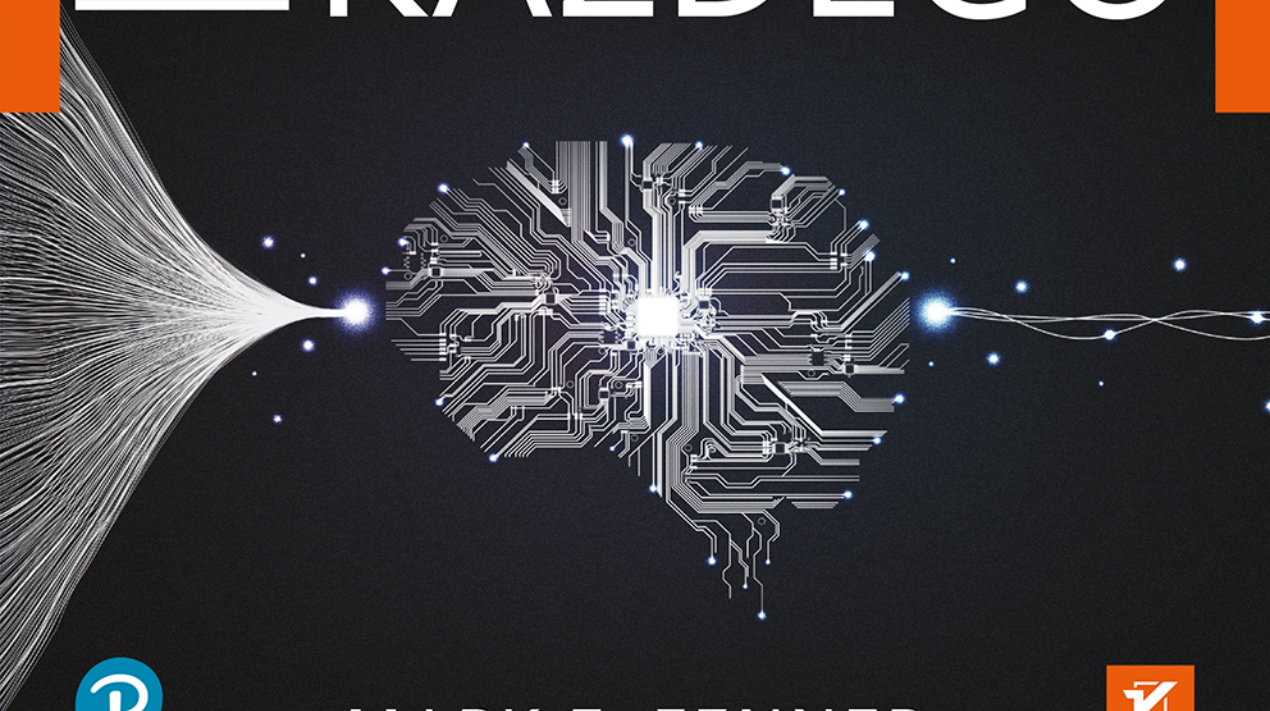
UCZENIE MASZYNOWE W PYTHONIE DLA KAŻDEGO



MARK E. FENNER

Helion

Tytuł oryginału: Machine Learning with Python for Everyone

Tłumaczenie: Jakub Hubisz (rozdz. 1 – 6), Tomasz Walczak (wprowadzenie, rozdz. 7 – 15, dodatek)

ISBN: 978-83-283-6425-7

Cover image: cono0430/Shutterstock
Pages 83, 113: Screenshot of seaborn © 2012–2018 Michael Waskom.
Pages 191, 201, 215, 221, 296, 301, 490, 503: Screenshot of seaborn heatmap © 2012–2018 Michael Waskom.
Pages 202, 208, 218, 219, 344, 345: Screenshot of seaborn swarmplot © 2012–2018 Michael Waskom.
Page 242: Screenshot of seaborn stripplot © 2012–2018 Michael Waskom.
Pages 367, 369: Screenshot of seaborn implot © 2012–2018 Michael Waskom.
Pages 368, 370: Screenshot of seaborn distplot © 2012–2018 Michael Waskom.
Page 472: Screenshot of Manifold © 2007–2018, scikit-learn developers.
Page 491: Screenshot of cluster © 2007–2018, scikit-learn developers.
Pages 494, 495, 496: Image of accordion, Vereshchagin Dmitry/Shutterstock.
Page 496: Image of fighter jet, 3dgenerator/123RF.
Page 533: Screenshot of seaborn jointplot © 2012–2018 Michael Waskom.

Authorized translation from the English language edition, entitled MACHINE LEARNING WITH PYTHON FOR EVERYONE, 1st Edition by FENNER, MARK, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2020 Pearson Education, Inc.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/umpydk.zip

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

Printed in Poland.

Spis treści

7

Ocena metod regresji

```
wejście [1]:
# Przygotowanie
from mlwpy import *
%matplotlib inline

diabetes = datasets.load_diabetes()

tts = skms.train_test_split(diabetes.data,
                            diabetes.target,
                            test_size=.25,
                            random_state=42)

(diabetes_train_ftrs, diabetes_test_ftrs,
 diabetes_train_tgt,  diabetes_test_tgt) = tts
```

Omówiłem już ocenę systemów uczących się i techniki oceny specyficzne dla klasyfikatorów. Teraz pora skupić się na ocenie metod regresji. Dla metod regresji istnieje mniej technik oceny niż dla klasyfikatorów. Nie istnieją na przykład macierze błędów lub krzywe ROC. Poznasz jednak ciekawe nowe rozwiązanie w postaci wykresów składników resztowych. Dzięki dodatkowemu miejscu część tego rozdziału przeznaczę na pomocnicze zagadnienia związane z oceną metod. Zobaczysz, jak utworzyć własne wskaźniki oceny zgodne z biblioteką `sklearn`, a także zrobisz pierwsze podejście do przetwarzania potoków. Potoki są używane, gdy systemy uczące się muszą wykonywać kilka kroków. Tu potok posłuży do *standaryzacji* danych przed próbą uczenia się na ich podstawie.

7.1. Metody regresji będące punktem odniesienia

Podobnie jak w przypadku klasyfikatorów, tak i tu potrzebna jest prosta metoda regresji będąca punktem odniesienia, z którą będzie można porównywać inne rozwiązania. Wiesz już, jak prognozować wartość *środkową* dla różnych jej definicji. Za pomocą sztuczek z biblioteki `sklearn` można łatwo tworzyć modele odniesienia, które generują predykcje w postaci średniej i mediany. Średnia i mediana są określone dla danego treningowego zbioru danych. Po procesie nauki na podstawie tego zbioru danych uzyskiwana jest jedna

wartość, która jest używana jako predykcja dla wszystkich przykładów. Można też samodzielnie wybrać arbitralne stałe. Możliwe, że posiadasz jakieś doświadczenie lub wiedzę z dziedziny i możesz zakładać, że określona wartość — na przykład minimum, maksimum lub 0,0 — jest odpowiednim punktem wyjścia. Na przykład, jeśli rzadka choroba powoduje gorączkę, a większość ludzi jest zdrowych, temperatura w pobliżu 36,6°C jest dobrą predykcją wyjściową.

Ostatnia możliwość, kwantyle, to uogólnienie mediany. Gdy matematycy mówią, że coś jest uogólnieniem, mają na myśli to, że dane zagadnienie można opisać za pomocą bardziej ogólnego szablonu. W punkcie 4.2.1 dowiedziałeś się, że mediana to środkowa wartość w posortowanych danych. Mediana ma ciekawą właściwość: połowa wartości jest od niej mniejsza, a połowa — większa. W bardziej ogólnym ujęcia mediana odpowiada określonemu percentylowi; jest to percentyl 50%. To ujęcie można zastosować nie tylko do punkt środkowego, ale też do dowolnego innego punktu. Na przykład percentyl 75% to punkt, względem którego 75% danych ma wartość mniejszą, a 25% wartość większą.

Używając kwantylów, można zastosować dowolną wartość procentową jako punkt podziału. Dlaczego używam nazw kwantyle zamiast percentyle? Ponieważ kwantyle to *dowolny* zestaw punktów podziału w przedziale od 0 do 100 o stałych odległościach. Na przykład kwartyle to wartości 25%, 50%, 75% i 100%. Percentyle to *określony* zestaw 100 wartości od 1% do 100%. Kwantyle można też podzielić bardziej precyzyjnie niż na kroki o wielkości jednego procenta — na przykład na 1000 wartości 0,1%, 0,2%, …, 1,0%, …, 99,8%, 99,9%, 100,0%.

wejście [2]:
```
baseline = dummy.DummyRegressor(strategy='median')
```

wejście [3]:
```
strategies = ['constant', 'quantile', 'mean', 'median', ]
baseline_args = [{"strategy":s} for s in strategies]

# Dodatkowe argumenty dla stałej wartości i kwantylów
baseline_args[0]['constant'] = 50.0
baseline_args[1]['quantile'] = 0.75

# Podobnie jak w rozdziale 5., ale z użyciem wyrażenia listowego;
# przetwarzanie jednego argumentu (słownika)
def do_one(**args):
    baseline = dummy.DummyRegressor(**args)
    baseline.fit(diabetes_train_ftrs, diabetes_train_tgt)
    base_preds = baseline.predict(diabetes_test_ftrs)
    return metrics.mean_squared_error(base_preds, diabetes_test_tgt)

# Pobieranie wszystkich wyników za pomocą wyrażenia listowego
mses = [do_one(**bla) for bla in baseline_args]

display(pd.DataFrame({'mse':mses}, index=strategies))
```

	mse
constant	14,657.6847
quantile	10,216.3874
mean	5,607.1979
median	5,542.2252

7.2. Dodatkowe miary w metodach regresji

Do tej pory jako miarę sukcesu (lub, co może jest bardziej precyzyjne, miarę błędu) w regresji stosowaliśmy *błąd średniokwadratowy* (MSE). Ponadto zmodyfikowaliśmy błąd średniokwadratowy, tworząc *pierwiastek błędu średniokwadratowego* (RMSE), ponieważ skala błędu średniokwadratowego jest niedopasowana do predykcji. MSE jest mierzony na tej samej skali, co kwadraty błędów, dlatego RMSE pozwala wrócić do skali używanej dla błędów. Tę konwersję do tej pory stosowaliśmy w „improwizowany" sposób, wyciągając w różnych miejscach pierwiastek kwadratowy. Jednak miara RMSE jest całkiem popularna. Dlatego zamiast cały czas obliczać ją samodzielnie za pomocą ręcznie pisanego kodu, warto zintegrować ją z naszym zestawem sztuczek opartych na bibliotece `sklearn`.

7.2.1. Tworzenie własnych miar oceny

Biblioteka `sklearn` na ogólnym poziomie korzysta z wyników liczbowych, przy czym wyższe wartości są lepsze. Dlatego możemy opracować nową miarę do oceny metod regresji. Wymaga to trzech kroków: zdefiniowania miary błędów, użycia błędu do zdefiniowania oceny i wykorzystania oceny w funkcji oceniającej. Po zdefiniowaniu funkcji oceniającej można przekazać ją za pomocą parametru `scoring` do funkcji `cross_val_score`. Pamiętaj, że: (1) w przypadku *błędów* i *funkcji straty* niższe wartości są lepsze, a (2) w przypadku *funkcji oceniających* lepsze są wyższe wyniki. Dlatego potrzebna jest odwrotna zależność między miarą błędów a wynikiem. Jednym z najłatwiejszych rozwiązań jest negacja błędów. Utrudnia to jednak myślenie. Wyniki oparte na RMSE będą ujemne, a *lepszym* wynikiem z dwóch będzie ten bliższy zeru (choć nadal ujemny). Możliwe, że początkowo Ci się to nie spodoba. Zauważ, że jest to trochę podobne jak utrata *mniejszej* kwoty pieniędzy. Jeśli i tak musisz je stracić, najlepiej będzie stracić zero.

Pora przejść do szczegółów implementacji. Funkcje do obliczania błędów i funkcje oceniające muszą przyjmować trzy argumenty: dopasowany model, predyktory i wartość docelową. To prawda, nazwy argumentów w kodzie są dość dziwne. W `sklearn` stosowana jest konwencja, zgodnie z którą nazwy argumentów typu „im mniej, tym lepiej" kończą się członem `_error` lub `_loss`, a nazwy argumentów typu „im więcej, tym lepiej" kończą się członem `_score`. Obiekt `*_scorer` odpowiada za zastosowanie modelu do cech, generowanie predykcji i porównywanie ich z rzeczywistymi znanymi wartościami. Do ilościowej oceny

skuteczności modelu używane są funkcje do obliczania błędów lub funkcje oceniające. Nie trzeba definiować wszystkich trzech opisanych argumentów. Możesz wybrać, które elementy do obliczania RMSE zaimplementujesz. Jednak napisanie kodu wszystkich trzech elementów pozwala zobaczyć, jak są one ze sobą powiązane.

```
wejście [4]:
def rms_error(actual, predicted):
    ' Funkcja do obliczania RMSE '
    # Mniejsze wartości są lepsze (a < b oznacza, że a jest lepsze)
    mse = metrics.mean_squared_error(actual, predicted)
    return np.sqrt(mse)

def neg_rmse_score(actual, predicted):
    ' Funkcja do obliczania wyniku oparta na RMSE '
    # Wyższe wartości są lepsze (a < b oznacza, że b jest lepsze)
    return -rms_error(actual, predicted)

def neg_rmse_scorer(mod, ftrs, tgt_actual):
    ' Oparta na RMSE funkcja oceniająca odpowiednia jako argument scoring '
    tgt_pred = mod.predict(ftrs)
    return neg_rmse_score(tgt_actual, tgt_pred)

knn = neighbors.KNeighborsRegressor(n_neighbors=3)
skms.cross_val_score(knn,
                     diabetes.data, diabetes.target,
                     cv=skms.KFold(5, shuffle=True),
                     scoring=neg_rmse_scorer)

wyjście [4]:
array([-58.0034, -64.9886, -63.1431, -61.8124, -57.6243])
```

Funkcja `hand_and_till_M_statistic` z punktu 6.4.1 zwracała ocenę modelu, a za pomocą wywołania `make_scorer` przekształciliśmy ją w funkcję oceniającą. Tu przedstawiłem wszystkie wymagane przez bibliotekę `sklearn` komponenty do obliczania RMSE: miarę błędów, wynik i funkcję oceniającą. W wywołaniu `make_scorer` można nakazać, by wyższe wartości były traktowane jako *lepsze*. Służy do tego argument `greater_is_better`.

7.2.2. Inne wbudowane miary regresji

W poprzednim rozdziale zapoznałeś się z długą listą miar. Przypominam, że są one dostępne za pomocą wywołania `metrics.SCORERS.keys()`. Domyślną miarę stosowaną dla regresji liniowej można sprawdzić za pomocą wywołania `help(lr.score)`.

```
wejście [5]:
lr = linear_model.LinearRegression()

# help(lr.score) # Włącza pełne dane wyjściowe
print(lr.score.__doc__.splitlines()[0])

Returns the coefficient of determination R^2 of the prediction.
```

Miarą domyślną na potrzeby regresji liniowej jest R^2. Jest to miara domyślna dla wszystkich metod regresji. Więcej na temat R^2 piszę dalej. Inne ważne wbudowane miary do oceny skuteczności metod regresji to: średni błąd bezwzględny (ang. *mean absolute error*, MAE), MSE (którego używaliśmy) i medianowy błąd bezwzględny (ang. *median absolute error*).

Z praktycznego punktu widzenia można tu porównać błędy MAE i MSE. Zignorujmy na razie człon M, ponieważ w obu miarach dotyczy on dzielenia przez liczbę przykładów. W MAE dla dużych i niewielkich różnic względem *rzeczywistych* wartości kary są równe wielkości błędu. Błąd o 10 jednostek oznacza karę równą 10. Jednak w MSE większe błędy oznaczają wyższe kary: błąd o 10 jednostek to kara równa 100. W MAE przejście od błędu 2 do błędu 4 oznacza wzrost kary z 2 do 4. W MSE kara rośnie wtedy z 4 do 16, a ostateczny efekt jest taki, że dla wysokich błędów kary są *naprawdę duże*. Ostatnia uwaga: jeśli dwie predykcje są błędne o 5 jednostek (błędy wynoszą w obu przypadkach 5), ich sumaryczny wpływ to 5 + 5 = 10. W MSE dwa błędy równe 5 oznaczają wpływ $5^2 + 5^2 = 25 + 25 = 50$. W MAE sumaryczny błąd równy 50 możemy otrzymać jako błąd 5 jednostek dla 10 punktów danych, błąd 25 jednostek dla dwóch punktów danych lub błąd 50 jednostek dla jednego punktu danych. W MSE wystarczy błąd 7 jednostek w jednym punkcie danych (ponieważ $7^2 = 49$). Co gorsza, w MSE błąd 50 jednostek w jednym punkcie danych oznacza $50^2 = 2500$ *punktów błędu średniokwadratowego*. Chyba jesteśmy spłukani.

Medianowy błąd bezwzględny działa w nieco inny sposób. Przypomnij sobie omówienie średniej i mediany na przykładzie wag w punkcie 4.2.1. Mediana ma chronić przed nadmiernym wpływem pojedynczych wysokich błędów na inne, mniejsze pomyłki. Jeśli akceptujesz kilka naprawdę poważnych odstępstw (jeśli pozostałe predykcje są poprawne), medianowy błąd bezwzględny może okazać się przydatny.

7.2.3. R^2

Miara R^2 z natury jest statystyczna, dlatego jest też obciążona sporym bagażem wiedzy z tej dziedziny. Jednak ta pozycja nie jest książką poświęconą samej statystyce. Miara R^2 jest też koncepcyjnie powiązana z regresją liniową, jednak nie jest to książka dotycząca samej regresji liniowej. Choć zarówno statystyka, jak i regresja liniowa są zagadnieniami wchodzącymi w zakres książki na temat uczenia maszynowego, chcę się tu skupić na ogólnych kwestiach, a nie na szczegółach matematycznych (zwłaszcza że istnieją odpowiednie książki, zajęcia, wydziały i dziedziny). Dlatego mierze R^2 poświęcam tu tylko kilka stron. Dlaczego w ogóle się nią zajmuję? Ponieważ jest ona domyślnie stosowana jako miara regresji w bibliotece `sklearn`.

Czym jest miara R^2? Zdefiniuję ją na podstawie tego, jak oblicza ją biblioteka `sklearn`. Aby przejść do tego, najpierw należy omówić dwie wartości. Warto zauważyć, że nie stosuję tu nazw używanych w podręcznikach do statystyki. Zamiast tego opisuję R^2 w kategoriach dwóch modeli. Model 1. to *interesujący nas model*. Na tej podstawie można obliczyć, jak dobre wyniki daje nasz model. Wykorzystamy do tego *sumę kwadratów błędów* (SSE) z punktu 2.5.3.

$$\text{SSE}_{\text{nasz}} = \sum_i \text{nasz\_błędy}_i^2 = \sum_i (\text{nasz\_predykcje}_i - \text{rzeczywiste}_i)^2$$

Drugi model to prosty model odniesienia, który jako predykcje zawsze zwraca *średnią* wartości docelowych. Suma kwadratów błędów dla modelu zwracającego średnią to:

$$\text{SSE}_{\text{średnia}} = \sum_i \text{średnia\_błędy}_i^2 = \sum_i (\text{średnia\_predykcje}_i - \text{rzeczywiste}_i)^2 = \sum_i (\text{średnia}_i - \text{rzeczywiste}_i)^2$$

Przepraszam za zasypywanie Cię tymi wszystkimi symbolami Σ. Oto zapis w kodzie:

```
wejście [6]:
our_preds  = np.array([1, 2, 3])
mean_preds = np.array([2, 2, 2])
actual     = np.array([2, 3, 4])

sse_ours = np.sum(( our_preds - actual)**2)
sse_mean = np.sum((mean_preds - actual)**2)
```

Za pomocą tych dwóch komponentów można obliczyć R^2 w podobny sposób jak w opisie dokumentacji biblioteki `sklearn`. Ściśle rzecz biorąc, obliczenia są tu nieco inne, co objaśniam dalej:

```
wejście [7]:
r_2 = 1 - (sse_ours / sse_mean)
print("ręcznie obliczone r2:{:5.2f}".format(r_2))

ręcznie oblicznone r2: 0.40
```

Wzór z dokumentacji biblioteki `sklearn` to:

$$R^2 = 1 - \frac{SSE_{\text{nasz}}}{SSE_{\text{średnia}}}$$

Czym jest drugi wyraz w tym wzorze? $\frac{SSE_{\text{nasz}}}{SSE_{\text{średnia}}}$ to stosunek skuteczności naszego modelu do skuteczności prostego modelu. Dla obu modeli miarą jest suma kwadratów błędów. Używanym tu modelem odniesienia jest `dummy.DummyRegressor(strategy='mean')` z początku rozdziału. Na przykład, jeśli błędy naszego wymyślnego predyktora są równe w sumie 2500, a błędy przy prostych predykcjach średniej to 10 000, stosunek tych błędów wyniesie 1/4, a $R^2 = 1 - 1/4 = 3/4 = 0{,}75$. Ten proces to *normalizacja* (przeskalowanie) wyników naszego modelu na podstawie modelu, który zawsze prognozuje średnią wartości docelowych. Co jednak oznacza wyrażenie *jeden minus* ten stosunek?

7.2.3.1. Interpretacja R^2 w świecie uczenia maszynowego

W kontekście regresji liniowej drugi wyraz, $\frac{SSE_{\text{nasz}}}{SSE_{\text{średnia}}}$, jest równy od zera do jeden.

Jeśli w regresji liniowej używany jest tylko wyraz stały, skuteczność naszego modelu będzie identyczna jak modelu używającego średniej, a wynik wyrażenia będzie równy 1.

Z drugiej strony, jeżeli regresja liniowa nie popełnia żadnych błędów dla danych, wynik wyrażenia wyniesie 0. Model z regresją liniową dopasowywany do danych treningowych i oceniany na podstawie tych danych nigdy nie jest mniej skuteczny niż model używający średniej.

Jednak ten opis *nie zawsze* jest prawdziwy, ponieważ *nie zawsze* używany jest model liniowy. Aby się o tym przekonać, wystarczy pokazać, że nasz „wymyślny" model daje wyniki *gorsze* niż model używający średniej. Choć trudno wyobrazić sobie tak kiepskie wyniki dla danych treningowych, jest to wyobrażalne dla danych *testowych*. Jeśli utworzysz model gorszy niż model używający średniej i zastosujesz wzór na R^2 z biblioteki `sklearn`, otrzymasz wartość *ujemną*. W przypadku wartości o nazwie R^2 jest to dezorientujące. W końcu kwadraty liczb są zwykle dodatnie, prawda?

Mamy więc wzór 1 – *coś* i wygląda na to, że należy go czytać jako „100% minus coś", co powinno dać nam pozostałość ze 100%. Nie możemy jednak zastosować tego rozwiązania, ponieważ to *coś* może być dodatnie *lub* ujemne i nie wiemy, jak potraktować wartość powyżej *prawdziwego* maksimum równego 100% dla przedziału uwzględniającego wszystkie możliwości.

Jak więc wygląda sensowna interpretacja wartości R^2 z biblioteki `sklearn`? Stosunek między wynikami SSE pozwala uzyskać znormalizowaną skuteczność na podstawie standardowego modelu odniesienia. Na zapleczu SSE jest równe MSE, ale bez średniej. Co ciekawe, można wykorzystać tu obliczenia z algebry, o której myślałeś, że nigdy Ci się nie przyda. Jeśli podzielisz oba SSE ze stosunku przez n, otrzymasz:

$$R^2 = 1 - \frac{\frac{SSE_{\text{nasz}}}{n}}{\frac{SSE_{\text{średnia}}}{n}} = 1 - \frac{MSE_{\text{nasz}}}{MSE_{\text{średnia}}}$$

Widać tu, że tak naprawdę pracujemy z zakamuflowanym stosunkiem MSE. Pozbędziemy się teraz członu 1 –, aby ułatwić interpretację wyrażenia po prawej stronie:

$$R^2 = 1 - \frac{SSE_{\text{nasz}}}{SSE_{\text{średnia}}} = 1 - \frac{MSE_{\text{nasz}}}{MSE_{\text{średnia}}}$$

$$1 - R^2 = \frac{SSE_{\text{nasz}}}{SSE_{\text{średnia}}} = \frac{MSE_{\text{nasz}}}{MSE_{\text{średnia}}}$$

Zaletą tych przekształceń jest to, że możemy traktować wyrażenie 1 – R^2 (dla dowolnych modeli z obszaru uczenia maszynowego) jako MSE znormalizowane na podstawie MSE dla prostego modelu odniesienia, którego predykcje zawsze są równe średniej. Jeśli istnieją dwa modele i porównasz (za pomocą dzielenia) 1 – R^2 dla modelu numer 1 i 1 – R^2 dla modelu numer 2, otrzymasz:

$$\frac{1-R^2_{M1}}{1-R^2_{M2}} = \frac{\frac{MSE_{M1}}{MSE_{\text{średnia}}}}{\frac{MSE_{M2}}{MSE_{\text{średnia}}}} = \frac{MSE_{M1}}{MSE_{M2}}$$

Jest to po prostu stosunek MSE (lub SSE) dla dwóch modeli.

7.2.3.2. R^2 z biblioteki sklearn — niewygodne fakty

Obliczmy teraz kilka prostych wartości R^2 ręcznie i za pomocą biblioteki `sklearn`. Obliczymy `r2_score` na podstawie rzeczywistych wartości i predykcji dla zbioru testowego w prostym modelu, który jako predykcje zwraca średnie:

```
wejście [8]:
baseline = dummy.DummyRegressor(strategy='mean')

baseline.fit(diabetes_train_ftrs, diabetes_train_tgt)
base_preds = baseline.predict(diabetes_test_ftrs)

# Wartość r2 nie jest symetryczna, ponieważ do obliczenia średniej
# wartości docelowych używane są rzeczywiste wartości
base_r2_sklearn = metrics.r2_score(diabetes_test_tgt, base_preds)
print(base_r2_sklearn)

-0.014016723490579253
```

Teraz zobaczmy, jakie wartości uzyskamy za pomocą ręcznych obliczeń:

```
wejście [9]:
# Średnia z danych treningowych jest predykcją dla testowych wartości docelowych (sklearn)
base_errors    = base_preds - diabetes_test_tgt
sse_base_preds = np.dot(base_errors, base_errors)

# Średnia z danych treningowych jest predykcją dla testowych wartości docelowych (ręcznie)
train_mean_errors = np.mean(diabetes_train_tgt) - diabetes_test_tgt
sse_mean_train    = np.dot(train_mean_errors, train_mean_errors)

# Średnia z danych testowych jest predykcją dla testowych wartości docelowych (uwaga!)
test_mean_errors = np.mean(diabetes_test_tgt) - diabetes_test_tgt
sse_mean_test    = np.dot(test_mean_errors, test_mean_errors)

print("SSE(dla danych test.) wg średniej dla dan. tren. - sklearn:", sse_base_preds)
print("SSE(dla danych test.) wg średniej dla dan. tren. - ręczne :", sse_mean_train)
print("SSE(dla danych test.) wg średniej dla dan. test. - ręczne :", sse_mean_test)

SSE(dla danych test.) wg średniej dla dan. tren. - sklearn: 622398.9703179051
SSE(dla danych test.) wg średniej dla dan. tren. - ręczne : 622398.9703179051
SSE(dla danych test.) wg średniej dla dan. test. - ręczne : 613795.5675675676
```

Po co, do diabła, dodałem trzecią z tych możliwości? Obliczyłem średnią dla zbioru *testowego* i sprawdziłem błąd względem testowych wartości docelowych. Nie jest

zaskoczeniem, że ponieważ proces uczenia jest dostosowany do danych testowych, wyniki są nieco lepsze niż w pozostałych przypadkach. Zobacz, co się stanie, jeśli użyjemy wartości z uczenia dostosowanego do danych testowych jako punktu odniesienia w obliczeniach R^2:

wejście [10]:
```
1 - (sse_base_preds / sse_mean_test)
```

wyjście [10]:
```
-0.014016723490578809
```

Abrakadabra. Czy wszystko zauważyłeś? Zrobię to jeszcze raz.

wejście [11]:
```
print(base_r2_sklearn)
print(1 - (sse_base_preds / sse_mean_test))

-0.014016723490579253
-0.014016723490578809
```

W bibliotece `sklearn` w obliczeniach wartości R^2 model bazowy (używający średniej) jest tworzony na podstawie rzeczywistych wartości z etapu testów. Nie porównujemy więc skuteczności `my_model.fit(train)` i `mean_model.fit(train)`. Zamiast tego w `sklearn` porównywane są `my_model.fit(train)` i `mean_model.fit(test)`, a do oceny używane są dane testowe. Ponieważ jest to nieintuicyjne, warto przedstawić ten proces w pełnej postaci:

wejście [12]:
```
#
#  UWAGA! Nie próbujcie robić tego w domu, chłopcy i dziewczęta!
#  Dopasowywanie odbywa się z użyciem *testowego* zbioru danych, aby odzwierciedlić
#  obliczenia R^2 z biblioteki sklearn.
#
testbase = dummy.DummyRegressor(strategy='mean')
testbase.fit(diabetes_test_ftrs, diabetes_test_tgt)
testbase_preds = testbase.predict(diabetes_test_ftrs)
testbase_mse = metrics.mean_squared_error(testbase_preds,
                                          diabetes_test_tgt)

models = [neighbors.KNeighborsRegressor(n_neighbors=3),
          linear_model.LinearRegression()]
results = co.defaultdict(dict)
for m in models:
    preds = (m.fit(diabetes_train_ftrs, diabetes_train_tgt)
              .predict(diabetes_test_ftrs))

    mse = metrics.mean_squared_error(preds, diabetes_test_tgt)
    r2  = metrics.r2_score(diabetes_test_tgt, preds)
    results[get_model_name(m)]['R^2'] = r2
    results[get_model_name(m)]['MSE'] = mse

print(testbase_mse)

df = pd.DataFrame(results).T
df['Norm_MSE'] = df['MSE'] / testbase_mse
```

```
df['1-R^2'] = 1-df['R^2']
display(df)

5529.689797906013
```

	R^2	MSE	Norm_MSE	1-R^2
KNeighborsRegressor	0.3722	3,471.4194	0.6278	0.6278
LinearRegression	0.4849	2,848.2953	0.5151	0.5151

Tak więc 1 – R^2 obliczone przez bibliotekę `sklearn` odpowiada wartości MSE dla naszego modelu znormalizowanej na podstawie modelu używającego średniej *dopasowanego do danych testowych*. Jeśli *znasz* średnią testowych wartości docelowych, obliczony wynik informuje, jak skuteczny jest nasz model w porównaniu z predykcjami tej znanej średniej.

7.2.3.3. Zalecenia związane z R^2

Po całym tym opisie proponuję, aby nie używać miary R^2 — chyba że jesteś zaawansowanym użytkownikiem i uważasz, że wiesz, co robisz. Oto uzasadnienie tego zalecenia:

1. Miara R^2 związana jest z dużym bagażem naukowym i statystycznym. Gdy mówisz o R^2, ludzie mogą sądzić, że masz na myśli coś więcej niż przedstawione tu obliczenia. Jeśli w wyszukiwarce Google poszukasz informacji o R^2, używając statystycznej nazwy — *współczynnik determinacji* — znajdziesz tysiące stwierdzeń, które nie mają zastosowania w tym omówieniu. W kontekście miary R^2 z biblioteki `sklearn` każde stwierdzenie ze słowami „procent", „liniowe" lub „wyjaśnionej" należy traktować ze *skrajną* ostrożnością. Niektóre z tych stwierdzeń są prawdziwe w określonych warunkach, ale nie zawsze.
2. Istnieje wiele wzorów na R^2. W bibliotece `sklearn` używany jest jeden z nich. Te wzory są równoznaczne dla modelu liniowego z punktem przecięcia, jednak w niektórych scenariuszach *nie* są równoznaczne. Ta wieża Babel wzorów prowadzi do wątpliwości w związku z poprzednim punktem. Istnieją wzory na R^2, które w *określonych warunkach* oznaczają coś więcej niż wzór stosowany w bibliotece `sklearn`. Tu te nadmiarowe rzeczy nie są istotne.
3. Miara R^2 jest w prosty sposób powiązana z bardzo dziwną wartością: *znormalizowanym MSE obliczonym na podstawie modelu używającego średniej dopasowanego do danych testowych*. Można uniknąć tej zależności, wybierając inne miary.

Zamiast R^2 będziemy używać miar MSE lub RMSE. Jeśli *naprawdę* zależy Ci na znormalizowaniu tych miar, możesz porównać nasz model regresji z używającym średniej modelem dopasowanym do zbioru treningowego.

7.3. Wykresy składników resztowych

Zagłębiliśmy się w matematyczne gęstwiny. Pora zrobić krok wstecz i przyjrzeć się graficznym technikom oceny metod regresji. Opracujemy teraz odpowiednik macierzy błędów dla regresji.

7.3.1. Wykresy błędów

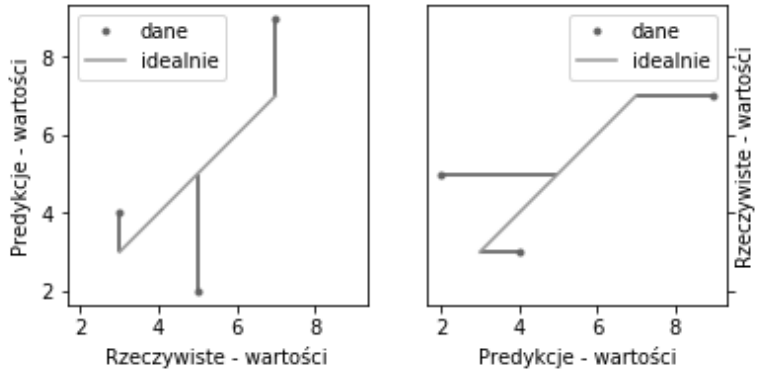
Najpierw utworzymy wykres rzeczywistych wartości docelowych i predykcji. Odległość między tymi wartościami reprezentuje wysokość błędu. Jeśli więc dany przykład miał *rzeczywistą* wartość 27,5, a *predykcja* była równa 31,5, trzeba narysować punkt (x = 27,5, y = 31,5) na osiach (rzeczywiste, predykcje). Krótka uwaga: *idealne* predykcje to punkty na linii $y = x$, ponieważ dla wszystkich danych wyjściowych prognozowane są rzeczywiste wartości. Ponieważ zwykle modele nie są idealne, można obliczyć błąd między predykcjami a rzeczywistymi wartościami. Często eliminowany jest znak wartości błędu. W tym celu wartość jest podnoszona do kwadratu lub używa się wartości bezwzględnej. Tu na razie zachowamy znak wartości błędu. Jeśli predykcje są zbyt wysokie, ta wartość jest dodatnia. Jeżeli predykcje są zbyt niskie, wartość błędu jest ujemna. Na drugim wykresie osie z predykcjami i rzeczywistymi wartościami są zamienione, dlatego wcześniejszy punkt ma postać (x = 31,5, y = 27,5): (predykcje, rzeczywiste). Jeśli wydaje Ci się to zbyt proste, czytaj dalej.

wejście [13]:
```
ape_df = pd.DataFrame({'predykcje'   : [4, 2, 9],
                       'rzeczywiste' : [3, 5, 7]})

ape_df['błąd'] = ape_df['predykcje'] - ape_df['rzeczywiste']

ape_df.index.name = 'przykład'
display(ape_df)
```

przykład	predykcje	rzeczywiste	błąd
0	4	3	1
1	2	5	-3
2	9	7	2

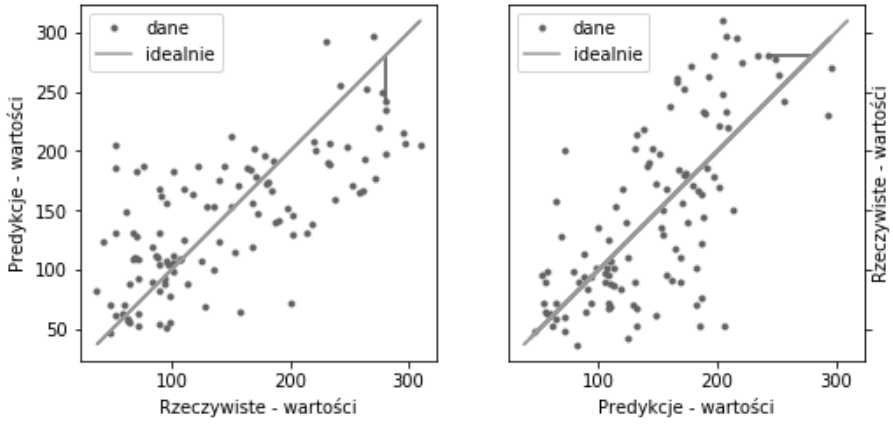
wejście [14]:
```
def regression_errors(figsize, predicted, actual, errors='all'):
    ''' figsize -> podwykresy;
        predicted i actual -> kolumny z ramki danych
        errors -> "all" lub sekwencja indeksów  '''
fig, axes = plt.subplots(1, 2, figsize=figsize,
                         sharex=True, sharey=True)
```

```
df = pd.DataFrame({'rzeczywiste':actual,
                   'predykcje':predicted})

for ax, (x,y) in zip(axes, it.permutations(['rzeczywiste',
                                            'predykcje'])):
    # Rysowanie danych jako punktów - '.'; idealny wynik to linia y = x
    ax.plot(df[x], df[y], '.', label='dane')
    ax.plot(df['rzeczywiste'], df['rzeczywiste'], '-',
            label='idealnie')
    ax.legend()

    ax.set_xlabel('{} - wartości'.format(x.capitalize()))
    ax.set_ylabel('{} - wartości'.format(y.capitalize()))
    ax.set_aspect('equal')

axes[1].yaxis.tick_right()
axes[1].yaxis.set_label_position("right")

# Wyświetlanie kresek łączących dane z idealną linią
# dla wszystkich lub tylko wybranych punktów
if errors == 'all':
    errors = range(len(df))
if errors:
    acts = df.rzeczywiste.iloc[errors]
    preds = df.predykcje.iloc[errors]
    axes[0].vlines(acts, preds, acts, 'r')
    axes[1].hlines(acts, preds, acts, 'r')

regression_errors((6, 3), ape_df.predykcje, ape_df.rzeczywiste)
```



W obu przypadkach pomarańczowa linia ($y = x$, którą tu tworzymy na podstawie wartości predykcje = rzeczywiste) reprezentuje wszechwiedzący model, w którym predykcje są równe rzeczywistym wartościom. Błąd na tej linii jest równy zeru. Na wykresie po lewej stronie różnica między predykcjami a rzeczywistymi wartościami jest mierzona w pionie. Na wykresie po prawej stronie ta różnica jest mierzona w poziomie. Zamiana osi skutkuje odbiciem punktów danych względem linii $y = x$. Dzięki cudownej technice ponownego wykorzystania kodu możemy zastosować to rozwiązanie do zbioru danych *diabetes*.

```
wejście [15]:
lr = linear_model.LinearRegression()
preds = (lr.fit(diabetes_train_ftrs, diabetes_train_tgt)
           .predict(diabetes_test_ftrs))

regression_errors((8, 4), preds, diabetes_test_tgt, errors=[-20])
```



Różnica między tymi wykresami polega na tym, że wersja po lewej stronie jest odpowiedzią na pytanie: „Jak nasz model wypada w porównaniu z rzeczywistością (*Rzeczywiste — wartości*)?”. Wykres po prawej stronie to odpowiedź na pytanie: „Jak nasz model wypadł dla danej predykcji (*Predykcje — wartości*)?”. Ta różnica jest podobna do obliczenia wrażliwości (i specyficzności) modelu na podstawie *rzeczywistych zachorowań* i obliczania precyzji modelu na podstawie *predykcji zachorowań*. Oto przykład: dla wartości rzeczywistych z przedziału od 200 do 250 większość predykcji jest zaniżona. Gdy prognozowana wartość wynosi ok. 200, rzeczywiste wartości pochodzą z przedziału od 50 do 300.

7.3.2. Wykresy składników resztowych

Teraz jesteś gotowy, aby zapoznać się z wykresami *składników resztowych*. Niestety, zderzysz się przy tym z przeszkodą w postaci problemów terminologicznych. Opisałem już *błędy* predykcji: błąd = predykcja – rzeczywista wartość. Jednak w *wykresach składników resztowych* trzeba odwrócić ten wzór: składnik resztowy = rzeczywista wartość – predykcja. Oto bardziej konkretne ujęcie:

```
wejście [16]:
ape_df = pd.DataFrame({'predykcje'   : [4, 2, 9],
                       'rzeczywiste' : [3, 5, 7]})

ape_df['błąd']      = ape_df['predykcje'] - ape_df['rzeczywiste']
ape_df['skł_reszt'] = ape_df['rzeczywiste'] - ape_df['predykcje']

ape_df.index.name = 'przykład'
display(ape_df)
```

przykład	predykcje	rzeczywiste	błąd	skł_reszt
0	4	3	1	-1
1	2	5	-3	3
2	9	7	2	-2

Przy omawianiu *błędów* można interpretować je jako wartość, o jaką predykcje były za wysokie lub za niskie. Błąd równy 2 oznacza, że predykcja była za wysoka o 2 jednostki. Możesz myśleć o błędzie jak o czymś, *co się stało*. Natomiast składniki resztowe można interpretować jako wartość, o jaką trzeba dostosować predykcję, aby ją *poprawić*. Składnik resztowy równy –2 oznacza, że trzeba odjąć 2, aby uzyskać poprawną odpowiedź.

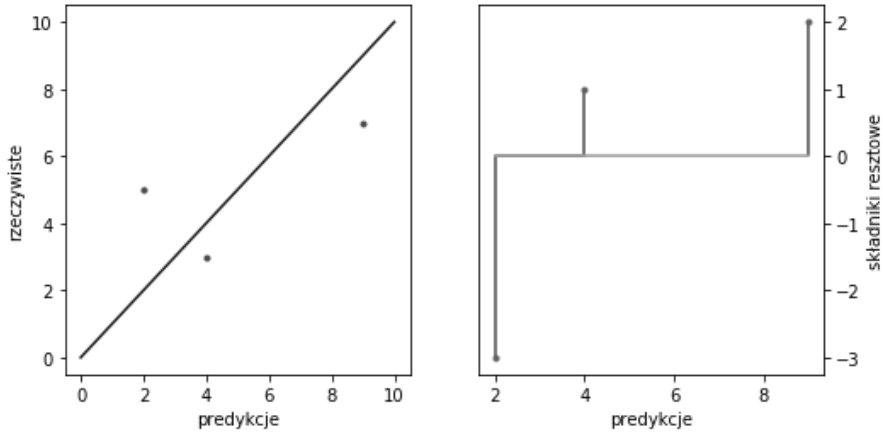
Wykresy składników resztowych powstają w wyniku narysowania predykcji i powiązanych z nimi składników resztowych. Potrzebna jest więc odmiana wykresu pokazanego wcześniej po prawej stronie (predykcji względem rzeczywistych wartości), gdzie używane są *składniki resztowe*, a nie *błędy*. Należy obliczyć składniki resztowe (odległość predykcji od rzeczywistych wartości z uwzględnieniem znaku) i wyświetlić je razem z predykcjami.

Model generuje na przykład predykcję 31,5 dla przykładu, którego rzeczywista wartość to 27,5. Składnik resztowy wynosi –4,0. Mamy więc punkt (x = predykcja = 31,5, y = składnik resztowy = –4,0). Składniki resztowe można traktować jak *pozostałości po wygenerowaniu predykcji*. Takie pozostałości czasem nazywane są *resztkami* (wyobraź sobie zieloną ektoplazmę z filmu *Pogromcy duchów*).

Teraz zobaczysz dwa wykresy: (1) rzeczywistych wartości względem predykcji i (2) predykcji względem składników resztowych.

wejście [17]:
```
def regression_residuals(ax, predicted, actual,
                         show_errors=None, right=False):
    ''' figsize -> podwykresy;
        predicted i actual -> kolumny z ramki danych
        errors -> "all" lub sekwencja indeksów '''
    df = pd.DataFrame({'rzeczywiste':actual,
                       'predykcje':predicted})
    df['błąd'] = df.rzeczywiste - df.predykcje
    ax.plot(df.predykcje, df.błąd, '.')
    ax.plot(df.predykcje, np.zeros_like(predicted), '-')

    if right:
        ax.yaxis.tick_right()
        ax.yaxis.set_label_position("right")

    ax.set_xlabel('predykcja')
    ax.set_ylabel('składnik resztowy')
```

```
    if show_errors == 'all':
        show_errors = range(len(df))
    if show_errors:
        preds = df.predykcje.iloc[show_errors]
        errors = df.błąd.iloc[show_errors]
        ax.vlines(preds, 0, errors, 'r')

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))

ax1.plot(ape_df.predykcje, ape_df.rzeczywiste, 'r.', # Predykcje względem rzeczywistych
                                                     # wartości
         [0, 10], [0, 10], 'b-')                     # Idealna linia
ax1.set_xlabel('predykcje')
ax1.set_ylabel('rzeczywiste')
regression_residuals(ax2, ape_df.predykcje, ape_df.rzeczywiste,
                     'all', right=True)
```

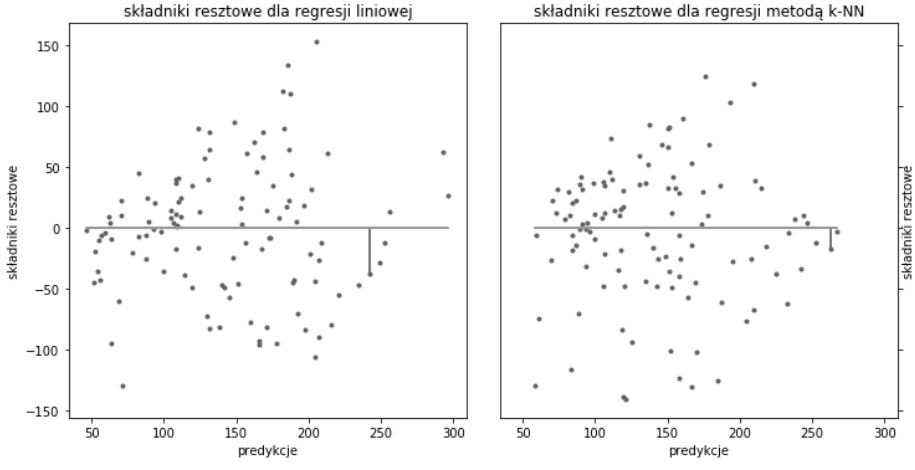


Teraz można porównać dwa różne modele na podstawie wykresów składników resztowych. Od teraz będziemy używać odpowiedniego podziału na dane treningowe i testowe, dlatego składniki resztowe można nazwać *składnikami resztowymi predykcji*. Na zajęciach ze statystyki zwykłe składniki resztowe są obliczane na podstawie treningowego zbioru danych (dla danych *z próbki*). Nie jest to typowy sposób oceny. Tu preferowana jest ocena na podstawie testowego zbioru danych.

wejście [18]:

```
lr  = linear_model.LinearRegression()
knn = neighbors.KNeighborsRegressor()

models = [lr, knn]

fig, axes = plt.subplots(1, 2, figsize=(10, 5),
                         sharex=True, sharey=True)
fig.tight_layout()
```

```
for model, ax, on_right in zip(models, axes, [False, True]):
    preds = (model.fit(diabetes_train_ftrs, diabetes_train_tgt)
                  .predict(diabetes_test_ftrs))

    regression_residuals(ax, preds, diabetes_test_tgt, [-20], on_right)

axes[0].set_title('składniki resztowe dla regresji liniowej')
axes[1].set_title('składniki resztowe dla regresji metodą k-NN');
```



Należą się tu objaśnienia. Ponieważ dwa używane modele prognozują inne wartości dla analizowanego przykładu, otrzymujemy inne punkty na poziomej osi *x*. W regresji liniowej predykcja jest odrobinę mniejsza od 250. W modelu regresji metodą *k*-NN predykcja jest nieco wyższa od 250. W obu przypadkach predykcja jest zbyt niska (pamiętaj, że składniki resztowe informują, o ile należy poprawić predykcję; tu trzeba więc dodać określone wartości do predykcji). Rzeczywista wartość to:

```
wejście [19]:
print(diabetes_test_tgt[-20])
```

```
280.0
```

Gdyby któryś z modeli zwrócił predykcję równą 280, składnik resztowy wyniósłby zero.

Na zajęciach ze statystyki wykresy składników resztowych służą do oceny, czy założenia regresji liniowej nie zostały naruszone. Ponieważ tu używamy regresji liniowej jako metody typu „czarna skrzynka”, to, czy założenia są spełnione, nie jest zbyt istotne. Jednak bardzo ważne mogą być zmiany składników resztowych względem wartości predykcji. Przedstawione wykresy pozwalają dostrzec takie zmiany. Dla najniższych predykcji w modelu z regresją liniową występuje dość stały ujemny błąd (składniki resztowe są więc dodatnie). To oznacza, że przy predykcjach niskich wartości model zapewne generuje zbyt niskie wartości. To samo dzieje się przy predykcjach wartości od około 200 do 250. W modelu

z regresją metodą *k*-NN występuje większy rozrzut wśród ujemnych błędów. Dodatnie błędy są bardziej skoncentrowane przy linii oznaczającej brak błędu. W rozdziale 10. omawiam techniki ulepszania predykcji modelu na podstawie analizy ich składników resztowych.

7.4. Pierwsze podejście do standaryzacji

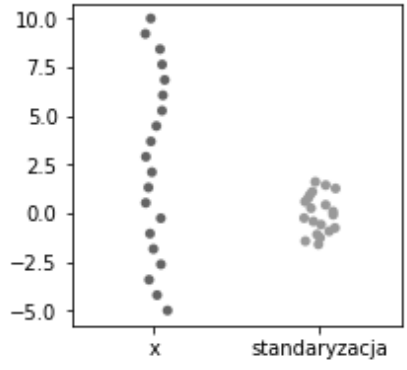
Teraz chcę przeanalizować inny zbiór danych. W tym celu muszę wprowadzić zagadnienie normalizacji. Na ogólnym poziomie normalizacja to proces umożliwiający bezpośrednie porównywanie różnych pomiarów. Często obejmuje on dwa etapy: (1) dostosowanie środka danych i (2) dostosowanie skali danych. Oto krótkie ostrzeżenie, do którego być może już się przyzwyczaiłeś: część osób używa określenia normalizacja w tym ogólnym sensie, natomiast niektórzy stosują bardziej specyficzną definicję (a jeszcze inni robią to i to jednocześnie).

Ogólne omówienie normalizacji i standaryzacji odkładam do podrozdziału 10.3. Jeśli więc interesuje Cię dokładniejsze omówienie, musisz trochę poczekać lub przeskoczyć kilka rozdziałów. Tu ważne są dwie kwestie. Po pierwsze niektóre metody *wymagają* normalizacji, aby sensowne było ich uruchamianie. Po drugie tu zastosujemy jedną z popularnych form normalizacji — *standaryzację*. W ramach *standaryzacji* danych wykonywane są dwie czynności: (1) *centrowanie* danych wokół zera i (2) skalowanie danych w taki sposób, aby odchylenie standardowe wyniosło 1. Te dwa kroki wymagają (1) odjęcia średniej od wartości i (2) podzielenia wyniku przez odchylenie standardowe. Odchylenie standardowe to bliski krewniak wariancji z punktu 5.6.1. Aby obliczyć odchylenie standardowe, należy wyciągnąć pierwiastek kwadratowy z wariancji. Przed przejściem do obliczeń chcę przedstawić standaryzację w formie graficznej.

wejście [20]:

```
# Standaryzacja danych jednowymiarowych.
# Umieszczanie wartości w ramce danych
xs = np.linspace(-5, 10, 20)
df = pd.DataFrame(xs, columns=['x'])

# Centrowanie ( - średnia) i skalowanie (/ odchylenie standardowe)
df['po_standaryzacji'] = (df.x - df.x.mean()) / df.x.std()

# Wyświetlanie pierwotnych i nowych danych oraz obliczanie statystyk
fig, ax = plt.subplots(1, 1, figsize=(3, 3))

sns.stripplot(data=df)
display(df.describe().loc[['mean', 'std']])
```

	x	standaryzacja
mean	2.5000	0.0000
std	4.6706	1.0000



Wszystko wygląda dobrze, ale zadanie staje się dużo ciekawsze dla dwóch wymiarów:

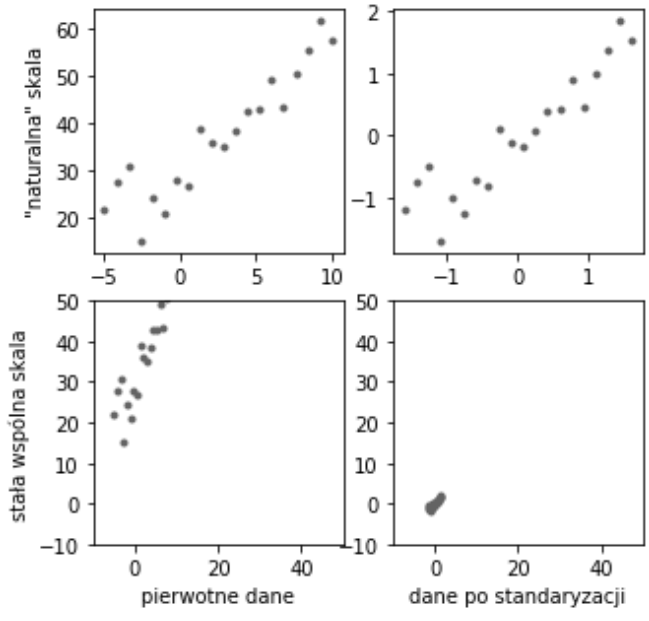
```
wejście [21]:
# Standaryzacja dla dwóch wymiarów
xs = np.linspace(-5, 10, 20)
ys = 3*xs + 2 + np.random.uniform(20, 40, 20)

df = pd.DataFrame({'x':xs, 'y':ys})
df_std_ized = (df - df.mean()) / df.std()

display(df_std_ized.describe().loc[['mean', 'std']])
```

	x	y
mean	0.0000	0.0000
std	1.0000	1.0000

Pierwotne i ustandaryzowane dane można wyświetlić na dwóch różnych skalach: naturalnej skali, którą biblioteka `matplotlib` chce zastosować do danych, i na prostej, stałej, przybliżonej skali:

```
wejście [22]:
fig, ax = plt.subplots(2, 2, figsize=(5, 5))

ax[0,0].plot(df.x, df.y, '.')
ax[0,1].plot(df_std_ized.x, df_std_ized.y, '.')
ax[0,0].set_ylabel('"naturalna" skala')

ax[1,0].plot(df.x, df.y, '.')
```

```
ax[1,1].plot(df_std_ized.x, df_std_ized.y, '.')

ax[1,0].axis([-10, 50, -10, 50])
ax[1,1].axis([-10, 50, -10, 50])

ax[1,0].set_ylabel('stała wspólna skala')
ax[1,0].set_xlabel('pierwotne dane')
ax[1,1].set_xlabel('dane po standaryzacji');
```



W tej siatce wykresów można dostrzec kilka rzeczy. Po standaryzacji kształt danych pozostaje taki sam. Widać to dobrze w górnym wierszu, gdzie dane wyświetlane są w różnych skalach i w innych miejscach. W górnym wierszu pozwoliłem bibliotece `matplotlib` używać różnych skal, aby podkreślić fakt, że *kształt* danych pozostaje taki sam. W dolnym wierszu używana jest stała wspólna skala, aby podkreślić to, że *lokalizacja* i *rozrzut* danych się zmieniają. Standaryzacja powoduje przesunięcie danych, tak że są wycentrowane w punkcie zero, a także przeskalowanie ich, tak aby wynikowe wartości miały odchylenie standardowe i wariancję równe 1,0.

W bibliotece `sklearn` standaryzację można przeprowadzić za pomocą specjalnego mechanizmu „uczącego się” o nazwie `StandardScaler`. Uczenie się ma w tym kontekście specjalne znaczenie: system uczący się oblicza średnią i odchylenie standardowe na podstawie danych treningowych i stosuje te wartości do transformacji treningowego lub testowego zbioru danych. W bibliotece `sklearn` takie narzędzia są nazywane *mechanizmami transformacji* (ang. *transformer*). Funkcja `fit` działa w nich tak samo jak w systemach uczących się poznanych do tej pory. Jednak tu zamiast wywołania `predict` używana jest funkcja `transform`.

```
wejście [23]:
train_xs, test_xs = skms.train_test_split(xs.reshape(-1, 1), test_size=.5)

scaler = skpre.StandardScaler()
scaler.fit(train_xs).transform(test_xs)

wyjście [23]:
array([[ 0.5726],
       [ 0.9197],
       [ 1.9608],
       [ 0.7462],
       [ 1.7873],
       [-0.295 ],
       [ 1.6138],
       [ 1.4403],
       [-0.1215],
       [ 1.0932]])
```

Ręczne wykonywanie podziału na dane treningowe i testowe *oraz* wielu kroków dopasowywania modelu, *a następnie* wielu kroków generowania predykcji byłoby trudne. Jeśli dodać do tego sprawdzian krzyżowy, powstałoby ćwiczenie na cierpliwość. Na szczęście biblioteka `sklearn` ułatwia tworzenie sekwencji kroków treningu i testów. Takie sekwencje są nazywane *potokami*. Oto jak wygląda standaryzacja i dopasowywanie modelu z użyciem potoku:

```
wejście [24]:
(train_xs, test_xs,
 train_ys, test_ys)= skms.train_test_split(xs.reshape(-1, 1),
                                           ys.reshape(-1, 1),
                                           test_size=.5)

scaler = skpre.StandardScaler()
lr = linear_model.LinearRegression()

std_lr_pipe = pipeline.make_pipeline(scaler, lr)

std_lr_pipe.fit(train_xs, train_ys).predict(test_xs)

wyjście [24]:
array([[17.0989],
       [29.4954],
       [41.8919],
       [36.9333],
       [61.7263],
       [24.5368],
       [31.9747],
       [49.3298],
       [51.8091],
       [59.247 ]])
```

Ten potok działa jak inne mechanizmy uczące się i udostępnia wywołania `fit` oraz `predict`. Potok można zastosować jako konfigurowalny zastępnik dla dowolnych innych

metod uczących się. Spójny interfejs systemów uczących się jest zapewne największą zaletą biblioteki `sklearn`. Możesz używać *tego samego interfejsu* niezależnie od tego, czy system uczący się jest niezależnym komponentem, czy został zbudowany z prostych komponentów. Ta spójność jest powodem, dla którego wszyscy powinniśmy fundować programistom biblioteki `sklearn` drinki na konferencjach.

Oto pewien szczegół dotyczący potoków: choć w niezależnie stosowanej klasie `StandardScaler` używane jest wywołanie `transform`, w całym potoku należy wywołać funkcję `predict`, aby przeprowadzić transformację. Oznacza to, że wywołanie `my_pipe.predict()` wykonuje transformacje niezbędne do dojścia do ostatniego kroku generowania predykcji.

Na koniec ostrzeżenie. Może zauważyłeś, że parametry używane do standaryzacji (średnia i odchylenie standardowe dla danych treningowych) uzyskiwane są w wyniku uczenia. Parametry te są obliczane dla danych *treningowych*. Podobnie jak nie chcemy „podglądać" danych testowych w kompletnych systemach uczących się, nie należy robić tego w ramach wstępnego przetwarzania danych. Wprawdzie to, na czym *dokładnie* polega podglądanie, może być trochę niejasne, dla bezpieczeństwa zachęcam, by *nigdy* nie podglądać danych testowych, chyba że: (1) masz formalny dowód na to, że nie wprowadzi to obciążenia do modelu i nie unieważni wyników, oraz (2) rozumiesz ograniczenia formalnych dowodów i wiesz, kiedy mogą one *nie* mieć zastosowania (wtedy wracamy do scenariusza, w którym nie należy podglądać). Unikaj stresu, zachowaj bezpieczeństwo i nie podglądaj.

7.5. Ocena mechanizmów regresji w bardziej zaawansowany sposób: podejście drugie

W bardziej rozbudowanym przykładzie wrócimy teraz do danych o uczniach z Portugalii. Są to te same dane, których używałeś w rozdziale 6., jednak tym razem docelowa cecha będzie wartością liczbową. Użyjemy samych liczbowych cech z pierwotnego zbioru danych, a wartości docelowe będą pochodzić z kolumny `G3`.

wejście [25]:
```
student_df = przykład.read_csv('data/portugese_student_numeric.csv')
display(student_df[['absences']].describe().T)
```

	count	mean	std	min	25%	50%	75%	max
absences	395.0000	5.7089	8.0031	0.0000	0.0000	4.0000	8.0000	75.0000

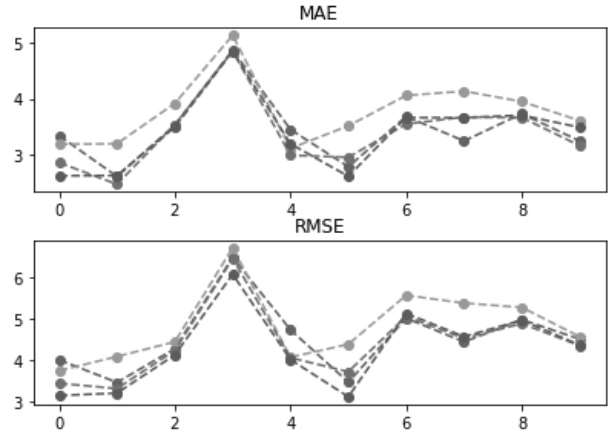
wejście [26]:
```
student_ftrs = student_df[student_df.columns[:-1]]
student_tgt  = student_df['G3']
```

7.5.1. Wyniki po sprawdzianie krzyżowym z użyciem różnych miar

W pokazanym dalej kodzie używane jest wywołanie `skms.cross_validate` do oceny wyników na podstawie wielu miar. Jest to bardzo wygodna funkcja pomocnicza. Umożliwia ocenę wielu miar za pomocą jednego wywołania. Ponadto rejestruje czas dopasowywania i generowania predykcji dla poszczególnych modeli. Te dodatkowe dane na razie pomijamy. Używane są tylko oceny zwracane na podstawie różnych miar. Ta funkcja, podobnie jak `skms.cross_val_score`, wymaga przekazania funkcji oceniającej jako argumentu `scoring`.

wejście [27]:

```
scaler = skpre.StandardScaler()

lr     = linear_model.LinearRegression()
knn_3  = neighbors.KNeighborsRegressor(n_neighbors=3)
knn_10 = neighbors.KNeighborsRegressor(n_neighbors=10)

std_lr_pipe    = pipeline.make_pipeline(scaler, lr)
std_knn3_pipe  = pipeline.make_pipeline(scaler, knn_3)
std_knn10_pipe = pipeline.make_pipeline(scaler, knn_10)

# Użycie średniej ze standaryzacją i bez standaryzacji powinno dać te same wyniki
regressors = {'baseline'  : dummy.DummyRegressor(strategy='mean'),
              'std_knn3'  : std_knn3_pipe,
              'std_knn10' : std_knn10_pipe,
              'std_lr'    : std_lr_pipe}

msrs = {'MAE'  : metrics.make_scorer(metrics.mean_absolute_error),
        'RMSE' : metrics.make_scorer(rms_error)}

fig, axes = plt.subplots(2, 1, figsize=(6, 4))
fig.tight_layout()

for mod_name, model in regressors.items():
    cv_results = skms.cross_validate(model,
                                     student_ftrs, student_tgt,
                                     scoring = msrs, cv=10)

    for ax, msr in zip(axes, msrs):
        msr_results = cv_results["test_" + msr]

        my_lbl = "{:12s} {:.3f} {:.2f}".format(mod_name,
                                               msr_results.mean(),
                                               msr_results.std())
        ax.plot(msr_results, 'o--', label=my_lbl)
        ax.set_title(msr)
        # ax.legend() # Przenieś wywołanie poza komentarz, aby wyświetlić statystyki zbiorcze
```

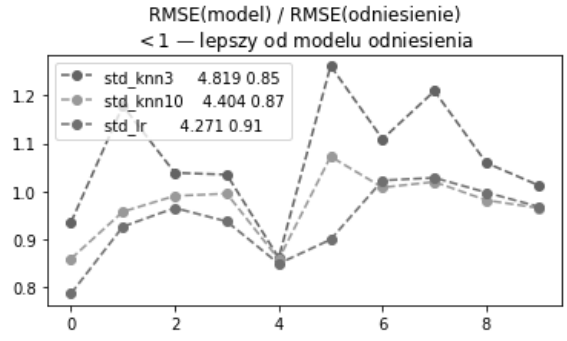


Widocznych jest tu kilka rzeczy. Metoda 3-NN nie sprawdza się w tym problemie zbyt dobrze. Metoda odniesienia cechuje się zwykle niższym błędem niż 3-NN. W kilku testach metody 10-NN i regresja liniowa dają bardzo podobne wyniki, a ich ogólna skuteczność jest zbliżona i nieco wyższa niż metody odniesienia.

Możemy zwiększyć rozbieżności między bliskimi wartościami (i ułatwić bezpośrednie porównanie z metodą regresji będącą punktem odniesienia), analizując stosunek MSE: $\frac{\sqrt{MSE_{\text{nasz}}}}{\sqrt{MSE_{\text{odniesienia}}}}$

wejście [28]:

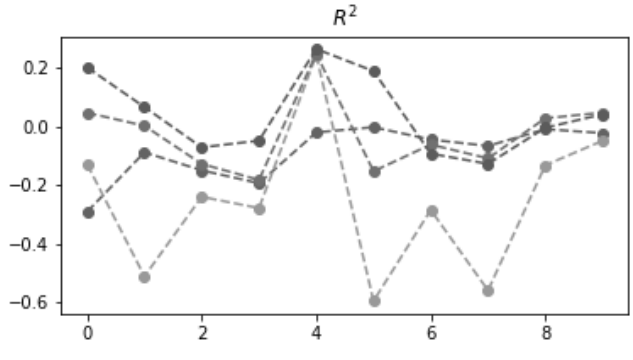
```
fig,ax = plt.subplots(1, 1, figsize=(6, 3))
baseline_results = skms.cross_val_score(regressors['baseline'],
                                        student_ftrs, student_tgt,
                                        scoring = msrs['RMSE'], cv=10)

for mod_name, model in regressors.items():
    if mod_name.startswith("std_"):
        cv_results = skms.cross_val_score(model,
                                          student_ftrs, student_tgt,
                                          scoring = msrs['RMSE'], cv=10)

        my_lbl = "{:12s} {:.3f} {:.2f}".format(mod_name,
                                               cv_results.mean(),
                                               cv_results.std())

        ax.plot(cv_results / baseline_results, 'o--', label=my_lbl)
ax.set_title("RMSE(model) / RMSE(odniesienie)\n$<1$ – lepszy od modelu odniesienia")
ax.legend();
```



Tu dobrze widać, że metoda 3-NN generuje wyższe błędy niż metoda odniesienia (stosunek błędów jest większy niż jeden) i daje gorsze wyniki niż dwie pozostałe metody regresji. Widać także, że w większości testów najskuteczniejsza jest regresja liniowa, choć metoda 10-NN okazuje się najlepsza, gdy krotność sprawdzianu krzyżowego wynosi od 6 do 9.

Choć łatwo o niewłaściwe zastosowanie tej techniki (co opisałem w punkcie 7.2.3), warto zobaczyć wyniki oparte na domyślnej mierze R^2:

```
wejście [29]:
fig, ax = plt.subplots(1, 1, figsize=(6, 3))
for mod_name, model in regressors.items():
        cv_results = skms.cross_val_score(model,
                                          student_ftrs, student_tgt,
                                          cv=10)
        my_lbl = "{:12s} {:.3f} {:.2f}".format(mod_name,
                                               cv_results.mean(),
                                               cv_results.std())

        ax.plot(cv_results, 'o--', label=my_lbl)
ax.set_title("$R^2$");
# ax.legend(); # Przenieś wywołanie poza komentarz, aby wyświetlić statystyki zbiorcze
```



Widać tu dwa ciekawe wzorce. Pierwszy dotyczy tego, że regresja liniowa daje lepsze wyniki niż regresja metodą *k*-NN. Po drugie zależność między tymi metodami wydaje się dość stała. *Kolejność* metod w każdym teście (pamiętaj, że im bardziej R^2 zbliża się do

jednego, tym metoda jest lepsza) zdaje się być identyczna. Z uwagi na opisaną wcześniej zależność między miarami R^2 i MSE nie powinno to być dla Ciebie zaskoczeniem.

Jeśli uważnie czytasz tę książkę, możesz się ponadto zastanawiać, dlaczego dla modelu odniesienia (w którym predykcje są równe średniej) R^2 nie wynosi zero. Przy okazji — zasługujesz na medal za spostrzegawczość. Czy potrafisz znaleźć odpowiedź? Wyjaśniłem już, że średnie są różne dla zbiorów treningowego i testowego. Ponieważ podział na dane treningowe i testowe odbywa się w ramach sprawdzianu krzyżowego, średnie dla danych treningowych i testowych różnią się między sobą (choć o niewielką wartość). Zauważ, że większość wartości R^2 w modelu używającym średniej jest zbliżona do zera. Jest to cena, jaką płacimy za losowość i używanie miary R^2.

7.5.2. Omówienie wyników ze sprawdzianu krzyżowego

Inne podejście do predykcji w sprawdzianie krzyżowym polega na tym, by potraktować cały proces sprawdzianu krzyżowego jak jeden system uczący się. Jeśli przejrzysz notatki, może przypomnisz sobie, że w ramach sprawdzianu krzyżowego każdy przykład jest używany *w tylko jednym* scenariuszu testowym. Dlatego można zebrać wszystkie predykcje (wygenerowane przez zestaw systemów uczących się, dla których w treningu używane są różne partycje danych) i porównać je ze znanymi wartościami docelowymi. Zastosowanie określonych miar do tych predykcji i wartości docelowych pozwala ostatecznie uzyskać *jedną* wartość dla każdego modelu i każdej miary. Uzyskać te predykcje można za pomocą wywołania `cross_val_predict`.

wejście [30]:

```
msrs = {'MAD'  : metrics.mean_absolute_error,
        'RMSE' : rms_error} # Bez funkcji oceniającej i bez modelu

results = {}
for mod_name, model in regressors.items():
    cv_preds = skms.cross_val_predict(model,
                                      student_ftrs, student_tgt,
                                      cv=10)
    for ax, msr in zip(axes, msrs):
        msr_results = msrs[msr](student_tgt, cv_preds)
        results.setdefault(msr, []).append(msr_results)
df = pd.DataFrame(results, index=regressors.keys())
df
```

wyjście [30]:

	MAD	RMSE
baseline	3.4470	4.6116
std_knn3	3.7797	4.8915
std_knn10	3.3666	4.4873
std_lr	3.3883	4.3653

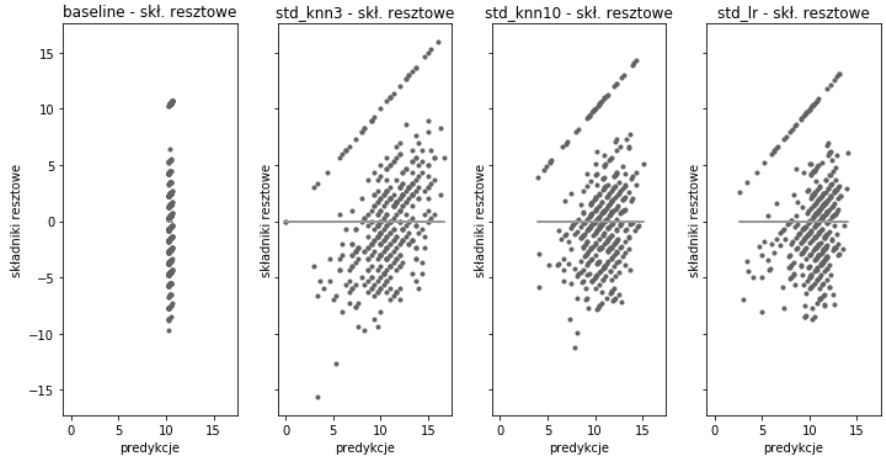
7.5.3. Składniki resztowe

Wcześniej pokazałem już podstawowe wykresy składników resztowych. Teraz pora je uatrakcyjnić, (1) przyglądając się składnikom resztowym dla modelu odniesienia i (2) używając ustandaryzowanych, wstępnie przetworzonych danych.

wejście [31]:

```
fig, axes = plt.subplots(1, 4, figsize=(10, 5),
                         sharex=True, sharey=True)
fig.tight_layout()

for model_name, ax in zip(regressors, axes):
    model = regressors[model_name]
    preds = skms.cross_val_predict(model,
                                   student_ftrs, student_tgt,
                                   cv=10)

    regression_residuals(ax, preds, student_tgt)
    ax.set_title(model_name + " - składniki resztowe")
pd.DataFrame(student_tgt).describe().T
```

wyjście [31]:



Oto kilka interesujących uwag:

- Choć modelem odniesienia jest tu model używający średniej, tych średnich jest kilka — jedna dla każdej porcji danych treningowych. Nie martw się jednak — różnice między predykcjami w modelach używających średniej są niewielkie.
- W składnikach resztowych w modelach „standaryzacja i dopasowanie” widoczne są wyraźne wzorce.

1. Na wszystkich wykresach widoczne są paski. Wynikają one z tego, że wartości docelowe to liczby całkowite. Mamy wartości docelowe 17 i 18, ale już nie 17,5. Pojawiają się więc wyraźne luki między paskami.
2. Ogólne wzorce są podobne dla wszystkich modeli oprócz modelu odniesienia.
3. Pojawia się cała grupa „wartości odstających", jeśli chodzi o błędy. Składniki resztowe w tej grupie są ujemne (tym bardziej, im wyższe są predykcje). Ujemne składniki resztowe oznaczają dodatnią wartość błędu. Wskazuje to na zbyt wysokie predykcje. Po prawej stronie predykcja wynosi 15 i jest za wysoka o około 15, ponieważ wartość rzeczywista jest bliska zeru. Po lewej stronie predykcja wynosi blisko pięć i jest za wysoka o około pięć (wartość rzeczywista ponownie jest bliska zeru). Tak więc *powodem* występowania tej grupy błędów jest to, że obrazuje ona *maksymalny* błąd (minimalne składniki resztowe) dla każdej możliwej predykcji. Jeśli predykcja jest równa x, a rzeczywista wartość to zero, błąd wynosi x (składnik resztowy to $-x$).

7.6. Koniec rozdziału

7.6.1. Podsumowanie

Dodaliśmy kilka pozycji do przybornika z narzędziami do oceny metod regresji: (1) modele regresji będące punktem odniesienia, (2) wykresy składników resztowych i (2) odpowiednie miary. Objaśniłem też trudności związane z używaniem tych miar. Ponadto po raz pierwszy zetknąłeś się z potokami i standaryzacją, których bardziej szczegółowe omówienie znajdziesz w dalszych rozdziałach.

7.6.2. Uwagi

Na wykresach składników resztowych mogłeś zauważyć, że punkty (rzeczywista wartość, predykcja) znajdują się w równej odległości od linii $y = x$ zarówno w poziomie, jak i w pionie. Taka regularność, *jeśli jest nieoczekiwana*, może wskazywać na błędy. Okazuje się jednak, że *powinniśmy* spodziewać się tej regularności.

Każdy trójkąt, który ma jeden kąt 90° i dwa kąty 45°, ma krótsze boki (przyprostokątne) o tej samej długości (ach, co za podróż we wspomnieniach do geometrii z poziomu szkoły średniej). Koncepcyjnie oznacza to, że: (1) odległość *od rzeczywistej wartości do predykcji* jest taka sama jak (2) odległość *od predykcji do rzeczywistej wartości*. Nie jest to zaskoczeniem. Odległość z Warszawy do Krakowa jest taka sama jak z Krakowa do Warszawy. Nie ma się nad czym rozwodzić, idziemy dalej.

Gdy dostrzeżesz problematyczny wzorzec na wykresie składników resztowych, możesz zadać sobie pytanie: „Co zrobić, aby poprawić wyniki?". Oto kilka zaleceń ze świata statystyki (zapoznaj się na przykład z rozdziałem 3. książki *Applied Linear Statistical*

Models Kutnera i współpracowników). Zauważ, że te wskazówki zwykle dotyczą regresji liniowej i są punktem wyjścia do wprowadzania zmian. Twoja sytuacja może wyglądać inaczej.

- Jeśli rozrzut składników resztowych jest w miarę jednorodny, należy przekształcić dane wejściowe (a nie wartości docelowe).
- Jeśli rozrzut składników resztowych rośnie (ma kształt lejka), użycie logarytmu wartości docelowych może pozwolić uzyskać jednorodny rozrzut.
- Jeśli występuje wyraźna zależność funkcyjna w składnikach resztowych (na przykład, jeśli spróbujesz utworzyć model dla równania x^2 z użyciem regresji liniowej, składniki resztowe będą miały postać wykresu funkcji $x^2 - (ax + b)$, który przypomina parabolę), pomocne może być przekształcenie danych wejściowych.

Wykonywanie tych operacji jest opisane w rozdziale 10.

Statystycy — znowu na nich natrafiamy — lubią korzystać ze studentyzowanych składników resztowych. Studentyzacja *wymaga* modelu opartego na regresji liniowej. Ponieważ nie zawsze używamy takiego modelu, można zastosować *częściowo studentyzowane* składniki resztowe, dzieląc błędy (składniki resztowe) przez RMSE: $\frac{\text{błędy}}{RMSE}$ Oznacza to normalizację wielkości błędów mniej więcej na podstawie ich średniej.

Jeśli upierasz się przy statystycznym porównywaniu algorytmów (test *t*, porównywanie średnich itd.), możesz zapoznać się z następującym artykułem:

- Dietterich Thomas G., *Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms*, „Neural Computation", 10 (7), 1998, s. 1895 – 1923.

W podrozdziale 11.3 wykorzystamy jeden z pomysłów z tego artykułu — sprawdzian krzyżowy 5×2.

W artykule z 2010 roku *What You Can and Can't Properly Do with Regression* Richard Berk opisuje trzy poziomy zastosowań regresji liniowej.

- Na pierwszym poziomie regresja liniowa jest modelem czysto opisowym. W ten sposób zwykle używamy modeli w tej książce. Mówimy: „Model liniowy najlepiej opisuje zależności między predyktorami a wartościami docelowymi". Nie przyjmujemy żadnych założeń co do sposobu powstawania danych ani co do wspomnianej zależności. Po prostu wyciągamy linijkę i sprawdzamy, jak dobrze mierzy uzyskane dane.
- Na poziomie drugim stosowane jest wnioskowanie statystyczne. Obliczane są przedziały ufności i tworzy się formalne testy hipotez statystycznych. Aby przejść na poziom drugi, dane muszą być odpowiednią losową próbką z większej populacji.
- Na poziomie trzecim można przedstawiać twierdzenia na temat *przyczynowości*: predyktory *powodują* wartości docelowe. Wynika z tego, że *na podstawie zmian wartości predyktorów możemy określić, jaka będzie zmiana wartości docelowych*. Jest to bardzo mocne stwierdzenie i *nie będziemy tu iść w tym kierunku*. Jeśli możesz

zmieniać wartości zmiennych i tworzyć przypadki testowe, aby sprawdzić wyniki (czyli przeprowadzać eksperymenty), prawdopodobnie możesz też wyeliminować mylące czynniki i pozorne korelacje.

Więcej informacji na temat R^2. Ściśle rzecz biorąc, R^2 jest w bibliotece `sklearn` domyślną miarą dla klas dziedziczących po głównej klasie regresji, `RegressorMixin`. Omawianie dziedziczenia wykracza poza zakres tej książki. Oto krótkie objaśnienie: możesz umieścić wspólne operacje w klasie *bazowej* i używać tych operacji bez ponownego ich implementowania w klasach *pochodnych*. Jest to podobne do dziedziczenia cech genetycznych przez dzieci po rodzicach.
Jeśli chcesz zapoznać się z ograniczeniami miary R^2, zacznij od następujących artykułów:

- Tarald O. Kvalseth, *Cautionary Note about R2*, „The American Statistician", 39 (4), 1985, s. 279 – 285.
- F.J. Anscombe, *Graphs in Statistical Analysis*, „American Statistician", 27 (1), 1973, s. 17 – 21.

To prawda, oba artykuły pochodzą z magazynów statystycznych. Nie, nie mam zamiaru za to przepraszać. Pierwszy tekst dotyczy przede wszystkim obliczeń R^2, a drugi — interpretowania tej miary. Wcześniej ograniczyłem krytykę miary R^2 do jej zastosowań do oceny systemów uczących się. Jednak w ogólniejszym kontekście statystycznym (1) nawet w sytuacjach, gdy używanie miary R^2 jest w pełni uzasadnione, ludzie często mylnie ją interpretują, a ponadto (2) często stosuje się ją w niewłaściwy sposób. Można powiedzieć, że ludzie biegają z nożem w ręku, narażając się na potknięcie i zrobienie sobie krzywdy. Jeśli chcesz zabezpieczyć się przed najczęstszymi przypadkami błędnego użycia tej miary, zapoznaj się z rozdziałem 2. książki *Advanced Data Analysis* Shaliziego.

Miara R^2 w formie używanej przez statystyków oznacza *znacznie więcej* niż zaprezentowane tu określone obliczenia. Dla wielu osób ze świata nauki miara R^2 jest nieodłącznie powiązana z *modelami liniowymi*, czyli z regresją liniową. Dodatkowo często przyjmowane są inne założenia. Dlatego często zetkniesz się ze zdaniami takimi jak „R^2 to procent wariancji wyjaśnionej przez relację *liniową* między predyktorami a wartościami docelowymi". To stwierdzenie jest prawdziwe, *jeśli do generowania predykcji używasz modelu liniowego*. Tu jednak interesują nas modele, które wykraczają poza typową regresję liniową.

Pobieranie danych o uczniach. Oto kod do pobierania i wstępnego przetwarzania danych używanych w ostatnim przykładzie.

```
wejście [32]:
student_url = ('https://archive.ics.uci.edu/' +
               'ml/machine-learning-databases/00320/student.zip')
def grab_student_numeric():
    # Pobieranie pliku zip i rozpakowywanie go.
    # Wypakowywanie nieznanych plików jest zagrożeniem
    import urllib.request, zipfile
    urllib.request.urlretrieve(student_url,
                               'port_student.zip')
```

```
    zipfile.ZipFile('port_student.zip').extract('student-mat.csv')

    # Wstępne przetwarzanie
    df = pd.read_csv('student-mat.csv', sep=';')

    # g1 i g2 są mocno skorelowane z g3;
    # pominięcie ich znacznie utrudnia rozwiązanie problemu.
    # Usuwane są także wszystkie kolumny zawierające dane nieliczbowe
    df = df.drop(columns=['G1', 'G2']).select_dtypes(include=['number'])

    # Zapisywanie jako pliku csv
    df.to_csv('portugese_student_numeric.csv', index=False)

# grab_student_numeric()
```

7.6.3. Ćwiczenia

Spróbuj zastosować techniki oceny metod regresji do zbioru danych *boston*, który można pobrać za pomocą wywołania `datasets.load_boston`.

A oto pytanie prowokujące do myślenia: jaka zależność między rzeczywistymi wartościami i predykcjami da w efekcie *liniowy* wykres składników resztowych?

UCZENIE MASZYNOWE Z PYTHONEM: OD DZIŚ DLA KAŻDEGO!

Sztuczna inteligencja i uczenie maszynowe rozwijają się z niezwykłą dynamiką i znajdują coraz więcej różnorodnych zastosowań w niemal wszystkich branżach. Ten spektakularny postęp jest silnie związany z osiągnięciami w świecie sprzętu i oprogramowania. Obecnie do uczenia maszynowego używa się wielu języków programowania, takich jak R, C, C++, Fortran i Go, jednak najpopularniejszym wyborem okazał się Python wraz z jego specjalistycznymi bibliotekami. Znajomość tych bibliotek i narzędzi umożliwia tworzenie systemów uczących się nawet tym osobom, które nie dysponują głęboką wiedzą z dziedziny matematyki.

Ta książka jest przeznaczona dla każdego, kto choć trochę zna Pythona i chce nauczyć się uczenia maszynowego. Zagadnienia matematyczne zostały tu zaprezentowane w minimalnym stopniu, za to więcej uwagi poświęcono koncepcjom, na których oparto najważniejsze i najczęściej używane narzędzia oraz techniki uczenia maszynowego. Następnie pokazano praktyczne zasady implementacji uczenia maszynowego z wykorzystaniem najdoskonalszych bibliotek i narzędzi Pythona. Opisano używane dziś komponenty systemów uczących się, w tym techniki klasyfikacji i regresji, a także inżynierię cech, która pozwala przekształcać dane na użyteczną postać. Przeanalizowano liczne algorytmy i najczęściej stosowane techniki uczenia maszynowego. Pokrótce przedstawiono modele grafowe i sieci neuronowe, w tym sieci głębokie, jak również połączenie tych technik z bardziej zaawansowanymi metodami, przydatnymi choćby w pracy na danych graficznych i tekstowych.

W książce między innymi:

- algorytmy i modele uczenia maszynowego
- zasady oceny skuteczności systemów uczących
- techniki przekształcania danych
- techniki uczenia maszynowego do obrazu i tekstu
- sieci neuronowe i modele grafowe
- biblioteka scikit-learn i inne narzędzia Pythona

Dr Mark E. Fenner uczy dorosłych informatyki i matematyki. Prowadził badania w dziedzinach uczenia maszynowego, bioinformatyki i bezpieczeństwa systemów komputerowych. Zajmował się też analizą bezpieczeństwa repozytoriów oprogramowania, probabilistycznym modelowaniem białek oraz analizą i wizualizacją danych pochodzących z badań ekologicznych i mikroskopowych. Mieszka z rodziną w południowo-wschodniej Pensylwanii.

ISBN 978-83-283-6425-7