcji `root_mean_squared_error()`:

```
>>> from sklearn.metrics import root_mean_squared_error
>>> lin_rmse = root_mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse
68972.88910758484
```

W tym przypadku nie używamy metody `score()`, ponieważ zwraca ona **współczynnik determinacji R^2** zamiast RMSE. Współczynnik R^2 reprezentuje stosunek wariancji w danych, którą model jest w stanie wyjaśnić: im bliżej 1 (która jest wartością maksymalną), tym lepiej. Jeśli model po prostu przewiduje średnią przez cały czas, nie wyjaśnia żadnej części wariancji, jego wynik R^2 wynosi 0. A jeśli model radzi sobie jeszcze gorzej, jego wynik R^2 może być ujemny i w rzeczywistości dowolnie niski.

Lepsze to niż nic, ale wynik ten nie jest powalający: wartości atrybutu `median_housing_values` dla większości dystryktów mieszczą się w zakresie pomiędzy 120 000 a 265 000 dolarów, dlatego standardowy błąd predykcji rzędu 68 628 dolarów naprawdę nie jest satysfakcjonujący. Widzimy tu klasyczny przykład niedotrenowania modelu wobec danych uczących. Taka sytuacja oznacza, że cechy nie dostarczają odpowiedniej ilości informacji pozwalających na uzyskanie dobrych prognoz albo że model nie jest wystarczająco dobry. Jak wiemy z poprzedniego rozdziału, podstawowymi sposobami radzenia sobie z problemem niedotrenowania są wybór potężniejszego algorytmu, wprowadzenie lepszych cech lub zmniejszenie ograniczeń modelu. Nasz model nie jest regularyzowany, dlatego odpada ostatnia możliwość. Możemy spróbować dodać więcej cech, najpierw jednak chcesz sprawdzić bardziej skomplikowany model i zobaczyć, jak sobie poradzi.

Postanawiasz wypróbować model `DecisionTreeRegressor`, gdyż jest to całkiem zaawansowany model potrafiący wyszukiwać w danych skomplikowane, nieliniowe zależności (w rozdziale 5. zajmiemy się szczegółowo tematem drzew decyzyjnych):

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = make_pipeline(preprocessing, DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)
```

Po wyuczeniu modelu sprawdzasz jego wydajność wobec zbioru uczącego:

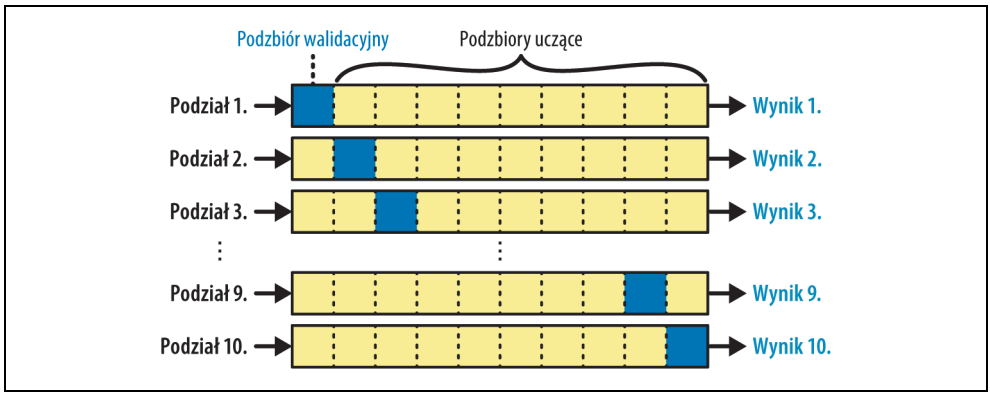
```
>>> housing_predictions = tree_reg.predict(housing)
>>> tree_rmse = root_mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse
0.0
```

Że jak? Żadnego błędu? Czyżby ten model był stuprocentowo bezbłędny? Oczywiście jest o wiele bardziej prawdopodobne, że model po prostu został znacznie przetrenowany. Skąd mamy mieć pewność? Wiesz już, że nie chcemy używać zbioru uczącego, dopóki nie będziemy w pełni przekonani o skuteczności wybranego modelu, dlatego musimy wykorzystać część zbioru uczącego do trenowania, a część do ocenienia modelu.

Dokładniejsze ocenianie za pomocą sprawdzianu krzyżowego

Jednym ze sposobów oceniania modelu drzewa decyzyjnego byłoby wykorzystanie funkcji `train_test_split()` do rozdzielenia zestawu uczącego na podzbiory trenujący i walidacyjny, następnie wyuczenie naszego modelu przy użyciu tego zbioru trenującego, po czym ewaluacja za pomocą zbioru walidacyjnego. Proces ten oznacza dodatkowy wysiłek, ale nie jest zbyt skomplikowany i całkiem dobrze spełnia swoje zadanie.

Doskonałą alternatywą jest użycie funkcji *k*-krotnego **sprawdzianu krzyżowego** (**kroswalidacji,** ang. *k-fold cross-validation*). W tym podejściu zbiór uczący zostaje podzielony na *k* nienakładających się **podzbiorów** (ang. *folds*). Następnie przeprowadzane jest *k*-krotne trenowanie i ocenianie modelu, w którym za każdym razem inny podzbiór służy do oceny, a pozostałe *k-1* podzbiorów do treningu. W rezultacie otrzymujemy *k* wyników ewaluacji (rysunek 2.20).



Rysunek 2.20. k-krotny sprawdzian krzyżowy, gdzie k = 10

Scikit-Learn zawiera wygodną funkcję `cross_val_score()`, która robi dokładnie to, czego potrzebujemy, i zwraca tablicę zawierającą *k* wyników ewaluacji. Na przykład użyjmy jej do oceny naszego regresora drzewiastego, stosując *k = 10*:

```
from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                              scoring="neg_root_mean_squared_error", cv=10)
```

Funkcja sprawdzianu krzyżowego będąca częścią modułu *Scikit-Learn* oczekuje funkcji użyteczności (im większa wartość, tym lepiej), a nie funkcji kosztu (im mniejsza wartość, tym lepiej), dlatego funkcja zliczająca wynik stanowi w rzeczywistości przeciwieństwo błędu RMSE. Ma ujemną wartość, dlatego musisz zmienić znak rezultatu, aby otrzymać wyniki RMSE.

Spójrzmy na wyniki:

```
>>> pd.Series(tree_rmses).describe()
count       10.000000
mean     66573.734600
std       1103.402323
min      64607.896046
25%      66204.731788
50%      66388.272499
75%      66826.257468
max      68532.210664
dtype: float64
```

Model drzewa decyzyjnego nie wygląda teraz już tak dobrze. W rzeczywistości wygląda na to, że sprawuje się niemal tak kiepsko jak model regresji liniowej! Zwróć uwagę, że za pomocą

sprawdzianu krzyżowego możemy nie tylko oszacować wydajność naszego modelu, lecz także zmierzyć precyzję oszacowań (tj. odchylenie standardowe). RMSE drzewa decyzyjnego wynosi w przybliżeniu 66 574 z odchyleniem standardowym rzędu ok. 1103. Nie uzyskalibyśmy takich informacji, gdybyśmy skorzystali wyłącznie z jednego zbioru walidacyjnego. Jednak ceną kroswalidacji jest konieczność kilkukrotnego uczenia modelu, dlatego rozwiązanie to nie zawsze jest możliwe.

Jeżeli obliczysz ten sam wskaźnik dla modelu regresji liniowej, okaże się, że średni RMSE wynosi 70 003, a odchylenie standardowe to 4182. Model drzewa decyzyjnego zdaje się zatem radzić sobie nieznacznie lepiej od modelu liniowego, ale różnica jest minimalna z powodu sporego przetrenowania. Wiemy, że występuje tu problem z przetrenowaniem, ponieważ błąd uczenia jest niewielki (a w zasadzie równy 0), natomiast błąd walidacji jest duży.

Sprawdźmy jeszcze jeden model: `RandomForestRegressor`. Jak się przekonasz w rozdziale 6., mechanizm działania modeli losowego lasu polega na uczeniu wielu drzew decyzyjnych za pomocą różnych podzbiorów cech, po czym następuje uśrednienie otrzymanych prognoz. Takie modele składające się z wielu innych modeli nazywane są **zespołami** (ang. *ensemble*): jeśli modele bazowe są bardzo zróżnicowane, ich błędy nie będą ze sobą silnie skorelowane, a zatem uśrednienie predykcji wygładzi błędy, zmniejszy przetrenowanie i poprawi ogólną wydajność. Kod jest w dużej mierze taki sam jak poprzednio:

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
                                          RandomForestRegressor(random_state=42))
forest_rmses = -cross_val_score(forest_reg, housing, housing_labels,
                                          scoring="neg_root_mean_squared_error", cv=10)
```

Spójrzmy na wyniki:

```
>>> pd.Series(forest_rmses).describe()
count           10.000000
mean      47038.092799
std            1021.491757
min          45495.976649
25%         46510.418013
50%        47118.719249
75%        47480.519175
max         49140.832210
dtype: float64
```

O proszę, teraz jest znacznie lepiej: model losowego lasu prezentuje się bardzo obiecująco! Jeśli jednak wytrenujesz `RandomForestRegressor` i zmierzysz RMSE dla zbioru uczącego, uzyskasz wynik mniej więcej 17 551: jest on znacznie mniejszy, co oznacza, że wciąż występuje tu spore przetrenowanie. Możliwymi rozwiązaniami są uproszczenie modelu, jego ograniczenie (np. regularyzacja) lub pozyskanie znacznie większej liczby danych uczących. Zanim jednak wybierzesz model losowego lasu, powinieneś wypróbować wiele innych modeli, reprezentujących różne rodzaje algorytmów uczenia maszynowego (np. kilka maszyn wektorów nośnych przy użyciu różnych jąder i być może sieć neuronową), bez spędzania dużej ilości czasu na strojeniu hiperparametrów. Naszym celem jest stworzenie krótkiej listy (od dwóch do pięciu pozycji) najbardziej obiecujących modeli.

Wyreguluj swój model

Załóżmy, że masz już sporządzoną listę obiecujących modeli. Musisz je teraz dostroić. Możemy tego dokonać na kilka sposobów.

Metoda przeszukiwania siatki

Jednym z rozwiązań jest własnoręczne dobieranie wartości hiperparametrów, dopóki nie uzyskasz ich znakomitej kombinacji. Jest to bardzo żmudne zajęcie i być może nie masz tyle czasu, aby sprawdzić wszystkie możliwości.

Zamiast tego możesz zlecić poszukiwania obiektowi `GridSearchCV`. Wystarczy podać interesujące nas hiperparametry oraz ich proponowane wartości, a wszystkie kombinacje zostaną ocenione za pomocą sprawdzianu krzyżowego. Na przykład poniższy kod poszukuje najlepszej kombinacji wartości hiperparametrów dla modelu `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
    ("preprocessing", preprocessing),
    ("random_forest", RandomForestRegressor(random_state=42)),
])
param_grid = [
    {'preprocessing__geo__n_clusters': [5, 8, 10],
     'random_forest__max_features': [4, 6, 8]},
    {'preprocessing__geo__n_clusters': [10, 15],
     'random_forest__max_features': [6, 8, 10]},
]
grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
                                          scoring='neg_root_mean_squared_error')
grid_search.fit(housing, housing_labels)
```

Zwróć uwagę, że możesz się odnosić do dowolnego hiperparametru dowolnego estymatora w potoku, nawet jeśli estymator ten jest zagnieżdżony głęboko wewnątrz kilku potoków i transformatorów kolumnowych. Na przykład gdy moduł *Scikit-Learn* widzi `"preprocessing__ ↪geo__n_clusters"`, to podzieli ten łańcuch znaków w miejscach wyznaczonych przez podwójne znaki podkreślenia, a następnie poszukuje estymatora o nazwie `"preprocessing"` w potoku oraz znajduje klasę wstępnego przetwarzania `ColumnTransformer`. Następnie poszukuje transformatora o nazwie `"geo"` wewnątrz tej klasy `ColumnTransformer` i znajduje transformator `ClusterSimilarity`, wykorzystany przez nas w przypadku atrybutów współrzędnych geograficznych. Teraz znajduje hiperparametr `n_clusters` tego transformatora. Na takiej samej zasadzie `random_forest__max_features` dotyczy hiperparametru `max_features` estymatora o nazwie `"random_forest"`, będącego oczywiście modelem `RandomForestRegressor` (hiperparametr `max_features` zostanie wyjaśniony w rozdziale 6.).

Umieszczenie etapów wstępnego przetwarzania danych w potoku *Scikit-Learn* umożliwia strojenie hiperparametrów wstępnego przetwarzania wraz z hiperparametrami modelu. Jest to dobre zjawisko, ponieważ obydwa typy hiperparametrów często oddziałują ze sobą. Na przykład zwiększenie wartości `n_clusters`

może również wymagać zwiększenia wartości `max_features`. Jeżeli dopasowanie transformatorów potoku jest kosztowne obliczeniowo, możesz wyznaczyć ścieżkę do katalogu pamięci podręcznej w parametrze `memory`: po pierwszym dopasowaniu potoku biblioteka *Scikit-Learn* zapisze w tym katalogu dopasowane transformatory. Jeżeli dopasujesz ponownie potok z tymi samymi hiperparametrami, moduł *Scikit-Learn* wczyta po prostu transformatory przechowywane w pamięci podręcznej.

W parametrze `param_grid` występują dwa słowniki, dlatego `GridSearchCV` najpierw obliczy wszystkie 3·3 = 9 kombinacji wartości hiperparametrów `n_clusters` i `max_features` określonych w pierwszym obiekcie `dict`, następnie zaś wypróbuje 2·3 = 6 kombinacji wartości parametrów z drugiego obiektu `dict`. Łącznie mechanizm przeszukiwania siatki sprawdzi zatem 9+6 = 15 kombinacji wartości hiperparametrów, a każdy potok zostanie wytrenowany trzykrotnie na każdą kombinację, gdyż korzystamy z trzykrotnego sprawdzianu krzyżowego. Oznacza to, że będzie łącznie 15·3 = 45 rund uczenia! Może to zająć trochę czasu, ale ostatecznie możesz uzyskać najlepszą kombinację hiperparametrów, na przykład taką:

```
>>> grid_search.best_params_
{'preprocessing__geo__n_clusters': 15, 'random_forest__max_features': 6}
```

W tym przykładzie najlepszy model zostanie uzyskany po wyznaczeniu wartości 15. parametru `n_clusters` i wartości 6. parametru `max_features`.

Wartość 15 jest największa spośród obliczonych dla `n_clusters`, dlatego prawdopodobnie warto byłoby przeprowadzić kolejne przeszukiwanie siatki, tym razem z podanymi większymi wartościami — możemy dzięki temu uzyskać jeszcze lepsze wyniki.

Możesz również bezpośrednio uzyskać dostęp do najlepszego estymatora za pomocą `grid_search.best_estimator_`. Jeżeli obiekt `GridSearchCV` zostanie zainicjowany z parametrem `refit=True` (jest to wartość domyślna), to po znalezieniu najlepszego estymatora za pomocą sprawdzianu krzyżowego ta funkcja oszacowująca będzie wykorzystana wobec całego zbioru uczącego. Zazwyczaj jest to dobre rozwiązanie, ponieważ im więcej danych zostanie użytych, tym bardziej wzrośnie wydajność modelu.

Wyniki ewaluacji są dostępne za pomocą `grid_search.cv_results_`. Jest to słownik, ale jeśli umieścisz go w obiekcie `DataFrame`, otrzymasz listę wszystkich wyników testowych dla każdej kombinacji hiperparametrów i dla każdego podziału sprawdzianu krzyżowego, jak również średni wynik testu ze wszystkich podziałów:

```
>>> cv_res = pd.DataFrame(grid_search.cv_results_)
>>> cv_res.sort_values(by="mean_test_score", ascending=False, inplace=True)
>>> […] # zmienia nazwy kolumn tak, aby zmieściły się na stronie, a także pokazuje rmse = −wynik
>>> cv_res.head() # uwaga: pierwsza kolumna jest identyfikatorem rzędu
    n_clusters  max_features  split0  split1  split2  mean_test_rmse
12          15             6   42725   43708   44335           43590
13          15             8   43486   43820   44900           44069
6           10             4   43798   44036   44961           44265
9           10             6   43710   44163   44967           44280
7           10             6   43710   44163   44967           44280
```

Średni wynik testowy RMSE dla najlepszego modelu wynosi 43 590, co stanowi lepszy wynik niż w przypadku standardowych wartości tych hiperparametrów (47 038). Gratulacje! Właśnie skutecznie dostroiliśmy nasz najlepszy model!

Metoda losowego przeszukiwania

Mechanizm przeszukiwania siatki przydaje się wtedy, gdy chcemy sprawdzić względnie niewielką liczbę kombinacji (tak jak w powyższym przykładzie), często jednak jest preferowana klasa `RandomizedSearchCV`, zwłaszcza w przypadku dużej przestrzeni przeszukiwania hiperparametrów. Klasa ta jest używana w bardzo podobny sposób jak `GridSearchCV`, tutaj jednak nie są sprawdzane wszystkie możliwe kombinacje, lecz następuje ewaluacja stałej liczby losowych kombinacji poprzez dobór losowej wartości hiperparametru w każdym przebiegu. Może to być zaskakujące, ale takie rozwiązanie cechuje kilka zalet:

- Jeżeli niektóre hiperparametry są ciągłe (lub dyskretne, ale z dużą liczbą możliwych wartości) i zezwolisz na, dajmy na to, 1000 przebiegów losowego przeszukiwania, zostanie sprawdzonych 1000 wartości dla każdego hiperparametru, podczas gdy w metodzie przeszukiwania siatki zostałoby sprawdzonych jedynie kilka zdefiniowanych wartości.
- Załóżmy, że hiperparametr nie robi zbyt dużej różnicy, ale jeszcze o tym nie wiesz. Jeśli ma 10 możliwych wartości i dodasz je do przeszukiwania siatki, to uczenie zajmie dziesięciokrotnie więcej czasu. Jeśli jednak dodasz je do przeszukiwania losowego, nie będzie to miało wpływu na czas uczenia.
- Jeżeli istnieje sześć hiperparametrów do sprawdzenia, każdy zawierający 10 możliwych wartości, to w metodzie przeszukiwania siatki model musi być wytrenowany milion razy, natomiast przeszukiwanie losowe można zawsze uruchomić z dowolnie wybraną liczbą iteracji.

W przypadku każdego hiperparametru musisz wprowadzić listę dostępnych wartości lub określić rozkład prawdopodobieństwa:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {'preprocessing__geo__n_clusters': randint(low=3, high=50),
                  'random_forest__max_features': randint(low=2, high=20)}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_distribs, n_iter=10, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
```

Biblioteka *Scikit-Learn* zawiera również klasy przeszukiwania hiperparametrów `Halving` ↪`RandomSearchCV` i `HalvingGridSearchCV`. Ich celem jest skuteczniejsze wykorzystanie zasobów obliczeniowych w kontekście szybszego trenowania lub eksplorowania większej przestrzeni hiperparametrów. Mechanizm ich działania jest następujący: w pierwszej rundzie wiele kombinacji hiperparametrów (zwanych „kandydatami") jest generowanych albo metodą przeszukiwania siatki, albo metodą przeszukiwania losowego. Te kandydaty są następnie

używane do trenowania modeli ocenianych jak zwykle za pomocą sprawdzianu krzyżowego. Jednak w uczeniu wykorzystywane są ograniczone zasoby, co znacznie przyspiesza pierwszą rundę. Domyślnie „ograniczone zasoby" oznaczają, że modele są uczone na małym fragmencie zbioru uczącego. Są jednak możliwe również inne ograniczenia, takie jak zmniejszenie liczby iteracji uczenia, jeżeli model ma odpowiedzialny za nią hiperparametr. Po sprawdzeniu wszystkich kandydatów tylko najlepsze z nich przejdą do następnej rundy, gdzie zostają sprawdzone na większej puli zasobów. Po kilku rundach zostają wyłonione ostateczne kandydaty za pomocą wszystkich zasobów. W ten sposób możesz zaoszczędzić nieco czasu w trakcie strojenia hiperparametrów.

Metody zespołowe

Innym sposobem strojenia modelu jest próba połączenia najlepiej spisujących się modeli. Grupa (lub „zespół") zawsze ma większą wydajność od jej elementów składowych (tak jak losowe lasy są skuteczniejsze od składających się na nie pojedynczych drzew decyzyjnych), zwłaszcza jeśli poszczególne modele popełniają odmienne rodzaje błędów. Możesz na przykład wytrenować i dostroić model *k*-najbliższych sąsiadów, a następnie stworzyć model zespołowy przewidujący jedynie średnią z predykcji lasu losowego, stanowiącą predykcję całego modelu. Przyjrzymy się temu zagadnieniu dokładniej w rozdziale 6.

Analizowanie najlepszych modeli i ich błędów

Często uzyskasz wiele informacji na temat problemu, sprawdzając najlepsze modele. Na przykład model `RandomForestRegressor` może wskazywać względną istotność każdego atrybutu w generowaniu dokładnych prognoz:

```
>>> final_model = rnd_search.best_estimator_  # w tym wstępne przetwarzanie danych
>>> feature_importances = final_model["random_forest"].feature_importances_
>>> feature_importances.round(2)
array([0.07, 0.05, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, [...], 0., 0.01])
```

Uszeregujmy teraz te wyniki istotności w kolejności malejącej i wyświetlmy je obok odpowiednich nazw atrybutów:

```
>>> sorted(zip(feature_importances,
...                     final_model["preprocessing"].get_feature_names_out()),
...                     reverse=True)
…
[(np.float64(0.18599734460509476), 'log__median_income'),
(np.float64(0.07338850855844489), 'cat__ocean_proximity_INLAND'),
(np.float64(0.06556941990883976), 'współczynnik_sypialni__ratio'),
(np.float64(0.053648710076725316), 'pokoje_na_rodzinę__ratio'),
(np.float64(0.04598870861894749), 'liczba_osób_na_dom__ratio'),
(np.float64(0.04175269214442519), 'geo__Podobieństwo 30 skupienia'),
(np.float64(0.02597697232869678), 'geo__Podobieństwo 25 skupienia'),
(np.float64(0.023595895886342255), 'geo__Podobieństwo 36 skupienia'),
[…]
(np.float64(0.0004325970342247361), 'cat__ocean_proximity_NEAR BAY'),
(np.float64(3.0190221102670295e-05), 'cat__ocean_proximity_ISLAND')]
```

Dzięki uzyskanym informacjom możesz zrezygnować z niektórych mniej przydatnych cech (przykładowo wygląda na to, że tylko kategoria `ocean_proximity` jest użyteczna, dlatego warto spróbować usunąć pozostałe).

Transformator `sklearn.feature_selection.SelectFromModel` może automatycznie odrzucić najmniej użyteczne cechy: gdy dopasowujesz go, trenuje model (zazwyczaj metodą lasu losowego), wyszukuje jego atrybut `feature_importances_` i wybiera najprzydatniejsze cechy. Następnie po wywołaniu metody `transform()` zostają porzucone pozostałe cechy.

Przyjrzyj się również błędom popełnianym przez Twój system, a następnie postaraj się poznać ich przyczynę oraz znaleźć rozwiązanie: dodać cechy lub usunąć nieprzydatne, pozbyć się elementów odstających itd.

Teraz jest również dobra okazja, aby sprawdzić **bezstronność modelu** (ang. *model fairness*): powinien działać poprawnie nie tylko ogólnie, lecz również dla różnych kategorii dystryktów, wiejskich lub miejskich, ubogich lub bogatych, północnych lub południowych, zamieszkanych przez mniejszości itd. Wymaga to szczegółowej **analizy stronniczości** (ang. *bias analysis*): utworzenia podzbiorów zbioru walidacyjnego dla każdej kategorii i przeanalizowania wydajności modelu na nich. To sporo pracy, ale istotnej: jeżeli Twój model nie radzi sobie z całą kategorią dystryktów, to prawdopodobnie nie należy go wdrażać aż do rozwiązania problemu, a przynajmniej nie należy używać do uzyskiwania przewidywań dla tej kategorii, gdyż może to przynieść więcej szkody niż pożytku.

Oceń system za pomocą zbioru testowego

Po etapie strojenia modeli w końcu uzyskasz system sprawujący się wystarczająco dobrze. Jesteś gotów, aby ocenić jego wydajność za pomocą zbioru danych testowych. Proces ten nie wyróżnia się niczym szczególnym; pobierz przykłady i etykiety z zestawu testowego, a następnie uruchom `final_model` do przekształcenia tych danych i uzyskania przewidywań, które należy ocenić:

```
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

final_predictions = final_model.predict(X_test)

final_rmse = root_mean_squared_error(y_test, final_predictions)
print(final_rmse) # wyświetla wynik 41445.533268606625
```

W pewnych sytuacjach takie punktowe oszacowanie błędu uogólniania może się okazać niewystarczająco przekonujące: a jeżeli model jest zaledwie o 0,1% lepszy od obecnie używanego w środowisku produkcyjnym? Być może chcesz się dowiedzieć, jak precyzyjne jest to oszacowanie. W tym celu możesz obliczyć 95-procentowy **przedział ufności** (ang. *confidence interval*) dla błędu uogólniania za pomocą funkcji `scipy.stats.bootstrap()`. Uzyskujesz dość duży przedział w zakresie od 39 521 do 43 702, a Twoje wcześniejsze oszacowanie punktowe o wartości 41 445 znajduje się mniej więcej pośrodku tego przedziału:

```
from scipy import stats

def rmse(squared_errors):
    return np.sqrt(np.mean(squared_errors))


confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
boot_result = stats.bootstrap([squared_errors], rmse,
                              confidence_level=confidence, random_state=42)
rmse_lower, rmse_upper = boot_result.confidence_interval
```

W przypadku intensywnego strojenia hiperparametrów będziemy zazwyczaj otrzymywać nieco gorsze wyniki niż zmierzone za pomocą sprawdzianu krzyżowego. Wynika to z faktu, że system zostaje dostrojony do danych walidacyjnych i prawdopodobnie nie będzie sobie tak dobrze radził z nieznanymi próbkami. Akurat tak się nie dzieje w naszym przykładzie, ponieważ testowy RMSE jest mniejszy od walidacyjnego RMSE, ale jeśli Ci się to przytrafi, musisz powstrzymać chęć dalszego poprawiania hiperparametrów w celu uzyskania dobrych wyników dla zbioru testowego; wszelkie usprawnienia już raczej nie poprawią procesu uogólniania wobec nowych danych.

Teraz następuje faza przeduruchomieniowa projektu. Efektywna prezentacja rozwiązania wyróżnia świetnych danetyków spośród dobrych. Powinieneś tworzyć zwięzłe raporty (w formacie Markdown lub jako PDF czy slajdy), wizualizować kluczowe spostrzeżenia (np. używając *Matplotlib* lub innych narzędzi, choćby SeaBorn czy Tableau) i dostosowywać przekaz do odbiorców: techniczny dla współpracowników, ogólny dla interesariuszy. Przedstawiaj znaczące i łatwe do zapamiętania stwierdzenia (np. „mediana dochodów jest najważniejszym predyktorem cen mieszkań"). Podkreślaj, czego się nauczyłeś, co zadziałało, a co nie, jakie założenia zostały przyjęte i jakie są ograniczenia Twojego systemu.

Twoje wyniki powinny być odtwarzalne (w miarę możliwości): udostępnij kod swojemu zespołowi (np. przez GitHub) i dodaj ustrukturyzowany plik *README*, który przeprowadzi osobę techniczną przez proces instalacji. Dostarcz przejrzyste notatniki (np. Jupyter) z kodem, objaśnieniami i wynikami — kod powinien być czysty i dobrze skomentowany. Zdefiniuj plik *requirements.txt* (wymagania) lub *environment.yml* (środowisko) zawierający wszystkie wymagane biblioteki wraz z ich dokładnymi wersjami (lub utwórz obraz Docker). Ustaw ziarna dla generatorów losowych i usuń wszelkie inne źródła zmienności.

W przykładzie z zestawem danych *Housing* ostateczna wydajność systemu okazuje się niewiele lepsza od oszacowań ekspertów, które często były nawet o 30% inne od rzeczywistych cen, mimo to jednak warto uruchomić taki model choćby po to, aby zaoszczędzić czas ekspertów — mogą się oni wtedy zająć ciekawszymi lub bardziej produktywnymi zadaniami.

Uruchom, monitoruj i utrzymuj swój system

Znakomicie! Otrzymaliśmy pozwolenie na uruchomienie systemu! Musisz teraz przygotować nasze rozwiązanie do warunków produkcyjnych (tzn. dopieścić kod, napisać dokumentację oraz testy itd.). Następnym etapem jest wdrożenie modelu do środowiska produkcyjnego.

Najprostszym sposobem jest po prostu zapisanie najlepszego wytrenowanego modelu, przesłanie pliku do środowiska produkcyjnego i jego wczytanie. Do zapisania modelu możesz użyć biblioteki *joblib* w następujący sposób:

```
import joblib

joblib.dump(final_model, "my_california_housing_model.pkl")
```

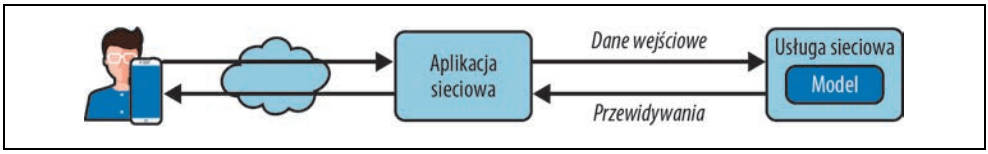
Warto często zapisywać każdy model, z którym eksperymentujesz, dzięki czemu będziesz w stanie szybko powracać do dowolnego modelu. Możesz także zapisywać wyniki sprawdzianu krzyżowego i być może również prognoz uzyskanych dla zbioru walidacyjnego. W ten sposób możesz z łatwością porównywać wyniki poszczególnych rodzajów modeli i popełniane przez nie błędy.

Po przeniesieniu modelu do środowiska produkcyjnego możesz go wczytać i użyć. W tym celu musisz najpierw zaimportować wszelkie niestandardowe klasy i funkcje używane przez model (co oznacza przeniesienie kodu do środowiska produkcyjnego), a następnie wczytać model za pomocą biblioteki *joblib* i uzyskać przy jego użyciu przewidywania:

```
import joblib
[…]  # importuje klasy KMeans, BaseEstimator, TransformerMixin, rbf_kernel itd.

def column_ratio(X):[…]
def ratio_name(function_transformer, feature_names_in): [...]
class ClusterSimilarity(BaseEstimator, TransformerMixin): [...]

final_model_reloaded = joblib.load("my_california_housing_model.pkl")

new_data = […]  # nowe dystrykty, dla których mają być uzyskiwane przewidywania
predictions = final_model_reloaded.predict(new_data)
```

Na przykład być może dany model będzie wykorzystywany na stronie internetowej: użytkownik wpisze jakieś dane dotyczące nowego dystryktu, a następnie kliknie przycisk *Oszacuj cenę*. Do serwera sieciowego zostanie wysłane zapytanie zawierające dostarczone dane, skąd trafi do aplikacji sieciowej, a na koniec kod wywoła metodę `predict()` (model powinien zostać wczytany po uruchomieniu serwera; nie chcesz, żeby to się działo za każdym razem, gdy model zostaje użyty). Ewentualnie możesz umieścić model w wyspecjalizowanej usłudze sieciowej, do której aplikacja sieciowa będzie przesyłać zapytania poprzez REST API[14] (rysunek 2.21). W ten sposób ułatwiamy proces aktualizowania modelu do nowych wersji bez konieczności przerywania działania głównej aplikacji. Jednocześnie upraszczamy skalowanie, ponieważ możemy uruchamiać dowolną liczbę usług sieciowych oraz równoważyć obciążenie zapytań przesyłanych przez aplikację sieciową. Ponadto aplikacja sieciowa może umożliwiać korzystanie z innych języków programowania, nie tylko Pythona.

[14] Mówiąc krótko, REST (lub RESTful) API to bazujący na protokole HTTP interfejs API zgodny z pewnymi konwencjami, takimi jak np. wykorzystywanie czasowników HTTP do odczytywania, aktualizowania, tworzenia lub usuwania zasobów (odpowiednio `GET`, `POST`, `PUT` i `DELETE`) oraz definiowanie danych wejściowych i wyjściowych w standardzie JSON.



Rysunek 2.21. Model wdrożony jako usługa sieciowa i wykorzystywany przez aplikację sieciową

Kolejną popularną strategią jest wdrożenie modelu w chmurze, na przykład serwisie Google Vertex AI (wcześniej Google Cloud AI Platform i Google Cloud ML Engine): wystarczy zapisać swój model za pomocą modułu *joblib* i przesłać go do Google Cloud Storage (GCS), następnie przejść do Vertex AI i stworzyć nową wersję modelu, wyznaczając plik umieszczony w magazynie GCS. I to wszystko! Uzyskujesz dostęp do prostej usługi sieciowej, która przejmuje zadania równoważenia obciążenia i skalowania. Przyjmuje zapytania JSON zawierające dane wejściowe (np. dotyczące dystryktu) i zwraca odpowiedzi JSON przechowujące predykcje. Możesz następnie użyć takiej usługi sieciowej na swojej własnej stronie internetowej (lub w dowolnym innym środowisku produkcyjnym).

Jednak nasza praca nie kończy się na wdrażaniu. Musisz także przygotować kod odpowiedzialny za monitorowanie systemu, który będzie sprawdzał działanie w regularnych odstępach czasu i wysyłał powiadomienia w przypadku pojawienia się problemu. Problem może się pojawić bardzo szybko w przypadku na przykład awarii jakiegoś składnika infrastruktury, lecz równie dobrze może się ujawniać bardzo powoli, niezauważony przez dłuższy czas. Jest to dość powszechny problem związany z dryfem danych: jeżeli model został wyuczony za pomocą zeszłorocznych danych, może nie być dostosowany do dzisiejszych.

Musisz więc monitorować bieżącą wydajność modelu. Jak to jednak robić? To zależy. W niektórych przypadkach skuteczność modelu można wywnioskować bezpośrednio ze wskaźników cyklu pracy. Na przykład jeżeli Twój model stanowi część systemu rekomendacji i sugeruje produkty, które mogą zainteresować użytkowników, to z łatwością można monitorować liczbę sprzedanych zalecanych produktów. Jeżeli liczba ta zmaleje (w stosunku do produktów niezalecanych), to model staje się głównym podejrzanym. Być może potok danych uległ załamaniu albo należy wyuczyć model za pomocą świeżych danych (wkrótce wrócę do tego tematu).

Być może jednak nie obędzie się bez analizy skuteczności modelu przez człowieka. Załóżmy na przykład, że wyuczyliśmy model klasyfikujący obrazy (zajmiemy się nim w rozdziale 3.) do wykrywania różnych usterek produktów na linii montażowej. W jaki sposób zostaniesz zaalarmowany o spadku skuteczności modelu, zanim tysiące wadliwych produktów trafi do klientów? Jednym ze sposobów jest przesłanie do ludzkich kontrolerów jakości części obrazów sklasyfikowanych przez model (zwłaszcza tych, przy których nie miał dużej pewności). W zależności od zadania kontrolerzy mogą być ekspertami lub laikami, na przykład pracownikami źródeł społecznościowych (takich jak Amazon Mechanical Turk). W pewnych sytuacjach mogą być oni nawet samymi użytkownikami, odpowiadającymi na przykład za pomocą ankiet lub zmodyfikowanych testów captcha[15].

[15] Captcha to test sprawdzający, czy użytkownik jest robotem. Testy te są często używane jako tani sposób oznaczania danych uczących.

Tak czy inaczej, musisz ustanowić system monitorujący (uwzględniający ludzkich kontrolerów kontrolujących pracujący model, ale niekoniecznie), a także wszelkie istotne procesy definiujące zachowanie w przypadku wystąpienia awarii i sposoby przygotowania się do nich. Niestety zadanie to jest pracochłonne. Tak naprawdę często wymaga poświęcenia więcej czasu niż w przypadku stworzenia i wytrenowania modelu.

Jeżeli dane ciągle ewoluują, musisz zaktualizować swoje zestawy danych i regularnie trenować model. W miarę możliwości zautomatyzuj cały proces. Oto kilka elementów, które można zautomatyzować:

- Regularne gromadzenie i oznaczanie świeżych danych (np. za pomocą ludzkich kontrolerów).
- Stworzenie skryptu automatycznie uczącego model i dostrajającego hiperparametry. Skrypt ten może być uruchamiany automatycznie, na przykład codziennie albo co tydzień, w zależności od zapotrzebowania.
- Napisanie kolejnego skryptu, który ocenia wydajność zarówno nowego, jak i starego modelu za pomocą zaktualizowanego zestawu testowego, a także wdraża model do środowiska produkcyjnego, jeżeli jego skuteczność nie zmalała (jeżeli zmalała, to musisz znaleźć przyczynę). Skrypt powinien prawdopodobnie sprawdzać skuteczność modelu za pomocą różnych podzbiorów zbioru testowego, na przykład dystryktów biednych lub bogatych, wiejskich lub miejskich itd.

Musisz także ocenić jakość danych wejściowych. Czasami wydajność nieznacznie spada z powodu słabej jakości sygnału (np. pochodzącego z uszkodzonego czujnika przesyłającego losowe wartości lub niewielkiej dynamiki sygnałów wysyłanych przez zespół znajdujący się na wcześniejszym etapie potoku), jednak może minąć trochę czasu, zanim jakość systemu spadnie na tyle, żeby zostało wysłane powiadomienie. Jeśli będziesz obserwować dane wejściowe, możesz wykryć problem nieco wcześniej. Możesz przykładowo wprowadzić alert ostrzegający przed coraz większą liczbą danych wejściowych, w których albo brakuje jakiejś cechy, albo średnia lub odchylenie standardowe zbytnio odbiegają od zestawu testowego, albo w cesze kategorialnej zaczynają się pojawiać nowe kategorie.

Na koniec nie zapomnij o przygotowaniu kopii zapasowych każdego modelu, a także o wprowadzeniu procesu i narzędzi umożliwiających szybkie przywrócenie modelu z kopii zapasowej, na wypadek gdyby nowy model z jakiejś przyczyny zaczął się zachowywać nieprawidłowo. Dzięki kopiom zapasowym możesz również porównywać modele nowe z wcześniejszymi. To samo dotyczy kopii zapasowych zestawów danych, gdyż dzięki temu można cofnąć się do wcześniejszego zestawu danych, gdyby nowy zestaw został uszkodzony (np. gdyby w dodawanych świeżych danych występowało zbyt dużo elementów odstających). Za pomocą kopii zestawów danych możesz także oceniać każdy model przy użyciu dowolnej wersji zestawu danych.

Jak widać, na uczenie maszynowe składa się całkiem rozbudowana infrastruktura. Jest to bardzo szerokie zagadnienie zwane **operacjami uczenia maszynowego** (ang. *ML Operations* — MLOps), zasługujące na oddzielną książkę. Dlatego nie zdziw się, jeżeli stworzenie i wdrożenie Twojego pierwszego projekt UM będzie wymagało mnóstwa czasu i wysiłku. Na szczęście po przygotowaniu infrastruktury przejście od pomysłu do produkcji stanie się znacznie szybsze.

Teraz Twoja kolej!

Mam nadzieję, że dzięki niniejszemu rozdziałowi wiesz już mniej więcej, jak wygląda typowy projekt uczenia maszynowego, oraz że przydadzą Ci się omówione narzędzia, za pomocą których jesteś w stanie stworzyć wspaniały system. Jak widać, lwią część czasu pochłaniają etapy przygotowywania danych, tworzenia narzędzi monitorujących, konfigurowania potoków oceniania wydajności oraz automatyzowania regularnego uczenia modelu. Oczywiście algorytmy uczenia maszynowego są istotne, jednak bardziej się opłaca zaznajomić się z całym procesem przygotowywania projektu i dobrze opanować trzy lub cztery algorytmy, niż poświęcać cały swój czas na poznawanie zaawansowanych algorytmów.

Jeśli więc jeszcze tego nie zrobiłeś, teraz nadeszła pora na włączenie laptopa, wybranie interesującego Cię zbioru danych i przebrnięcie przez cały omawiany proces od początku do końca. Warto rozpocząć swoją przygodę od serwisu takiego jak Kaggle (*https://www.kaggle.com/*). Znajdziesz tam odpowiedni zbiór danych dla siebie, wyraźny cel oraz osoby, z którymi możesz się dzielić doświadczeniami. Baw się dobrze!

Ćwiczenia

Poniższe ćwiczenia bazują na omówionym w tym rozdziale zestawie danych *Housing*:

1. Wypróbuj regresor maszyny wektorów nośnych (`sklearn.svm.SVR`) przy użyciu różnych hiperparametrów, takich jak `kernel="linear"` (oraz różnych wartości hiperparametru `C`) lub `kernel="rbf"` (oraz różnych wartości hiperparametrów `C` i `gamma`). Zwróć uwagę, że maszyny wektorów nośnych nie skalują się dobrze do dużych zestawów danych, dlatego powinieneś prawdopodobnie wytrenować model na zaledwie pierwszych 5000 przykładów zbioru uczącego i przeprowadzić jedynie trzykrotny sprawdzian krzyżowy, gdyż w przeciwnym razie cały proces będzie trwał godzinami. Na razie nie przejmuj się tym, że nie wiesz, do czego te hiperparametry służą; zostały omówione w internetowym rozdziale dostępnym pod adresem *https://ftp.helion.pl/przyklady/umaszs.zip*. Jak się spisuje najlepszy predyktor maszyny wektorów nośnych?
2. Spróbuj zastąpić klasę `GridSearchCV` obiektem `RandomizedSearchCV`.
3. Spróbuj dodać w potoku przygotowawczym klasę `SelectFromModel` w taki sposób, aby były dobierane wyłącznie najistotniejsze atrybuty.
4. Spróbuj stworzyć niestandardowy transformator trenujący regresor *k*-najbliższych sąsiadów (`sklearn.neighbors.KNeighborsRegressor`) w jego metodzie `fit()`, umieszczający przewidywania w metodzie `transform()`. Regresor KNN powinien używać tylko szerokości i długości geograficznej jako danych wejściowych i przewidywać medianę dochodów. Następnie dodaj ten nowy transformator do potoku przetwarzania wstępnego. Spowoduje to dodanie cechy reprezentującej wygładzoną medianę dochodów w pobliskich dzielnicach.
5. Sprawdź automatycznie niektóre funkcje przygotowawcze za pomocą klasy `RandomizedSearchCV`.

6. Spróbuj zaimplementować ponownie od podstaw klasę `StandardScalerClone`, a następnie dodaj do niej obsługę metody `inverse_transform()`: uruchomienie `scaler.inverse_transform(scaler.fit_transform(X))` powinno zwrócić tablicę bardzo zbliżoną do `X`. Teraz dodaj obsługę nazw cech: wyznacz atrybut `feature_names_in` w metodzie `fit()`, jeżeli obiektem wejściowym jest `DataFrame`. Atrybut ten powinien być tablicą *NumPy* z nazwami kolumn. Na koniec zaimplementuj metodę `get_feature_names_out()`: powinna mieć jeden dodatkowy argument `input_features=None`. Po przekazaniu metoda ta powinna sprawdzić, czy długość jej jest zgodna z atrybutem `n_features_in_`, i powinna być zgodna z `feature_names_in`, jeśli atrybut ten został zdefiniowany; następnie powinien zostać zwrócony `input_features`. Jeżeli jego wartość wynosi `None`, to metoda ta powinna zwrócić albo atrybut `feature_names_in_`, jeśli został zdefiniowany, albo w przeciwnym wypadku `np.array(["x0", "x1", ...])` o długości `n_features_in_`.
7. Podejmij się zadania regresji według własnego wyboru, postępując zgodnie z procesem, którego nauczyłeś się w tym rozdziale. Na przykład możesz spróbować rozwiązać problem zestawu danych Vehicle (*https://homl.info/usedcars*), gdzie celem jest przewidywanie ceny sprzedaży używanego samochodu na podstawie jego wieku, liczby przejechanych kilometrów, marki i modelu oraz innych czynników. Innym dobrym zestawem danych do wypróbowania jest Bike Sharing (*https://homl.info/bikes*): celem jest predykcja liczby wypożyczonych rowerów w danym okresie (kolumna *cnt*) na podstawie dnia tygodnia, godziny i warunków pogodowych.

Rozwiązania tych ćwiczeń znajdziesz na końcu notatnika Jupyter zawierającego kod tego rozdziału, dostępnego pod adresem *https://ftp.helion.pl/przyklady/umaszs.zip* (po polsku) lub *https://homl.inf o/colab-p* (po angielsku).

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Y

Z

Od podstaw do transformatorów — praktyczny przewodnik po ML i AI

Uczenie maszynowe i sztuczna inteligencja to dziś najgorętsze tematy w branży IT. Od ChatGPT po autonomiczne pojazdy — technologie AI przekształcają każdą dziedzinę przemysłu. Popyt na specjalistów ML rośnie wykładniczo, a umiejętność budowania i wdrażania modeli uczenia maszynowego stała się poszukiwaną na rynku kompetencją potencjalnych pracowników. Ta książka to kompleksowy przewodnik po świecie uczenia maszynowego z wykorzystaniem najpopularniejszych narzędzi: Scikit-Learn do klasycznych algorytmów ML i PyTorch — wiodącej biblioteki do uczenia głębokiego.

Książka prowadzi czytelnika od podstaw uczenia maszynowego do zaawansowanych technik uczenia głębokiego. Część pierwsza obejmuje fundamenty ML: przygotowanie danych, wybór i trenowanie, algorytmy klasyfikacji i regresji, drzewa decyzyjne, lasy losowe, techniki redukcji wymiarowości. Druga jest poświęcona sieciom neuronowym i uczeniu głębokiemu w PyTorch: od podstawowych perceptronów, przez sieci splotowe do widzenia komputerowego, sieci rekurencyjne do przetwarzania sekwencji, aż po transformatory — architekturę stojącą za współczesnymi asystentami AI. Każdy rozdział zawiera praktyczne przykłady w postaci notatników Jupyter, gotowe do uruchominia w Google Colab.

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 978-83-289-3772-7
9 788328 937727
Cena: 199,00 zł

W książce:

- Kompletny projekt ML
- Klasyczne algorytmy
- Sieci neuronowe w PyTorch
- CNN do widzenia komputerowego i wykrywania obiektów
- Transformatory do NLP
- Uczenie przez wzmacnianie i generatywne modele AI
- Transfer learning, kwantyzacja, przyspieszanie modeli

Aurélien Géron jest konsultantem ML, wykładowcą, byłym pracownikiem Google, gdzie kierował zespołem klasyfikacji filmów YouTube. Założył kilka firm technologicznych, w tym Geron AI. Autor bestsellerowych książek o uczeniu maszynowym, które wykształciły całe pokolenie praktyków ML. Prowadził zajęcia na wielu uniwersytetach, jest uznanym ekspertem w dziedzinie AI.

Ta książka wychowała nowe pokolenie praktyków uczenia maszynowego. Została znakomicie zaktualizowana, aby uwzględnić PyTorch, i ponownie staje się najważniejszym praktycznym przewodnikiem w tej dziedzinie.

Tarun Narayanan, inżynier uczenia maszynowego w Amazon AGI