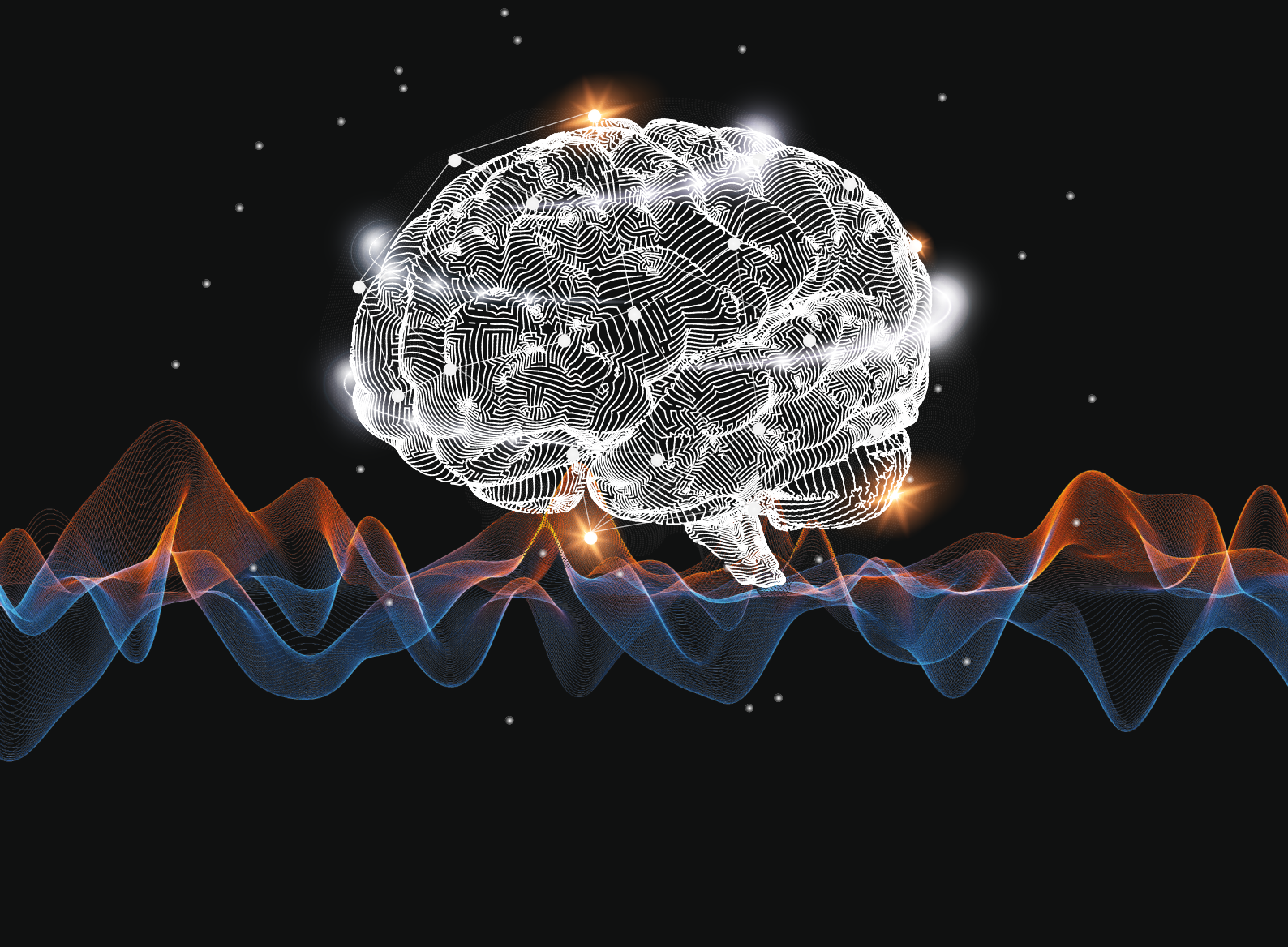
Aleksandra Król-Nowak Katarzyna Kotarba

Podstawy uczenia maszynowego

pod redakcją
Aleksandry Król-Nowak



WYDAWNICTWA AGH

KRAKÓW 2022

Podstawy uczenia maszynowego

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
AGH Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Unia Europejska Europejski Fundusz Społeczny

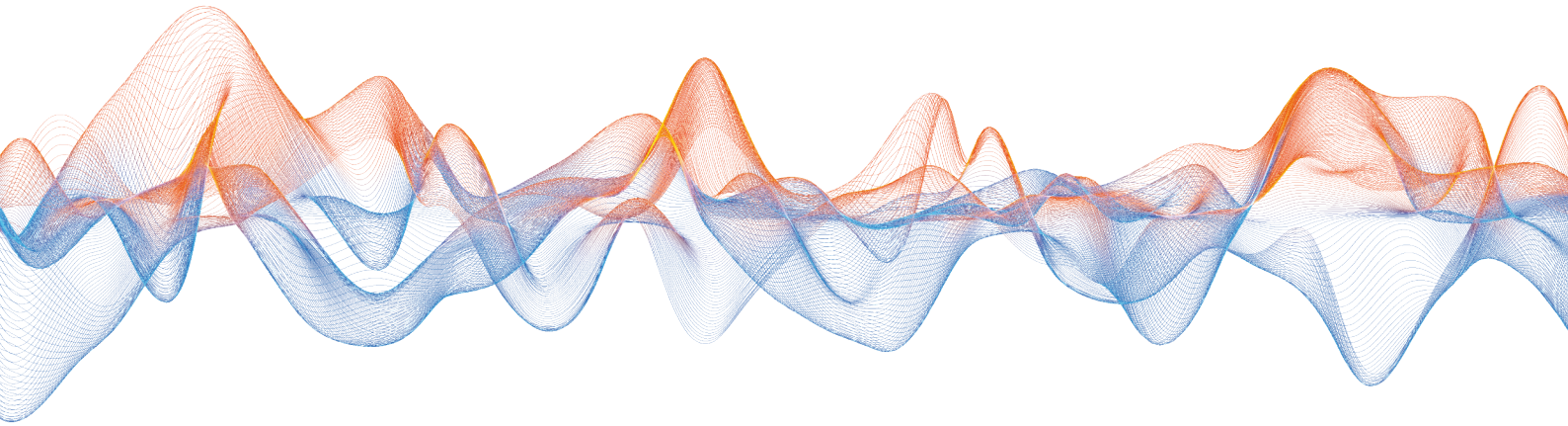
Informacja o projekcie:

Współfinansowano ze środków Europejskiego Funduszu Społecznego
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014–2020
Oś III Szkolnictwo wyższe dla gospodarki i rozwoju,
Działanie 3.5 Kompleksowe programy szkół wyższych
Zintegrowany Program Rozwoju Akademii Górniczo-Hutniczej w Krakowie,
nr POWR.03.05.00-00-Z307/17-00

Aleksandra Król-Nowak Katarzyna Kotarba

Podstawy uczenia maszynowego

pod redakcją
Aleksandry Król-Nowak



WYDAWNICTWA AGH

KRAKÓW 2022

Wydawnictwa Akademii Górniczo-Hutniczej im. Stanisława Staszica w Krakowie

e-ISBN 978-83-67427-05-0

Dyrektor Wydawnictw AGH: *Jan Sas*

Komitet Naukowy Wydawnictw AGH:
Marek Gorgoń (przewodniczący)
Barbara Gąciarz
Elżbieta Pamuła
Bogdan Sapiński
Stanisław Stryczek
Tadeusz Telejko

Recenzenci: *dr hab. inż. Marek Pluta, prof. AGH*
prof. dr hab. inż. Andrzej Maciej Skulimowski

Redakcja: *Aleksandra Król-Nowak*
AGH Akademia Górniczo-Hutnicza
Wydział Inżynierii Mechanicznej i Robotyki
Katedra Mechaniki i Wibroakustyki

Afiliacja autorek:
AGH Akademia Górniczo-Hutnicza
Wydział Inżynierii Mechanicznej i Robotyki
Katedra Mechaniki i Wibroakustyki

Opracowanie redakcyjne: *Agnieszka Rusinek*

Projekt okładki i stron tytułowych: *Alicja Pronobis*
Na okładce wykorzystano zasoby portalu Freepik.com

Skład: *MUNDA*

Redakcja Wydawnictw AGH
al. A. Mickiewicza 30, 30-059 Kraków
tel. 12 617 32 28, 12 636 40 38
e-mail: redakcja@wydawnictwoagh.pl
www.wydawnictwo.agh.edu.pl

Spis treści

Przedmowa

Niniejsza praca przeznaczona jest przede wszystkim dla studentów inżynierii akustycznej, jako pomoc naukowa w realizacji treści przedmiotu podstawy uczenia maszynowego w technologiach akustycznych. Jest to kompendium wykładanych treści w postaci zestawienia podstawowych algorytmów uczenia maszynowego wraz z wprowadzeniem do implementacji tych metod przy użyciu języka Python, na przykładzie wybranych zadań z rozwiązaniami.

Celem niniejszej monografii nie było szczegółowe przedstawienie każdej z metod uczenia maszynowego. Autorki chciały uniknąć stworzenia kolejnego bardzo fachowego podręcznika, którego lektura wymagałaby długiego czasu i dużego zaangażowania czytelnika, a tym samym ryzyka, że studenci chętniej sięgną po coraz liczniejsze, ogólnodostępne w Internecie teksty popularnonaukowe. Dlatego też zamierzeniem autorek było możliwie zwięzłe przedstawienie podstawowych metod uczenia maszynowego z odniesieniami do literatury fachowej, pozwalającymi zainteresowanym czytelnikom znaleźć bardziej szczegółowy opis poszczególnych zagadnień. Ponadto zadaniem publikacji jest uwypuklenie podstawowych treści, które na wykładzie z podstaw uczenia maszynowego w technologiach akustycznych są przedstawione w dużo szerszym zakresie. Praca ta ma być zatem bazą podstawowej wiedzy dla osób, które dopiero rozpoczynają naukę uczenia maszynowego. Dlatego też w większości przypadków, gdzie zrozumienie sposobu działania danego algorytmu tego nie wymagało, zrezygnowano z formalizmów matematycznych. Jednak w uzasadnionych przypadkach, pod warunkiem, że było to możliwe bez wykraczania poza podstawy matematyki wyższej, w celu wytłumaczenia powiązania pewnych algorytmów, zamieszczono wyprowadzenia matematyczne tych metod. Powinno to zaspokoić co bardziej dociekliwych czytelników.

W monografii zestawiono najbardziej podstawowe algorytmy, które pozwalają w przystępny sposób przedstawić istotę rozwiązywania problemów metodami uczenia maszynowego. Poznanie idei działania wybranych metod pozwoli studentom w przyszłości zrozumieć również bardziej zaawansowane algorytmy uczenia maszynowego i świadomie ich używać . Wybór opisanych metod podyktowany był też próbą przedstawienia szerokiego wachlarza technik stosowanych w uczeniu maszynowym.

Pomimo ścisłego powiązania niniejszej publikacji z konkretnym programem studiów zadbano, aby prezentowane zadania i przykłady zastosowań poszczególnych algorytmów nie ograniczały się wyłącznie do technologii akustycznych. W monografii można znaleźć proste przykłady z bardzo różnych dziedzin. Z jednej strony ma to na celu pokazanie szerokich możliwości zastosowania przedstawionych metod, natomiast z drugiej powoduje, że praca ta może być wartościową pozycją nie tylko dla studentów inżynierii akustycznej.

Pierwsze pięć rozdziałów ma na celu wprowadzenie czytelnika w ogólne nazewnictwo i ideę uczenia maszynowego. Rozdział 1 tłumaczy, czym jest uczenie maszynowe w kontekście sztucznej inteligencji. W drugim rozdziale przedstawiono ogólne kroki postępowania w rozwiązywaniu problemów metodami uczenia maszynowego. W rozdziale 3 ustalono oznaczenia obowiązujące w całej publikacji, a także podzielono metody uczenia maszynowego – zgodnie z przyjętą nomenklaturą – na uczenie nadzorowane i nienadzorowane. Omówiono także zagadnienie podziału zbioru danych na zbiór treningowy i testowy. W rozdziale 4 rozróżniono trzy podstawowe problemy rozwiązywane metodami uczenia maszynowego, czyli regresję, klasyfikację i klasteryzację. Rozdział 5 posłużył wprowadzeniu miar sukcesu w zależności od tego, czy mamy do czynienia z regresją, klasyfikacją, czy klasteryzacją. W rozdziale tym przedstawiono jedynie najbardziej uniwersalne miary sukcesu, natomiast te bardziej specyficzne, jak np. strata logarytmiczna czy metryka ROC AUC, zostały opisane w późniejszych rozdziałach wraz z odpowiednimi algorytmami.

Celem kolejnych dwóch rozdziałów jest wprowadzenie do implementacji metod uczenia maszynowego. Najpierw przedstawiono konfigurację środowiska programistycznego, a następnie sposoby parametryzacji sygnału. W rozdziałach 8–17 przedstawiono konkretne algorytmy stosowane w uczeniu maszynowym. Większość z tych rozdziałów składa się z trzech części. Pierwsza część to syntetyczne wprowadzenie do metody, ogólna idea jej działania oraz kroki algorytmu, druga to implementacja konkretnych przykładów, natomiast trzecia to zadania do samodzielnego opracowania, przy czym ich rozwiązania można znaleźć w ostatnim rozdziale.

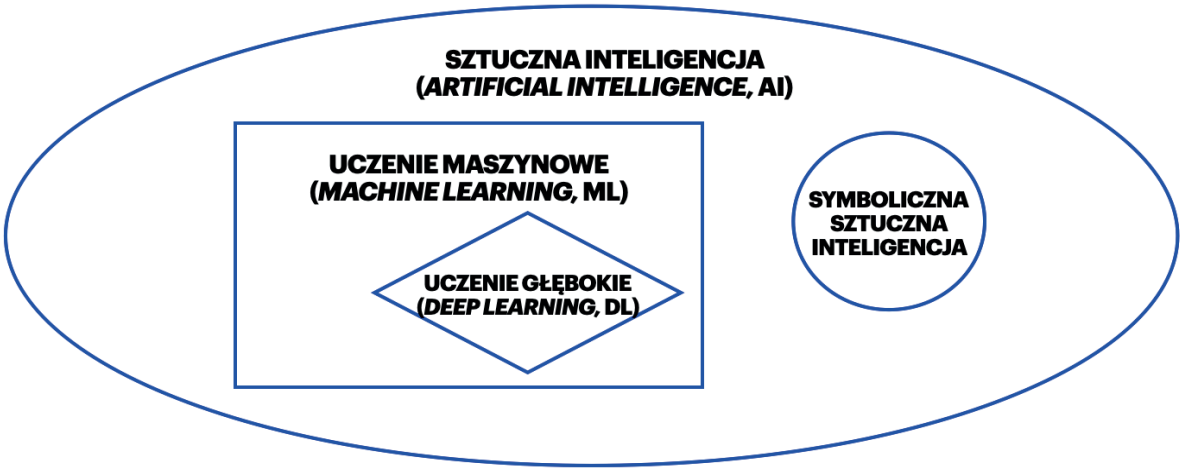
Kolejność przedstawionych metod została ustalona tak, aby stopniować trudność w zrozumieniu działania przedstawianych metod, ale też sukcesywnie wprowadzać kolejne funkcje w języku Python. Tym samym, w rozdziale 8 opisano najprostszą z metod uczenia maszynowego, czyli klasyfikację algorytmem *k*-najbliższych sąsiadów. W następnym rozdziale omówiono problem przekleństwa wymiarowości oraz przedstawiono dwie metody redukcji wymiarowości. Kolejne dwa rozdziały przedstawiają metody klasteryzacji – algorytm *k*-średnich oraz algorytmy hierarchiczne. Rozdziały 12–14 stanowią pewną integralną całość i dotyczą uogólnionych modeli liniowych. W rozdziale 12 przedstawiono model regresji liniowej. Rozdział 13 jest teoretycznym wprowadzeniem pojęcia uogólnionych modeli liniowych. Wykazano też, że opisana we wcześniejszym rozdziale regresja liniowa jest szczególnym przypadkiem uogólnionego modelu liniowego. Natomiast w rozdziale 14 wprowadzono regresję logistyczną jako uogólniony model liniowy, pokazując tym samym sposób, w jaki należy konstruować model liniowy adekwatnie do rozwiązywanego problemu. W rozdziale 15 omówiono trudniejszą metodę klasyfikacji, jaką jest maszyna wektorów nośnych. Rozdział 16 to opis drzew klasyfikacyjnych z przykładem agregacji klasyfikatorów metodą lasów losowych. Rozdział 17 stanowi wprowadzenie do sieci neuronowych. Aby nie zdominować publikacji zagadnieniami z tej tematyki, ograniczono się do przedstawienia jedynie ogólnej idei i najprostszych realizacji. Jeżeli w którymś z rozdziałów potrzebna była znajomość pewnych definicji, twierdzeń czy własności z zakresu podstawowego kursu matematyki na studiach, ich przypomnienie podano w dodatku.

1 Wprowadzenie

Sztuczna inteligencja jest dziś bardzo popularnym kierunkiem. Można o niej przeczytać zarówno w bardzo fachowych czasopismach, jak i – coraz częściej – w źródłach popularnonaukowych. Wraz ze wzrostem popularności rośnie grupa entuzjastów prezentujących nieraz wręcz utopijne wizje tego jak sztuczna inteligencja już niebawem zmieni nasz świat. Nie brakuje jednak także sceptyków. Do tej grupy należą często osoby, które śledzą historię dokonań w dziedzinie sztucznej inteligencji i wiedzą, że wiele z nierealnych wizji zweryfikowała już rzeczywistość. Wydaje się zatem, że w środowisku naukowym minęły już czasy wyolbrzymiania możliwości, jakie daje wykorzystanie narzędzi sztucznej inteligencji. W czasie rozwoju tej dziedziny w ostatnich dekadach obserwowano zarówno wzloty, jak i upadki. Odkrycie niektórych metod sztucznej inteligencji rzeczywiście zrewolucjonizowało wiele różnych dziedzin, ale zwykle po tym przychodziły momenty stagnacji i zapomnienia, aż do kolejnych przełomów. Dzisiaj w niemal każdym obszarze naszego życia można doszukać się mniej lub bardziej znaczących zastosowań sztucznej inteligencji, także w obszarze akustyki i inżynierii dźwięku. Dlatego zdecydowanie warto poznać przynajmniej podstawowe narzędzia sztucznej inteligencji, a uczenie maszynowe w szczególności. Należy jednak pamiętać, że metody sztucznej inteligencji wspierają pracę ludzi, ale wciąż jeszcze to wiedza i doświadczenie człowieka są niezastąpione.

1.1. Sztuczna inteligencja, uczenie maszynowe i uczenie głębokie

Na początku należy wyjaśnić pojęcia, które kojarzą się z tematem uczenia maszynowego, ale dość często są mylnie używane. Najszerszym pojęciem jest tutaj sztuczna inteligencja (ang. *Artificial Intelligence*, AI) i jest to dział, do którego należą nie tylko metody uczenia maszynowego (ang. *Machine Learning*, ML), ale też wiele innych. Szczególnym przypadkiem uczenia maszynowego jest uczenie głębokie (ang. *Deep Learning*, DL). Natomiast najlepszym przykładem metod sztucznej inteligencji, które wyraźnie odróżniają się od metod uczenia maszynowego, jest symboliczna sztuczna inteligencja. Zależność pomiędzy tymi pojęciami przedstawia rysunek 1.1.

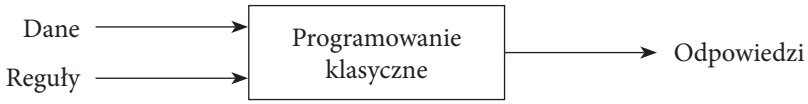


Rysunek 1.1. Podział metod z zakresu sztucznej inteligencji

Aby zrozumieć podstawową różnicę pomiędzy uczeniem maszynowym a symboliczną sztuczną inteligencją, wyjaśnijmy najpierw, czym jest symboliczna sztuczna inteligencja. Najlepszym jej przykładem są wczesne symulatory gry w szachy. W takich symulatorach to programista musiał wcześniej opracować reguły. Natomiast w uczeniu maszynowym dostarczane są dane, ale to algorytm ma za zadanie znalezienie reguł. Oczywiście nie oznacza to, że maszyna zastąpi człowieka. Sukces uczenia maszynowego wiąże się z wiedzą i doświadczeniem osoby, która go używa. Dotyczy to przede wszystkim wyboru adekwatnego algorytmu, odpowiadającego charakterowi problemu, z którym się mierzymy. Ważne jest także odpowiednie przygotowanie danych. Niemniej jednak ostatecznie to algorytm uczenia maszynowego ma dostarczyć zestawu reguł, które będą odpowiedzialne za udzielanie odpowiedzi w zadanym przez nas problemie, przy czym reguły te często nie są określane jako jawny wzór.

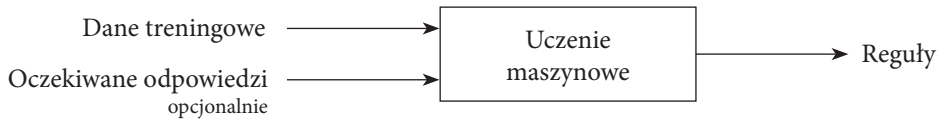
1.2. Klasyczne programowanie a uczenie maszynowe

Porównanie wspomnianego we wcześniejszym podrozdziale przykładu algorytmu sztucznej inteligencji z uczeniem maszynowym prowadzi do ogólnego rozróżnienia klasycznego programowania i metod uczenia maszynowego. Schemat rozwiązywania problemu za pomocą programowania klasycznego przedstawia rysunek 1.2.



Rysunek 1.2. Schemat rozwiązywania problemu za pomocą programowania klasycznego

Aby zwizualizować proces rozwiązywania problemów za pomocą uczenia maszynowego, trzeba wyszczególnić dwa etapy: etap trenowania systemu w celu określenia reguł, które będą podstawą modelu, oraz etap docelowego użycia otrzymanego modelu, co przedstawiają kolejne schematy (rys. 1.3 oraz rys. 1.4).



Rysunek 1.3. Schemat trenowania modelu za pomocą uczenia maszynowego

Zauważmy, że w przypadku uczenia maszynowego na wejściu możemy dysponować jedynie danymi treningowymi. Oczekiwane odpowiedzi podajemy opcjonalnie (w zależności od tego, czy mamy do czynienia z uczeniem nadzorowanym, czy nienadzorowanym, przyjrzymy się temu tematowi dokładniej w podrozdziale 3.2). Następnie na podstawie danych wejściowych wytrenowany model przewiduje dane wyjściowe.

Rysunek 1.4. Schemat rozwiązywania problemu za pomocą modelu wytrenowanego za pomocą metod uczenia maszynowego

2 Etapy uczenia maszynowego

Uczenie maszynowe obejmuje szeroki wachlarz bardzo zróżnicowanych metod mających jednak pewne wspólne cechy możliwe do uogólnienia. W ten sposób można przybliżyć zestaw kroków, które niezależnie od metody należy wykonać, aby za pomocą uczenia maszynowego rozwiązać dany problem. W rozdziale tym zostaną omówione poszczególne etapy uczenia maszynowego.

1. Zdefiniowanie problemu

Im precyzyjniej zostanie określony problem, którego rozwiązanie jest poszukiwane, tym lepiej będzie można dobrać najefektywniejszy algorytm uczenia maszynowego. Problem należy docelowo zdefiniować w języku metod uczenia maszynowego, czyli dopasować go do dostępnych metod uczenia maszynowego. Wybranie właściwej metody ułatwia uprzednie zaklasyfikowanie rozwiązywanego problemu do odpowiedniej grupy. Najczęściej wyszczególnia się trzy typy problemów, które można rozwiązać za pomocą algorytmów: regresji, klasyfikacji albo klasteryzacji. Języki, którymi posługują się wyszczególnione grupy algorytmów uczenia maszynowego, zostały opisane w rozdziale 4. Należy także nadmienić, że wiele problemów tylko pozornie nie spełnia wymagań metod uczenia maszynowego i często wystarczy problem przedefiniować, aby móc skutecznie użyć jednego z algorytmów. Dlatego też jest to bardzo ważny etap, od którego w dużym stopniu zależy ewentualny sukces zastosowania metod uczenia maszynowego.

2. Zebranie danych wejściowych i – jeżeli to możliwe – danych wyjściowych

Od jakości i ilości danych zależy jakość modelu. Szczegółowe informacje dotyczące zbioru danych opisano w rozdziale 3. Ważnym pojęciem związanym z wymaganą ilością danych jest przekleństwo wymiarowości.

3. Uporządkowanie danych

Etap ten dotyczy standardowych działań wykonywanych w celu przygotowania danych do analizy, takich jak: czyszczenie, imputacja, standaryzowanie, formatowanie, ewentualnie wstępne grupowanie czy selekcja. Zwykle na tym etapie dokonuje się również przygotowanie danych zgodnie z wymaganiami algorytmu, który będziemy chcieli zastosować.

4. Jeśli to możliwe, podział na zbiory: treningowy i testowy

Każdy model musi być trenowany na pewnym zbiorze danych, ale potrzebny jest również oddzielny zbiór obiektów do weryfikacji, czy wytrenowany model działa w sposób zadowalający. W tym celu, zanim rozpoczęty zostanie etap trenowania modelu, należy zbiór danych podzielić. Więcej informacji na ten temat można znaleźć w podrozdziale 3.3.

5. Wybór algorytmu

W pierwszym kroku, w wyniku zdefiniowania problemu w języku technik uczenia maszynowego, wybór został zawężony do konkretnej grupy algorytmów, np. regresji, klasyfikacji czy klasteryzacji. Wybór grupy algorytmów determinuje zwykle, czy powinno to być uczenie z nadzorem, czy bez nadzoru. Jednak w obrębie każdej grupy mamy wiele algorytmów, które w różny sposób będą prowadzić do tego samego

celu, ale ich np. sprawność czy efektywność będzie inna w zależności od danych. I w tym miejscu należy zaznaczyć, że uczenie maszynowe nie jest nauką teoretyczną, ale empiryczną. Oczywiście dysponujemy pewnym zasobem wiedzy o poszczególnych algorytmach, ale ostatecznie liczą się rezultaty. Jeżeli dany algorytm daje w naszym przypadku zadowalające efekty, to znaczy, że jest odpowiedni. Natomiast określenie, który algorytm będzie dla nas najlepszy, często następuje w wyniku prób i błędów. Dlatego zachęca się do próbowania i nie poddawania się, jeżeli pierwszy zastosowany algorytm zawiedzie. Uczenie maszynowe jest dla wytrwałych. W kolejnych rozdziałach zostaną scharakteryzowane i szczegółowo opisane wybrane algorytmy uczenia maszynowego.

UWAGA

Wybór algorytmu powinno się **zaczynać od najprostszych algorytmów. Nie jest prawdą, że im bardziej złożony algorytm, tym lepiej sobie poradzi i będzie bardziej uniwersalny**. Ponadto warto zawsze skorzystać z wiedzy i doświadczenia osób, które z podobnym problemem mierzyły się już wcześniej. Polecane źródła takiej wiedzy to: Papers with Code, Kaggle, Arxiv, *DeepLearn.org*, Github, publikacje naukowe.

6. Ustalenie wartości hiperparametrów dla wybranego algorytmu

Większość algorytmów uczenia maszynowego wymaga określenia wartości pewnych zmiennych konfiguracyjnych zwanych **hiperparametrami**. W zależności od algorytmu może to być np.:

- liczba *k* w algorytmie *k*-najbliższych sąsiadów,
- maksymalna głębokość drzewa decyzyjnego,
- liczba drzew w lesie losowym,
- liczba warstw w sieci neuronowej.

Hiperparametry muszą zostać podane a priori, nie uzyskuje się ich w wyniku trenowania modelu ani nie są modyfikowane w trakcie wykonywania algorytmu, chyba że są zadane w postaci funkcji zależnej od np. numeru iteracji. Wybór wartości hiperparametrów danego algorytmu wpływa na jego pracę i przekłada się na uzyskane wyniki

7. Określenie miary sukcesu

Miary sukcesu, najczęściej wyrażone jako miary błędu, są potrzebne, aby ocenić, czy wytrenowany model pozwala uzyskać zadowalające efekty. Wybór algorytmu determinuje zwykle dostępną grupę miar sukcesu, ale nie oznacza to, że nie możemy zdefiniować swojej miary, która będzie weryfikować to, na czym nam najbardziej zależy. Opis najczęściej stosowanych miar sukcesu można znaleźć w rozdziale 5.

8. Trening

Jest to etap bezpośredniego powstawania modelu. Dane treningowe są przedstawiane w wybranym algorytmie, a algorytm na ich podstawie buduje reguły. Celowo została tutaj użyta forma wskazująca na to, że algorytm wykonuje tę pracę samodzielnie. Tak jak zostało to opisane w podrozdziale 1.1, w odróżnieniu od programowania klasycznego, algorytmy uczenia maszynowego służą opracowaniu reguł. Należy jednak zaznaczyć, że zwykle finalnie opracowane reguły nie mają postaci jawnej. Wytrenowany model jest uogólnieniem zależności, które można zaobserwować w prezentowanych danych.

9. Ewaluacja otrzymanego modelu

W tym celu wykorzystuje się ustaloną wcześniej miarę sukcesu. Jeżeli wynik jest zadowalający, uczenie maszynowe dobiega końca i możemy korzystać z uzyskanego modelu. W przeciwny wypadku przechodzimy do kolejnego etapu.

10. Optymalizacja hiperparametrów

Jeżeli ewaluacja wskazała, że wytrenowany model nie jest zadowalający, zanim zdecydujemy się wykorzystać inny algorytm, warto dokonać zmian w obrębie ustalonych wartości hiperparametrów wybranego algorytmu. Po wyborze innej wartości hiperparametru należy wrócić do kroku numer 7, czyli na nowo wytrenować model i ponownie go ocenić. Jeżeli sterowanie hiperparametrami nie przyniesie rezultatów, trzeba wrócić do etapu wyboru algorytmu. Jeżeli kolejne algorytmy nie będą działać zgodnie z oczekiwaniami, warto przeanalizować możliwość zmian w punktach 1–4.

3 Reprezentacja danych

Oprócz znajomości algorytmów uczenia maszynowego ważnym kluczem do sukcesu jest również dysponowanie odpowiednimi danymi. Można nawet powiedzieć, że na osiągnięty sukces w dziedzinie uczenia maszynowego ma wpływ nie tylko umiejętnie użyty algorytm, ale w dużej mierze również to, jakie dane zostały użyte. Ważna jest świadomość tego, jak potrzebny jest ten etap, ponieważ od niego może zależeć powodzenie całej pracy. Przy czym konieczna jest tutaj nie tylko odpowiednia ilość, ale też jakość danych i adekwatna do rozpatrywanego problemu ich reprezentacja.

Aspekty, na które należy zwrócić uwagę przy przygotowywaniu danych, podano poniżej.

Reprezentatywność

Nie możemy oczekiwać od systemu, że nauczy się dobrze np. klasyfikować dowolne przykłady muzyczne, jeżeli w zbiorze treningowym znajdą się tylko przykłady wybranych gatunków muzycznych z biblioteki ulubionych utworów autora pracy. Aby zapewnić reprezentatywność danych, wystarczy zadbać, aby dobór przykładów treningowych był próbą losową prostą, tj. ciągiem niezależnych zmiennych losowych o jednakowych rozkładach, takich samych jak zmienna losowa w całej populacji. Zatem próbę losową pobieramy z populacji wszystkich możliwych przykładów dla danego problemu.

Jakość

Dane powinny być usystematyzowane, a wszelkie ich braki powinny zostać rozpatrzone pod kątem zysków i strat. Zatem jeżeli mamy odpowiednio dużo danych, odrzucamy te niekompletne, w przeciwnym wypadku dokonujemy imputacji brakujących wartości w obrębie wybrakowanych atrybutów.

Ilość

W tym punkcie należy wspomnieć o znamiennym przekleństwie wymiarowości, czyli wykładniczym wzroście wymaganej liczby danych wraz ze wzrostem liczby zmiennych w modelu (opisanym w rozdziale 9).

Ujednolicony zapis

Analizę bardzo upraszcza konsekwencja i jednolity zapis danych. Dlatego poniżej wprowadzony zostanie dość intuicyjny zapis wektorowy, który dobrze oddaje wielowymiarowy charakter obiektów w zbiorach treningowym i testowym.

3.1. Oznaczenia i nomenklatura

Najczęściej stosowaną reprezentacją danych w uczeniu maszynowym jest reprezentacja atrybut-wartość. W uczeniu maszynowym każdy obiekt jest opisany przez ustalony zestaw wartości zadanych atrybutów warunkowych. Zatem jeżeli rozważamy d atrybutów ($d \in \mathbb{N}$), to dla każdego obiektu w zbiorze danych musi zostać podanych d wartości, przy czym każda z nich pochodzi ze zbioru wartości tego atrybutu. Ustalone wartości atrybutów dla kolejnych obiektów w zbiorze danych przestawia tzw. tablica informacyjna (tab. 3.1).

Zatem w N-licznym zbiorze d-wymiarowych danych możemy przyjąć następujące oznaczenia:
X – d-wymiarowa przestrzeń, z której pochodzą **dane wejściowe**, $d \in N$,
$x^{(j)} = (x_1^{(j)}, x_2^{(j)}, ..., x_d^{(j)})$, $j = 1, ..., N$ – ***j*-ta dana wejściowa**,
$x_i^{(j)}$, $i = 1, 2, ..., d$ – **wartość *i*-tego atrybutu warunkowego dla *j*-tego przykładu**.

Tabela 3.1. Tablica informacyjna

	A_1	A_2	...	A_d
$\boldsymbol{x^{(1)}}$	$x_1^{(1)}$	$x_2^{(1)}$	...	$x_d^{(1)}$
$\boldsymbol{x^{(2)}}$	$x_1^{(2)}$	$x_2^{(2)}$	...	$x_d^{(2)}$
...	...	...	...	...
$\boldsymbol{x^{(N)}}$	$x_1^{(N)}$	$x_2^{(N)}$	...	$x_d^{(N)}$

Atrybuty warunkowe są też zamiennie nazywane cechami lub po prostu zmiennymi niezależnymi. Tak określonych danych używa się w celu nienadzorowanego trenowania modelu. W przypadku gdy algorytm realizuje uczenie z nadzorem, należy jeszcze dla każdego obiektu określić wartość tzw. atrybutu decyzyjnego. Wtedy:
Y – jednowymiarowa przestrzeń, z której pochodzą wartości atrybutu decyzyjnego,
$y^{(j)}$ – wartość atrybutu decyzyjnego dla j-tego obiektu.

Atrybut decyzyjny jest też zamiennie nazywany zmienną zależną albo daną wyjściową. Tablica informacyjna uwzględniająca atrybut decyzyjny nazywana jest rozszerzoną tablicą informacyjną (tab. 3.2).

Tabela 3.2. Rozszerzona tablica informacyjna

	A_1	A_2	...	A_d	D
$\boldsymbol{x^{(1)}}$	$x_1^{(1)}$	$x_2^{(1)}$	...	$x_d^{(1)}$	$y^{(1)}$
$\boldsymbol{x^{(2)}}$	$x_1^{(2)}$	$x_2^{(2)}$	...	$x_d^{(2)}$	$y^{(2)}$
...	...	...	...	...	...
$\boldsymbol{x^{(N)}}$	$x_1^{(N)}$	$x_2^{(N)}$	...	$x_d^{(N)}$	$y^{(N)}$

Obiekty i zbiór danych są wtedy postaci:
$(x^{(j)}, y^{(j)}) = (x_1^{(j)}, x_2^{(j)}, ..., x_d^{(j)}, y^{(j)})$ – pojedynczy obiekt, para danej wejściowej i odpowiadającej jej danej wyjściowej,
$X = \{(x^{(j)}, y^{(j)}), j = 1, ..., N\}$ – zbiór danych.

Mówiąc o zbiorze wartości poszczególnych atrybutów, należy wspomnieć, że każdy z nich może być określony innym typem zmiennej. Wyszczególnić należy trzy typy atrybutów:

1) **nominalne** (dziedziny są zbiorami nieuporządkowanymi, możliwe jest określenie tylko relacji równości i braku równości, np. płeć);
2) **porządkowe** (dziedziny są zbiorami uporządkowanymi, możliwe jest określenie relacji porządku liniowego, np. wysokość określona wartościami {bardzo niski, niski, średni, wysoki, bardzo wysoki}), uwaga: znajomość porządku nie oznacza znajomości odległości między obiektami);
3) **liczbowe/numeryczne/ciągłe** (dziedziny zdefiniowane na liczbowych skalach pomiarowych, przedziałowych lub ilorazowych, możliwe są obliczenia algebraiczne na wartościach).

3.2. Uczenie nadzorowane i nienadzorowane

Podczas wprowadzania oznaczeń wspomniane zostały atrybuty warunkowe i opcjonalny atrybut decyzyjny, który determinuje, czy przeprowadzane trenowanie modelu odbywa się z nadzorem, czy bez niego.

Uczenie nadzorowane (inaczej uczenie z nadzorem, uczenie z nauczycielem):
- $x^{(j)} = (x_1^{(j)}, x_2^{(j)}, ..., x_d^{(j)}, y^{(j)})$ przykład uczący,
- dane treningowe składają się z danych wejściowych i odpowiadających im danych wyjściowych (oczekiwane odpowiedzi systemu),
- na podstawie zbioru danych treningowych system uczy się zależności danych wyjściowych od danych wejściowych, przy czym powinien tę zależność uogólnić, czyli opracować funkcję f odwzorowującą $X \rightarrow Y$.

Uczenie nienadzorowane (inaczej uczenie bez nadzoru, uczenie bez nauczyciela):
- $x^{(j)} = (x_1^{(j)}, x_2^{(j)}, ..., x_d^{(j)})$ przykład uczący,
- brak podania oczekiwanych odpowiedzi,
- system ma sam odkryć regularności w zbiorze danych, np. grupowanie przykładów w klasy.

Należy zaznaczyć, że to, czy będziemy przeprowadzać uczenie z nadzorem, czy bez nadzoru, często nie zależy od nas, ale od charakteru naszego problemu i tego, jakimi danymi dysponujemy.

3.3. Zbiór treningowy i testowy

Dane, którymi dysponujemy, są konieczne, aby model mógł nauczyć się wykonywać zadania (np. regresji czy klasyfikacji). Dane odzwierciedlają specyfikę danego problemu i pozwalają ją zamodelować. Należy jednak sprawdzić działanie wytrenowanego modelu, w szczególności upewnić się, czy system poradzi sobie z nowym przykładem spoza danych treningowych. W przypadku uczenia nadzorowanego jest to bardzo proste. Wystarczy zostawić pewną liczbę obiektów „na później", to znaczy nie pokazywać ich systemowi na etapie trenowania, tylko użyć do sprawdzenia wytrenowanego modelu. Sprawdzenie polega na porównaniu wyniku generowanego przez model z wartością atrybutu decyzyjnego dla tej samej danej wejściowej.

Dlatego też już na samym początku, zanim zaczniemy trenować model, dane, którymi dysponujemy, dzielimy na zbiór danych treningowych (inaczej zwany zbiorem danych uczących) i testowych. Należy w tym miejscu zaznaczyć, że podział na zbiór treningowy i testowy musi być losowy. Dla dużych zbiorów danych, do których należy co najmniej kilka tysięcy przykładów, stosuje się jednokrotny podział losowy (ang. *hold-out*). Najczęściej przyjmowane wtedy proporcje zbiorów treningowego i testowego to odpowiednio 2:1 lub 9:1.

Problem pojawia się wtedy, gdy dysponujemy małym zbiorem danych. W takim przypadku może okazać się, że odłożenie pewnej części danych na etap testowania spowoduje, że dane pozostałe w zbiorze treningowym będą niewystarczające do wytrenowania dostatecznie dobrego modelu. Dlatego dla mniejszych zbiorów danych przeprowadza się trening modelu na wszystkich dostępnych danych, a w celu oceny otrzymanego klasyfikatora estymuje się błąd klasyfikatora przy zastosowaniu jednej z metod podziału wielokrotnego:
- random sub-sampling,
- k-krotna ocena krzyżowa (ang. *k-fold cross validation*),
- leaving-one-out,
- bootstrap (dla najmniejszych zbiorów danych).

Nazwy 'random sub-sampling', 'leaving-one-out' oraz 'bootstrap' zostały przytoczone bez polskiego tłumaczenia, ponieważ właśnie w takiej postaci są one powszechnie używane w środowisku inżynierów uczenia maszynowego.

Metoda random sub-sampling

W metodzie random sub-sampling powtarzane są jednokrotne podziały losowe. W wyniku każdego podziału otrzymujemy pary zbiorów treningowych i odpowiadających im zbiorów testowych. Następnie przeprowadzane jest trenowanie modelu i jego testowanie na podstawie wyznaczonych par zbiorów treningowych i testowych oraz obliczane są miary sukcesu (opisane w rozdziale 5) dla kolejnych modeli. Estymowaną oceną klasyfikatora wytrenowanego na całym zbiorze dostępnych danych będzie średnia arytmetyczna z ocen poszczególnych modeli wytrenowanych i ocenionych w kolejnych podziałach.

Metoda *k*-krotnej walidacji krzyżowej i leaving-one-out

Metoda *k*-krotnej walidacji krzyżowej przebiega dokładnie według takiego samego schematu jak w metodzie random sub-sampling z tą różnicą, że poszczególne podziały nie są jednokrotne, ale *N*-elementowy zbiór danych dzielony jest na *k* podzbiorów $X = Z_1 \cup Z_2 \cup ... \cup Z_k$. Następnie w *i*-tej iteracji ($i = 1, ..., k$) Z_i traktowany jest jak zbiór testowy, natomiast pozostałe zbiory tworzą zbiór treningowy. Liczba *k* powinna być tym większa, im mniejszy jest zbiór danych. Często przyjmuje się $k = 10$, a gdy nie uzyskuje się zadowalających wyników, zwiększa się tę wartość. W skrajnym przypadku można dokonać $k = N$ podziałów, czyli w każdej iteracji zbiór testowy będzie jednoelementowy. Jest to metoda leaving-one-out, która jest szczególnym przypadkiem metody *k*-krotnej walidacji krzyżowej. Podobnie jak we wcześniejszej metodzie estymowaną oceną klasyfikatora wytrenowanego na całym zbiorze dostępnych danych będzie średnia arytmetyczna z ocen poszczególnych modeli wytrenowanych i ocenionych w kolejnych podziałach. Można również obliczyć odchylenie standardowe wyników uzyskanych we wszystkich iteracjach, co dostarczy informacji czy model zawsze daje zbliżone wyniki, czy też zdarzyło się, że w pewnej iteracji dał wynik dużo niższy od pozostałych, co nie jest pożądane.

Bootstrap

Ostatnia ze wspomnianych metod to boostrap, którą stosuje się dla najmniej licznych zbiorów danych. W metodzie tej w każdej iteracji losowana jest ze zwracaniem próbka *n*-elementowa, która tworzy zbiór testowy, a pozostałe obiekty tworzą zbiór treningowy.

4 Regresja, klasyfikacja i klasteryzacja

Podstawowe problemy, możliwe do rozwiązania przy użyciu narzędzi uczenia maszynowego, można podzielić na trzy grupy. Pierwsza grupa obejmuje przypadki, w których trzeba przewidywać wyniki pochodzące ze zbioru gęstego, czyli z całego zbioru liczb rzeczywistych lub dowolnego przedziału liczb rzeczywistych. Takie problemy rozwiązywane są zwykle przy użyciu regresji. W kolejnych dwóch grupach są przypadki, w których należy wytrenować model służący do przewidywania klasy ze skończonego zbioru klas. Mowa tu o klasyfikacji i klasteryzacji. Te dwie grupy modeli uczenia maszynowego różni przede wszystkim to, czy dysponujemy wartościami atrybutu decyzyjnego. W przypadku klasyfikacji obiekty w zbiorze treningowym składają się z danych wejściowych i wyjściowych, a więc możemy zastosować uczenie nadzorowane. Tym samym po wytrenowaniu modelu będziemy mogli sklasyfikować nowy przykład spoza zbioru treningowego. W przypadku klasteryzacji mamy do czynienia z uczeniem nienadzorowanym i zadaniem jest pogrupowanie oraz nadanie danym etykiet. Opisany podział przedstawia tabela 4.1.

Tabela 4.1. Regresja, klasyfikacja i klasteryzacja – zestawienie

Regresja	**Klasyfikacja**	**Klasteryzacja**
$y \in R$	y przyjmuje wartości binarne lub nominalne	Brak y
Uczenie nadzorowane	**Uczenie nadzorowane**	**Uczenie nienadzorowane**
Przewidzenie wartości rzeczywistej	Nadanie etykiety nowemu przykładowi	Pogrupowanie przykładów i nadanie im etykiet

W tym miejscu należy dopowiedzieć kilka szczegółów dotyczących ogólnie zagadnienia klasteryzacji. Przede wszystkim należy uzupełnić wprowadzone do tej pory oznaczenia. Niech $C = \{C_1, C_2, ..., C_M\}$ oznacza **zbiór *M* klastrów** (grup), czyli wynik grupowania, tj. podziału zbioru X, natomiast c_k to **centroid** klastra C_k, np. średnia arytmetyczna punktów należących do danego klastra. Zauważmy, że centroid może być wartością spoza zbioru danych.

Podział na klastry może być twardy albo rozmyty. W przeciwieństwie do podziału rozmytego, w podziale twardym każdy obiekt może zostać zaklasyfikowany do dokładnie jednego klastra. W podziale rozmytym należy określić stopień przynależności każdego obiektu do każdego klastra. Wyraża to tzw. współczynnik przynależności u_{jk}:

$$u_{jk} \in [0,1]: \ \forall j \in 1, \ ..., \ N \ \sum_{k=1}^{d} u_{jk} = 1. \tag{4.1}$$

Zatem w twardym podziale, jeżeli $x^{(j)}$ należy do C_k, to $u_{jk} = 1$. Gdy $x^{(j)}$ nie należy do C_k, to $u_{jk} = 0$.

Współczynniki przynależności można zapisać w macierzy $U = [u_{ik}]$, gdzie $u_{ik} \in [0, 1]$ oznacza stopień przynależności $x^{(j)}$ do klastra C_k. W przypadku twardego podziału U będzie macierzą zer i jedynek.

Podział na klastry można reprezentować funkcją:

$$P_X : X \ni x^{(j)} \rightarrow k \in N, \tag{4.2}$$

gdzie $k \in \{1, 2, ..., M\}$ to numer klastra.

Niech P_X będzie przestrzenią podziałów zbioru X. Algorytm klasteryzacji można wyrazić funkcją

$$f_X : P_X \to R, \tag{4.3}$$

której wartości są ocenami poszczególnych podziałów. O tym, jakie przyjmuje się tu miary w ocenie kolejnych podziałów, można przeczytać w rozdziale 5. Natomiast po ustaleniu miary oceny zadaniem algorytmu klasteryzującego jest znalezienie podziału, który będzie najlepiej oceniony, czyli znalezienie maksimum funkcji f_X.

Warto zauważyć, że algorytm klasteryzacji dokonuje podziału na grupy, ale interpretacji, czyli opisu klastrów, dokonuje człowiek. Gdy trudno jest określić jednoznacznie poszczególne etykiety dla wyznaczonych klastrów, można sięgnąć po pewne uniwersalne metody, np.:

- opisać dany klaster przez punkt będący jego środkiem,
- opisać dany klaster przez punkty najbardziej oddalone od środka,
- opisać grupowanie poprzez zadanie koniunkcji odpowiednich wyrażeń logicznych, czyli zadanie warunków na wybrane atrybuty (np. $RT > 1{,}5$ i $\alpha < 0{,}2$).

Oczywiście im lepiej będziemy znać badany problem oraz lepiej przygotujemy dane, tym łatwiej będzie nam nazwać powstałe grupy adekwatnie do tego, co reprezentują w obszarze analizy.

5 Miary sukcesu

Aby ocenić, czy wytrenowany model spełnia oczekiwania i proces uczenia maszynowego można uznać za zakończony, konieczne jest użycie odpowiedniej miary sukcesu. Miary sukcesu to pewnego rodzaju wskaźniki, a ich postać zależy od ocenianej metody uczenia maszynowego. Będziemy mieć zupełnie inne metody oceny dla regresji, inne dla zadań klasyfikacji, jeszcze inne dla klasteryzacji. Często jednak zamiast „miara sukcesu", bardziej adekwatnie byłoby powiedzieć „miara błędu", bo to właśnie popełniane przez wytrenowany model błędy mierzone są najczęściej.

5.1. Regresja

W zadaniach rozwiązywanych metodą regresji do oceny modelu najczęściej stosuje się powszechnie znany błąd średniokwadratowy (ang. *Mean Square Error*, MSE):

$$E_{MSE} = \frac{\sum_{j=1}^{n} (\tilde{y}^{(j)} - y^{(j)})^2}{n}, \tag{5.1}$$

gdzie:

n – liczba obiektów w zbiorze testowym,
$\tilde{y}^{(j)}$ – wartość przewidywana przez model dla i-tego przykładu testowego,
$y^{(j)}$ – rzeczywista wartość atrybutu decyzyjnego dla i-tego przykładu testowego.

Innym parametrem służącym do oceny modelu jest średnia kwadratowa błędów (ang. *Root Mean Square Error*, RMSE)

$$E_{RMSE} = \frac{\sqrt{\sum_{j=1}^{n} (\tilde{y}^{(j)} - y^{(j)})^2}}{n}, \tag{5.2}$$

przy oznaczeniach jak we wcześniejszym wzorze.

Często stosowanym błędem jest też średni błąd bezwzględny (ang. *Mean Absolute Error*, MAE). Zaletą tego wskaźnika jest większa odporność na obserwacje odstające.

$$E_{MAE} = \frac{\sum_{j=1}^{n} |\tilde{y}^{(j)} - \tilde{y}^{(j)}|}{n}. \tag{5.3}$$

5.2. Klasyfikacja

Niech
n – liczba przykładów testowych,
n_T – liczba poprawnie sklasyfikowanych przykładów testowych,
n_F – liczba błędnie sklasyfikowanych przykładów testowych.

Miara trafności klasyfikowania (dokładność) wynosi:

$$N_{ov} = \frac{n_T}{n}. \tag{5.4}$$

Łączny błąd klasyfikowania (ang. *overall error rate*) jest określony wzorem:

$$E_{ov} = 1 - N_{ov} = \frac{n_F}{n}. \tag{5.5}$$

Wydawałoby się, że w zadaniach klasyfikacyjnych to są jedyne możliwe miary, oparte na liczbie obiektów dobrze i źle zaklasyfikowanych. Tymczasem żeby ocenić w pełni pracę klasyfikatora, należy utworzyć tzw. **macierz pomyłek**, w której rozróżniamy nie tylko sumę dobrze i sumę źle zaklasyfikowanych przypadków, ale wyszczególniamy sumy dobrze i źle zaklasyfikowanych przypadków dla każdej klasy oddzielnie. Przykłady macierzy dla klasyfikacji binarnej i wieloklasowej przedstawiają tabele 5.1–5.3.

Tabela 5.1 odpowiada modelowi podziału na dwie klasy, klasę „1" i „0". Przypadki dobrze zaklasyfikowane to te, w których wartość predykowana przez model odpowiada rzeczywistej wartości atrybutu decyzyjnego. Pamiętajmy, że klasyfikacja jest uczeniem nadzorowanym, czyli zarówno w zbiorze treningowym, jak i testowym dysponujemy obiektami, dla których określono wartość atrybutu decyzyjnego. Zatem podczas testów wytrenowanego modelu wiemy, jaka jest wartość rzeczywista atrybutu decyzyjnego, i porównujemy tę wartość z wartością przewidywaną, czyli przydzielaną przez model danemu przypadkowi testowemu.

Tabela 5.1. Macierz pomyłek w klasyfikacji binarnej

Wartość przewidywana	Wartość rzeczywista	
	1	0
1	n_{TP} prawdziwie pozytywne	n_{FP} fałszywie pozytywne
0	n_{FN} fałszywie negatywne	n_{TN} prawdziwie negatywne

Bardzo ważne jest, aby zwrócić uwagę na przypadki fałszywie pozytywne i fałszywie negatywne, które skojarzyć należy ze znanymi ze statystyki matematycznej pojęciami błędów pierwszego i drugiego rodzaju. Jak pamiętamy, błąd pierwszego rodzaju to prawdopodobieństwo odrzucenia hipotezy zerowej, która w rzeczywistości jest prawdziwa. Natomiast błąd drugiego rodzaju oznacza prawdopodobieństwo nieodrzucenia hipotezy zerowej, która w rzeczywistości jest fałszywa. Kluczowe jest to, że nie możemy kontrolować obydwu błędów równocześnie i musimy zdecydować o tym, który przypadek jest w badanym problemie mniej niebezpieczny. Niestety nie ma tu uniwersalnej odpowiedzi. Zastanówmy się przykładowo, czy wolelibyśmy, żeby nasz filtr antyspamowy w skrzynce mailowej „przepuszczał" na co dzień więcej spamu, ale nie ryzykował błędnego zaklasyfikowania jako niechcianej żadnej z potrzebnych nam wiadomości, czy jednak lepiej, żeby raz na jakiś czas potrzebna nam informacja trafiła do folderu spam, ale za to na co dzień otrzymywalibyśmy mniej niechcianych wiadomości, które prawidłowo zostałyby zaklasyfikowane jako spam? Dylematy te są jeszcze trudniejsze, gdy przychodzi nam rozwiązywać problemy np. z obszaru medycyny i klasyfikowania pacjentów do leczenia.

Aby łatwiej było porównywać efektywność poszczególnych klasyfikatorów na podstawie macierzy pomyłek, zdefiniowano cztery wskaźniki:

- czułość (ang. *True Positive Rate*, TPR),
- swoistość zwana również specyficznością (ang. *True Negative Rate*, TNR),
- dwie miary precyzji (ang. *Positive Predictive Value* PPV, *Negative Predictive Value*, NPV).

$$TPR = \frac{n_{TP}}{n_{TP} + n_{FN}}, \tag{5.6}$$

$$TNR = \frac{n_{TN}}{n_{TN} + n_{FP}}, \tag{5.7}$$

$$PPV = \frac{n_{TP}}{n_{TP} + n_{FP}}, \tag{5.8}$$

$$NPV = \frac{n_{TN}}{n_{TN} + n_{FN}}. \tag{5.9}$$

Współczynniki te najlepiej zwizualizować w macierzy pomyłek (tab. 5.2).

Tabela 5.2. Macierz pomyłek ze wskaźnikami w klasyfikacji binarnej

Wartość przewidywana	Wartość rzeczywista		
	1	0	
1	n_{TP} prawdziwie pozytywne	n_{FP} fałszywie pozytywne	$PPV = \frac{n_{TP}}{n_{TP} + n_{FP}}$
0	n_{FN} fałszywie negatywne	n_{TN} prawdziwie negatywne	$NPV = \frac{n_{TN}}{n_{TN} + n_{FN}}$
	$TPR = \frac{n_{TP}}{n_{TP} + n_{FN}}$	$TNR = \frac{n_{TN}}{n_{TN} + n_{FP}}$	

Czułość (TPR) oznacza odsetek obiektów klasy prawdziwie pozytywnej („1”) zaklasyfikowany jako klasa pozytywna. Swoistość (TNR) natomiast to odsetek obiektów, które w rzeczywistości były negatywne („0”) i zostały zaklasyfikowane jako klasa negatywna. Oba parametry mają swoje odpowiedniki:

– FNR = 1 – TPR, czyli odsetek obiektów klasy prawdziwie pozytywnej fałszywie zaklasyfikowanych jako klasa negatywna,
– FPR = 1 – TNR, czyli odsetek obiektów klasy prawdziwie negatywnej fałszywie zaklasyfikowanych jako klasa pozytywna.

Precyzja przewidywania pozytywnego (PPV) pokazuje, jaki odsetek obiektów zaklasyfikowanych jako pozytywne było faktycznie pozytywnych. Precyzja przewidywania negatywnego (NPV) wskazuje, jaki odsetek obiektów zaklasyfikowanych jako negatywne był faktycznie negatywnych. Wartości wskaźników mieszczą się w przedziale [0, 1] i od każdego z tych wskaźników oczekujemy oczywiście wartości jak najbliższych 1.

Macierz pomyłek dla klasyfikacji wieloklasowej (tab. 5.3) jest uogólnieniem przedstawionych wyżej pojęć, ale zauważmy, że poprawnie zaklasyfikowane przypadki mieszczą się tylko na przekątnej tej macierzy.

Tabela 5.3. Macierz pomyłek w klasyfikacji wieloklasowej

Wartość przewidywana	Wartość rzeczywista			
	y_1	y_2	…	y_M
y_1	n_{11}	n_{12}	…	n_{1M}
y_1	n_{21}	n_{22}	…	n_{2M}
…	…	…	…	…
y_M	n_{M1}	n_{M2}	…	n_{MM}

Bazując na macierzy pomyłek klasyfikacji wieloklasowej, możemy uogólnić wprowadzone wcześniej wzory na miarę trafności klasyfikowania i łączny błąd klasyfikowania dla dowolnej liczby klas $M \in N$.

$$N_{ov} = \frac{\sum_{i=1}^{M} n_{ii}}{n}, \tag{5.10}$$

$$E_{ov} = 1 - N_{ov} = \frac{\sum_{i=1,\ldots,M} \sum_{j=1,\ldots,M:i\neq j} n_{ij}}{n}. \tag{5.11}$$

5.3. Klasteryzacja

Metody wykorzystywane do oceny modeli klasteryzacji bazują na założeniu, że dobre grupowanie charakteryzuje się wysokim podobieństwem obiektów zaklasyfikowanych do tej samej grupy oraz małym podobieństwem obiektów z różnych grup.

Najczęściej stosowaną funkcją oceny grupowania jest wskaźnik fitness.

$$fitness(C) = \sum_{k=1,\ldots,M} \sum_{j=1,\ldots,N} d(x^{(j)}, c_k) \cdot u_{jk}, \tag{5.12}$$

gdzie $d(x^{(j)}, c_k)$ – odległość j-tego obiektu od k-tego centroidu według ustalonej miary odległości d.

Oczywiście im mniejsza wartość wskaźnika, tym lepszy podział, ponieważ obiekty w poszczególnych grupach są sobie bliższe.

UWAGA

Wskaźnik ten nie może być stosowany dla algorytmów, w których liczba klastrów nie jest ustalona a priori, ponieważ zwykle: $fitness(C_2) < fitness(C_1)$, gdy $|C_2| < |C_1|$, czyli algorytm na podstawie takiej miary sukcesu będzie dążył do podziałów na jak najmniej liczne zbiory i oczywiście „wygra" podział, w którym każdy klaster będzie zawierał tylko jedną obserwację.

Inną miarą jakości grupowania jest wskaźnik sylwetkowy grup, zwany inaczej miarą wewnętrzną (ang. *silhouette index*). Wskaźnik ten najpierw oblicza się dla każdego obiektu ze wzoru:

$$Silhouette(x^{(j)}) = \frac{b-a}{\max(a,b)}, \tag{5.13}$$

gdzie:

- a – średnia odległość $x^{(j)}$ od pozostałych obiektów zaklasyfikowanych do tego samego klastra,
- b – średnia odległość $x^{(j)}$ od obiektów zaklasyfikowanych do najbliższego klastra,
- $\max(a, b)$ – większa z wartości a i b.

Wtedy wskaźnik sylwetkowy dla podziału wynosi:

$$Silhouette(C) = \frac{\sum_{j=1}^{N} Silhouette(x^{(j)})}{N}. \tag{5.14}$$

6 Konfiguracja środowiska programistycznego

Wszystkie przykłady praktyczne i zadania prezentowane w niniejszej monografii zostały napisane w języku Python. Podczas opracowania materiałów wykorzystano następujące wersje bibliotek i modułów:

- ignite == 0.4
- ipywidgets == 7.6
- librosa == 0.9
- lightgbm == 3.3
- matplotlib == 3.5
- opensmile == 2.2
- optuna == 2.10
- pandas == 1.4
- pydub == 0.24
- pytables == 3.7
- python == 3.9
- pytorch == 1.11
- scikit-learn == 1.0
- scipy == 1.8
- sox == 1.4
- spafe == 0.1
- xgboost == 1.3

UWAGA

Biblioteka pytables jest biblioteką używaną przez system operacyjny Windows. Jej odpowiednikiem używanym przez system Linux jest biblioteka tables.

Rekomendowaną dystrybucją Pythona jest Anaconda, zawierająca większość niezbędnych bibliotek i umożliwiająca łatwą instalację brakujących modułów, która znajduje się pod adresem: https://www.anaconda.com/products/individual.

UWAGA

Użytkownicy systemu Linux muszą samodzielnie zainicjalizować Anacondę. Można to zrobić albo podczas instalacji (pojawi się pytanie, czy użytkownik chce to zrobić), albo później przy użyciu komend:

```
source <path to conda>/bin/activate
conda init
```

W miejsce <path to conda> należy wpisać ścieżkę dostępu do folderu, w którym zainstalowana jest Anaconda, np. /home/user/anaconda3/.

UWAGA

W języku polskim separatorem dziesiętnym jest przecinek, dlatego też w tekście ułamki dziesiętne podawane są zgodnie z tym formatem. Jednak w kodach źródłowych należy używać kropki zgodnie ze składnią języka Python.

7 Parametryzacja sygnału i ekstrakcja cech

Parametryzacja sygnału jest niezmiernie istotnym etapem przygotowania danych – komputery, a tym samym zaimplementowane algorytmy, operują na liczbach, nie zaś na abstrakcyjnych, bardzo często subiektywnych cechach sygnału, które potrafi wychwycić człowiek. Aby sygnał mógł być wykorzystany przez algorytm uczenia maszynowego, konieczne jest zatem jego opisanie w sposób obiektywny, sformalizowany matematycznie. Proces, który do tego prowadzi, nazywany jest ekstrakcją cech (ang. *feature extraction*).

W języku Python indeksowanie zaczyna się od 0, zatem pierwszy element dowolnej struktury danych, np. listy, tablicy czy macierzy, ma indeks 0, drugi element indeks 1 itd. W przedstawionych w dalszej części rozdziału wzorach matematycznych będzie stosowana taka sama konwencja.

Parametry sygnałów można podzielić na cztery podstawowe grupy:

1) parametry czasowe,
2) parametry widmowe,
3) parametry formantowe,
4) parametry czasowo-częstotliwościowe.

UWAGA

Sygnały zapisane w formacie .wav mogą zostać wczytane m.in. z wykorzystaniem biblioteki scipy (funkcja scipy.io.wavfile.read) oraz biblioteki librosa (funkcja librosa.load) – przykładowe użycie obu bibliotek i funkcji zostanie przedstawione w dalszej części rozdziału.

Po wczytaniu plików formatu .wav przy użyciu biblioteki scipy zostanie uzyskany wektor danych typu int, natomiast po wczytaniu ich przy użyciu biblioteki librosa zostanie uzyskany wektor danych typu float. Jeżeli sygnał został wczytany funkcją z biblioteki scipy i ma być on podany jako argument do funkcji z biblioteki librosa, to wówczas należy odpowiednio zmienić typ danych, np. używając metody signal.astype(numpy.float32).

7.1. Parametry czasowe

Parametry czasowe sygnału wyznaczane są na podstawie jego przebiegu czasowego. Do najczęściej używanych parametrów zaliczają się (Tkach i in., 2010):

- energia sygnału,
- wartość skuteczna,
- obwiednia sygnału,
- środek ciężkości obwiedni sygnału,
- częstotliwość przejść przez zero.

Energia sygnału dana jest wzorem:

$$E = \sum_{t=0}^{T-1} x^2[t], \tag{7.1}$$

gdzie:

T – długość sygnału,
$x[t]$ – wartość sygnału w chwili t.

Wartość skuteczną (ang. *Root Mean Square*, RMS) opisuje wzór:

$$RMS = \sqrt{\frac{1}{T}\sum_{t=0}^{T-1} x^2[t]}, \tag{7.2}$$

gdzie:

T – długość sygnału,
$x[t]$ – wartość sygnału w chwili t.

Obwiednia sygnału (ang. *envelope*) jest chwilową wartością amplitudy w funkcji czasu, zmieniającą się znacznie wolniej od samego sygnału (rys. 7.1) (Yang, 2017). Można ją zapisać następująco:

$$O[k] = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} x_k^2[n]}, \tag{7.3}$$

gdzie:

N – długość ramki sygnału,
k – numer ramki sygnału,
$x_k[n]$ – wartość sygnału w k-tej ramce w chwili n.

Środek ciężkości obwiedni sygnału (ang. *Temporal Centroid*, TC) wyznacza się z równania:

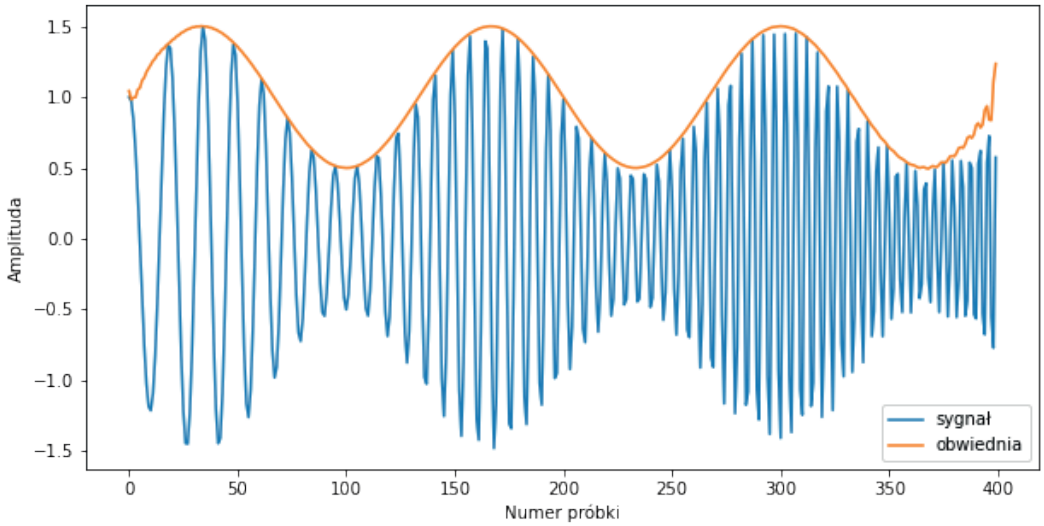
$$TC = \frac{\sum_{k=0}^{K-1}(k+1)\cdot O[k]}{\sum_{k=0}^{K-1} O[k]}, \tag{7.4}$$

gdzie:

O – obwiednia sygnału,
K – liczba ramek sygnału.

```
#kod realizujący ekstrakcję obwiedni sygnału i jej środka ciężkości

from scipy.signal import hilbert, chirp
import matplotlib.pyplot as plt
import numpy as np.

#generowanie sygnału
duration = 1.0
fs = 400.0
samples = int(fs*duration)
t = np.arange(samples) / fs
signal = chirp(t, 20.0, t[-1], 100.0)
signal *= (1.0 + 0.5 * np.sin(2.0*np.pi*3.0*t))

#obwiednia sygnału
analytic_signal = hilbert(signal)
amplitude_envelope = np.abs(analytic_signal)
fig = plt.figure(figsize = (10,5))
plt.plot(signal, label = 'sygnał')
plt.plot(amplitude_envelope, label = 'obwiednia')
plt.legend()
```

```
plt.ylabel('Amplituda')
plt.xlabel('Numer próbki')
plt.show()

#środek ciężkości obwiedni sygnału
TC = sum([index*value for index, value in enumerate(
                                amplitude_envelope)])/sum(amplitude_envelope)
```



Rysunek 7.1. Przebieg czasowy oraz obwiednia sygnału

Częstotliwość przejść przez zero (ang. *Zero-Crossing Rate*, ZCR, rys. 7.2) jest wyznaczana z wzoru:

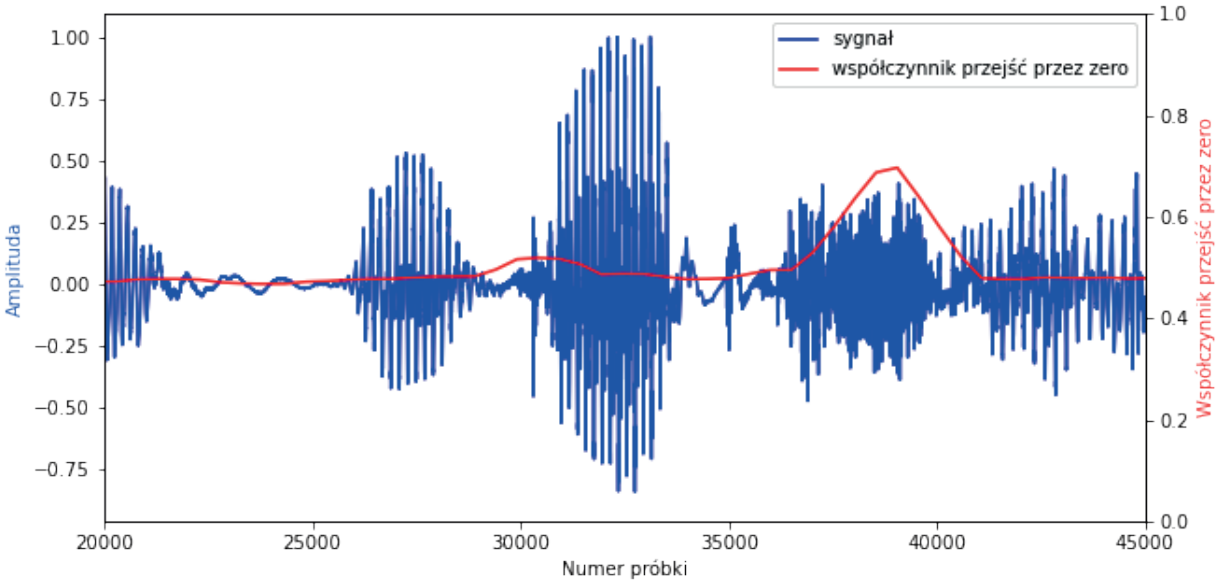
$$ZCR[k] = \frac{1}{2N}\sum_{n=1}^{N-1}\left|\mathrm{sgn}(x_k[n]) - \mathrm{sgn}(x_k[n-1])\right|, \tag{7.5}$$

gdzie:

k – numer ramki sygnału,
N – długość ramki sygnału,
$x_k[n]$ – wartość k-tej ramki sygnału w chwili n,
$\mathrm{sgn}(x)$ – funkcja signum.

```
#kod umożliwiający wyznaczenie współczynnika przejść przez zero

from librosa.feature import zero_crossing_rate
import librosa
import matplotlib.pyplot as plt
import numpy as np
signal, fs = librosa.load('signal.wav', sr = 22050, mono = True)
zcr = zero_crossing_rate(signal, frame_length = 2048,
                                 hop_length = 512,
                                 center = True).reshape(-1)
x_zcr = np.arange(0,len(signal), np.int32(len(signal)/len(zcr)))
fig, ax1 = plt.subplots(figsize = (10,5))
```

```
ax2 = ax1.twinx()
ax1.plot(signal, 'b-', label = 'sygnał')
ax1.plot(x_zcr, zcr, 'r-', label = 'współczynnik przejść przez zero')
plt.xlim(20000,45000)
ax1.legend()
ax1.set_ylabel('Amplituda', label = 'b')
ax2.set_ylabel('Współczynnik przejść przez zero', label = 'r')
ax1.set_xlabel('Numer próbki')
plt.show()
```



Rysunek 7.2. Przebieg czasowy sygnału mowy ciągłej oraz wyznaczony na jego podstawie współczynnik przejść przez zero w kolejnych ramkach

7.2. Parametry widmowe

Parametry widmowe sygnału wyznaczane są na podstawie estymaty jego widma. Do najczęściej stosowanych parametrów widmowych zaliczają się (Boashash, 2015):

- momenty widmowe,
- indeks spadku,
- płaskość widmowa.

Momenty widmowe są to deskryptory cech sygnału wykorzystywane do badania podobieństwa pomiędzy obiektami, obliczane na podstawie estymaty widma sygnału. Wyróżnia się momenty widmowe, momenty widmowe unormowane oraz momenty widmowe unormowane centralne.

Moment widmowy n-tego rzędu dany jest wzorem:

$$M(n) = \sum_{k=0}^{\infty} \left|G(k)\right| \cdot [f_k]^n, \tag{7.6}$$

gdzie:

$G(k)$ – wartość widma mocy k-tego pasma częstotliwościowego,
f_k – częstotliwość środkowa tego pasma.

Moment widmowy unormowany n-tego rzędu dany jest wzorem:

$$M_u(n) = \frac{M(n)}{M(0)}. \tag{7.7}$$

Moment widmowy unormowany centralny n-tego rzędu dany jest wzorem:

$$M_{uc}(n) = \sum_{k=0}^{\infty} \frac{\sum_{k=0}^{\infty} |G(k)| \cdot [f_k - M_u(1)]^2}{M(0)}. \tag{7.8}$$

Moment unormowany pierwszego rzędu może być interpretowany jako środek ciężkości widma, moment unormowany centralny drugiego rzędu jako kwadrat szerokości widma, zaś moment unormowany centralny trzeciego rzędu jako skośność widma. Ponadto momenty unormowane centralne drugiego i czwartego rzędu mogą posłużyć do wyznaczenia kurtozy widma zgodnie ze wzorem:

$$kurtosis = \frac{M_{uc}(4)}{[M_{uc}(2)]^2}. \tag{7.9}$$

Indeks spadku (ang. *rolloff factor*) jest to parametr określający częstotliwość graniczną definiowaną w taki sposób, że określona część mocy sygnału przypada na częstotliwości nie większe od częstotliwości granicznej.

```
#kod służący do wyznaczenia indeksu spadku

import librosa
from librosa.feature import spectral_rolloff

signal, fs = librosa.load('signal.wav', sr = 16000, mono = True)
rolloff = spectral_rolloff(y = signal, sr = fs, roll_percent = 0.85)
```

Płaskość widmowa (ang. *Spectral Flatness Measure*, SFM) jest to stosunek średniej geometrycznej i arytmetycznej współczynników widma:

$$SFM = 10\log\left\{\frac{\left[\prod_{k=1}^{N/2} P\left(e^{j\frac{2\pi k}{N}}\right)\right]^{\frac{1}{N/2}}}{\frac{1}{N/2}\sum_{k=1}^{N/2} P\left(e^{j\frac{2\pi k}{N}}\right)}\right\} \tag{7.10}$$

```
#kod służący do wyznaczenia płaskości widmowej

import librosa
from librosa.feature import spectral_flatness

signal, fs = librosa.load('signal.wav', sr = 16000, mono = True)
flatness = spectral_flatness(y = signal, n_fft = 2048, hop_length = 512)
```

7.3. Parametry formantowe

Parametry formantowe to rodzaj parametrów wyznaczanych z sygnału mowy, śpiewu lub nagrań instrumentów muzycznych. Formantem nazywane jest pasmo częstotliwości, w obrębie którego wszystkie składowe ulegają widocznemu wzmocnieniu w stosunku do częstotliwości zawartych w pozostałych pasmach, lub maksimum lokalne widma sygnału (Abhang i in., 2016). Od amplitudy i częstotliwości kolejnych formantów zależna jest barwa dźwięku. Formant o najmniejszej częstotliwości nazywany jest formantem pierwszym i oznaczany jest jako F_1, formant drugi oznaczany jest symbolem F_2 itd.

W przypadku sygnału mowy formanty wyznacza się najczęściej dla fragmentów sygnału odpowiadających samogłoskom – każda samogłoska charakteryzuje się określonym przedziałem częstotliwości formantowych, więc mogą one służyć do identyfikacji samogłosek. Zazwyczaj w tym celu wystarczające jest wyznaczenie częstotliwości dwóch pierwszych formantów.

Częstotliwości trzech pierwszych formantów można w prosty sposób wyznaczyć za pomocą biblioteki OpenSMILE (Eyben i in., 2010).

```
#kod służący do wyznaczenia parametrów formantowych

import opensmile
smile = opensmile.Smile(
    feature_set = opensmile.FeatureSet.GeMAPSv01b,
    feature_level = opensmile.FeatureLevel.LowLevelDescriptors)
result_df = smile.process_file('signal.wav')
centerformantfreqs = ['F1frequency_sma3nz',
                      'F2frequency_sma3nz',
                      'F3frequency_sma3nz']
formant_df = result_df[centerformantfreqs]
```

UWAGA

Maksimum lokalne o największej amplitudzie oraz najmniejszej częstotliwości widoczne w widmie sygnału mowy to tzw. ton podstawowy. Pomimo że oznaczany jest symbolem F_0, nie jest on formantem.

Wartość częstotliwości tonu podstawowego decyduje o wysokości głosu danej osoby. W przypadku mężczyzn ton podstawowy najczęściej przyjmuje wartości z przedziału 85–155 Hz, podczas gdy w przypadku kobiet wartości z przedziału 165–255 Hz.

Częstotliwość tonu podstawowego można wyznaczyć przy użyciu biblioteki librosa.

```
#kod służący do wyznaczenia częstotliwości tonu podstawowego przy użyciu
#biblioteki librosa

import librosa
import numpy as np

signal, fs = librosa.load('signal.wav', sr = 16000, mono = True)
fmin = librosa.note_to_hz('C2') #~65 Hz
fmax = librosa.note_to_hz('C7') #~2093 Hz
f0, voiced_flag, voiced_probs = librosa.pyin(signal, fmin, fmax, sr = fs)
f0 = np.mean(f0[voiced_flag == True])
```

Alternatywną metodą jest skorzystanie z biblioteki OpenSMILE.

```
#kod służący do wyznaczenia częstotliwości tonu podstawowego przy użyciu
#biblioteki OpenSMILE

import opensmile
smile = opensmile.Smile(
    feature_set = opensmile.FeatureSet.ComParE_2016,
    feature_level = opensmile.FeatureLevel.LowLevelDescriptors)
result_df = smile.process_file('signal.wav')
f0freq = ['F0final_sma']
formant_df = result_df[f0freq]
```

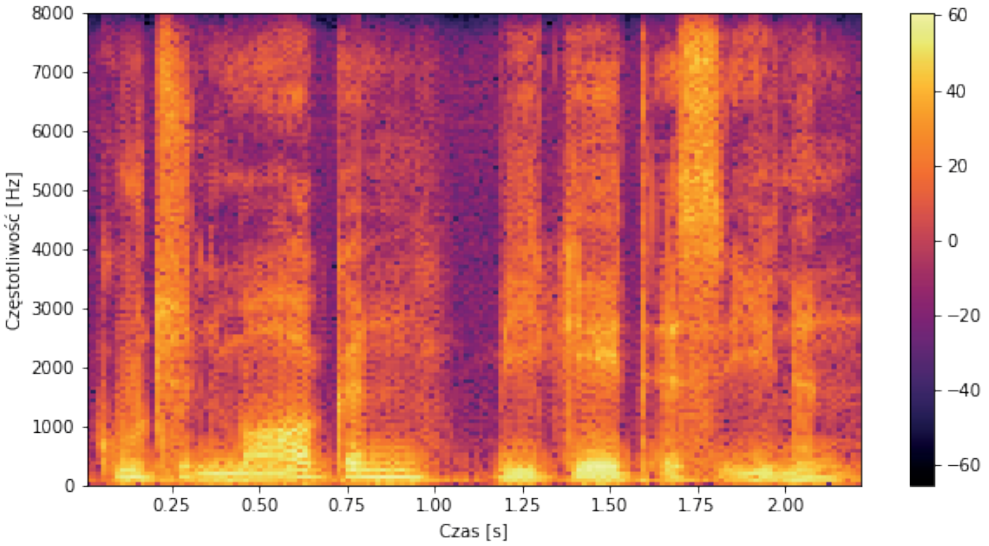
7.4. Parametry czasowo-częstotliwościowe

Parametry czasowo-częstotliwościowe wyznaczane są na podstawie widma sygnału w kolejnych odcinkach przebiegu czasowego (Boashash, 2015). Najczęściej wykorzystywana jest w tym celu tzw. krótkoczasowa transformata Fouriera (ang. *Short-Time Fourier Transform*, STFT). W wyniku jej działania uzyskiwana jest trójwymiarowa reprezentacja sygnału, której wykres nazywany jest spektrogramem. Zwyczajowo na osi *X* oznaczane są kolejne ramki sygnału, zaś na osi *Y* pasma częstotliwości. Mogą być one przeliczone na czas oraz częstotliwość, lecz nie muszą – równie często podawane są po prostu numery ramek i pasm. Wartości amplitud widma (w przypadku widma amplitudowego) lub mocy sygnału (w przypadku widma mocy) przedstawione są z użyciem skali barwnej (rys. 7.3).

W Pythonie spektrogram można wyznaczyć m.in. za pomocą biblioteki scipy. W tym celu należy podać do funkcji sygnał (oznaczenie fs), z którego wyznaczony zostanie spektrogram, oraz częstotliwość próbkowania tego sygnału. Ponadto duży wpływ na uzyskiwane wyniki mają parametry takie jak rozmiar okna wyrażony w liczbie próbek (oznaczenie nfft), funkcja okna (ang. *window*) oraz wielkość nakładkowania (ang. *noverlap*). Pełen opis funkcji wyznaczającej spektrogram sygnału oraz argumentów, jakie ona przyjmuje, można znaleźć pod adresem: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.spectrogram.html.

W celu uzyskania bardziej czytelnych wykresów wartości spektrogramu często wyraża się w decybelach.

```
#kod służący do wyznaczenia spektrogramu

from scipy.io.wavfile import read as read_wav
from scipy.signal import spectrogram
import matplotlib.pyplot as plt
import numpy as np.
fs, signal = read_wav('signal.wav')
f, t, spec = spectrogram(signal, fs)
fig = plt.figure(figsize = (10,5))
spec_plot = plt.pcolormesh(t, f, 10*np.log10(spec), cmap = 'inferno')
fig.colorbar(spec_plot)
#10*np.log10(spec) - przeliczenie na decybele
plt.ylabel('Częstotliwość [Hz]')
plt.xlabel('Czas [s]')
plt.show()
```



Rysunek 7.3. Przykładowy spektrogram wyznaczony z sygnału mowy ciągłej. Przebieg czasowy sygnału przedstawiono na rysunku 7.2

W przypadku sygnałów akustycznych o częstotliwościach mieszczących się w zakresie słyszalności, zazwyczaj sygnałów mowy czy muzyki, często stosowane są tzw. skale perceptualne. Są to skale bazujące na modelach psychoakustycznych układu słuchowego człowieka i mają na celu przetworzenie sygnału akustycznego w taki sposób, by imitować jego przetwarzanie przez ucho wewnętrzne. W tym celu wykorzystywane są tzw. banki filtrów, czyli zbiory filtrów słuchowych o parametrach zbliżonych do pasm krytycznych błony podstawnej ślimaka.

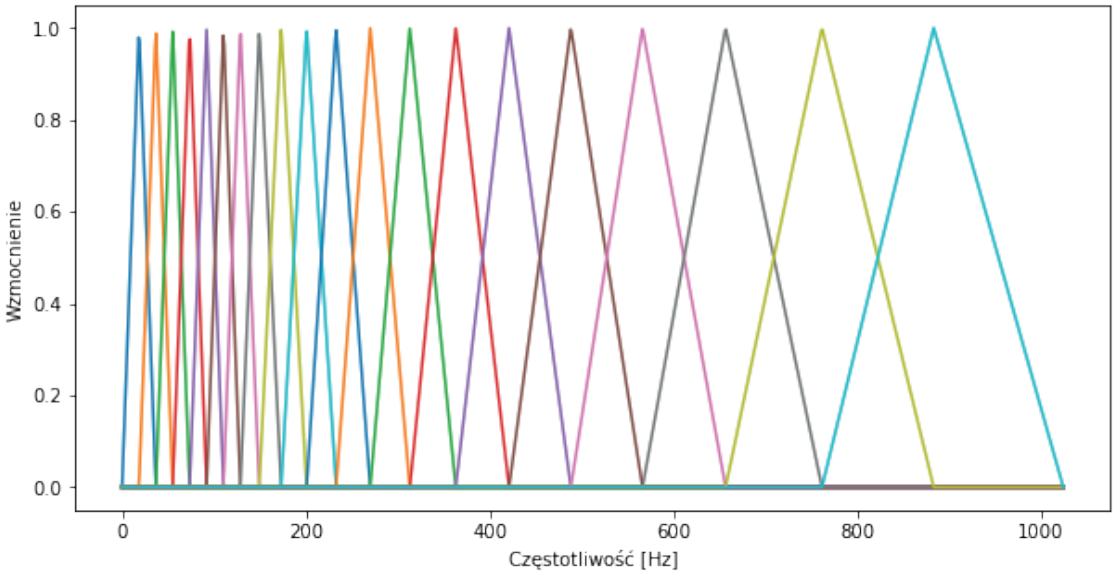
Najczęściej wykorzystywanym bankiem filtrów jest bank filtrów melowych służący do modelowania nieliniowości percepcji wysokości dźwięku. Filtry melowe mają kształt trójkątny i uwzględniają różnice w rozdzielczości częstotliwościowej układu słuchowego człowieka, która jest większa dla niskich częstotliwości oraz niższa dla wysokich (rys. 7.4). Bazują one na skali melowej, której zależność od skali częstotliwościowej wyrażonej w hercach określona jest wzorem:

$$mel = 2596 \cdot \log 10\left(1 + \frac{f}{700}\right), \tag{7.11}$$

gdzie f – częstotliwość [Hz].

```
#kod służący do wyznaczenia i wyświetlenia banku filtrów melowych

import librosa
import librosa.display
import matplotlib.pyplot as plt
import numpy as np
mels = librosa.filters.mel(sr = 16000,
                           n_fft = 2048,
                           n_mels = 20,
                           fmin = 0.0,
                           fmax = None,
                           norm = None)
fig = plt.figure(figsize = (10,5))
```

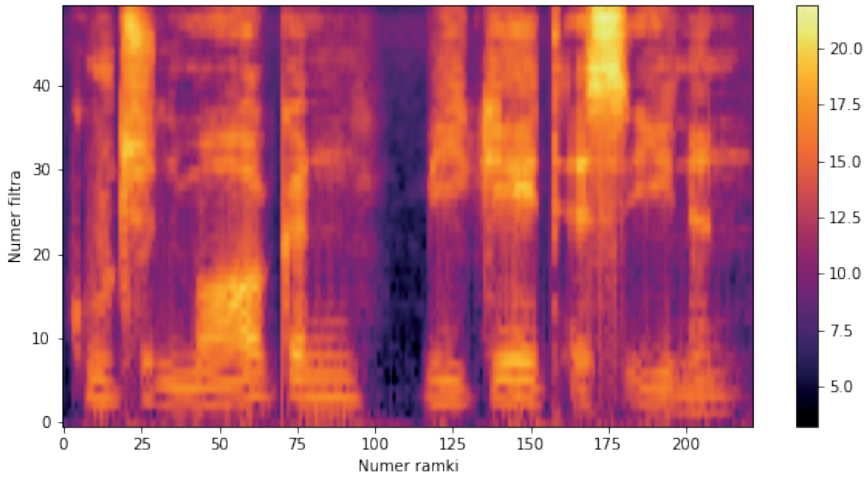
```
plt.plot(mels.T)
plt.xlabel('Częstotliwość [Hz]')
plt.ylabel('Wzmocnienie')
plt.show()
```



Rysunek 7.4. Bank filtrów melowych

Banki filtrów mogą być wykorzystane do ekstrakcji dwóch rodzajów cech. Pierwszym z nich są cechy w języku angielskim nazywane *filterbank features* (rys. 7.5). Są one wyznaczane w wyniku filtracji spektrogramów bankiem filtrów (zazwyczaj melowych) (Ravindran i in., 2003). W Pythonie można tę operację wykonać, korzystając z poniższego kodu.

```
#kod służący do wyznaczenia melspektrogramu

from python_speech_features import logfbank
from scipy.io.wavfile import read as read_wav
import matplotlib.pyplot as plt
fs,signal = read_wav('signal.wav')
filterbank_feats = logfbank(signal, samplerate = fs,
                                    nfilt = 50,
                                    winlen = 0.025,
                                    winstep = 0.01)
fig, ax = plt.subplots(figsize = (10,5))
#origin = 'lower' - piksel (0,0) ma znajdować się w lewym dolnym rogu
filterbank_feats_plot = ax.imshow(filterbank_feats.T, cmap = 'inferno',
                                                      aspect = 'auto',
                                                      origin = 'lower')
fig.colorbar(filterbank_feats_plot)
plt.ylabel('Numer filtra')
plt.xlabel('Numer ramki')
plt.show()
```



Rysunek 7.5. Spektrogram sygnału mowy ciągłej przedstawiony na rysunku 7.3 przefiltrowany przy użyciu banku filtrów melowych

Drugim rodzajem cech są meloczęstotliwościowe współczynniki cepstralne (ang. *Mel-Frequency Cepstral Coefficients*, MFCC), czyli wektory współczynników cepstrum sygnału w poszczególnych pasmach melowych (rys. 7.6). Istotną zaletą współczynników MFCC jest ich mała wrażliwość na szum.

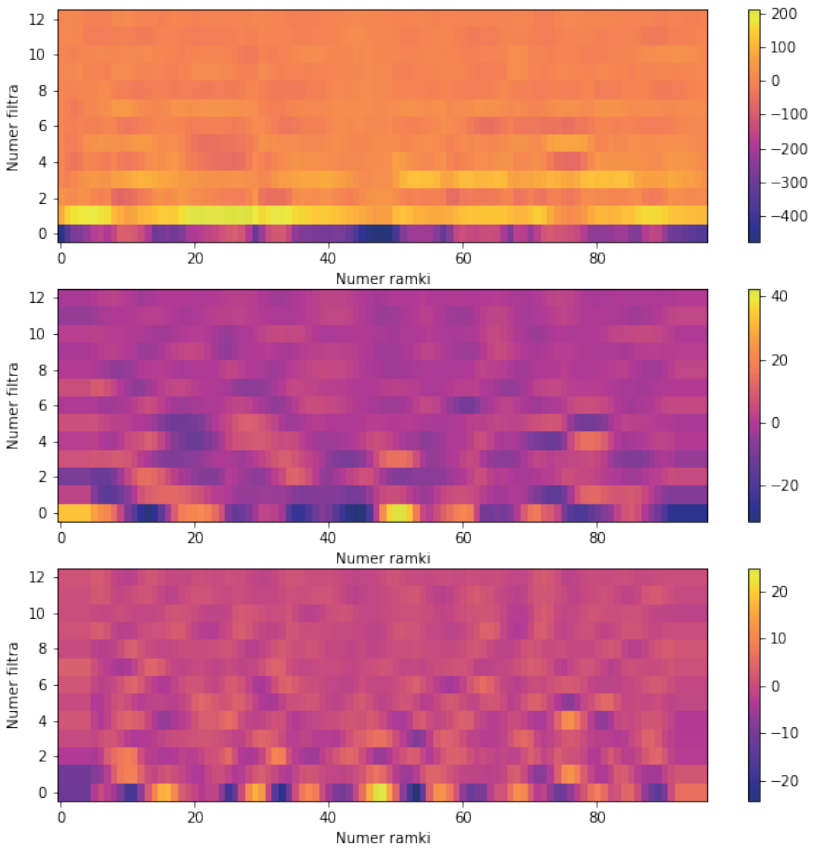
W połączeniu ze współczynnikami MFCC często wykorzystuje się ich pierwszą i drugą pochodną, zwane również deltami i oznaczane symbolami Δ oraz Δ^2 lub $\Delta\Delta$. Pochodne współczynników pozwalają w prosty sposób opisać dynamikę sygnału przy stosunkowo niewielkich kosztach obliczeniowych.

Wyznaczanie współczynników MFCC

1. Filtracja filtrem preemfazy.
2. Podział sygnału na ramki.
3. Wyznaczenie dyskretnej transformaty Fouriera każdej ramki sygnału.
4. Filtracja widma bankiem filtrów melowych.
5. Obliczenie logarytmu energii w poszczególnych pasmach częstotliwościowych.
6. Wyznaczenie dyskretnej transformaty kosinusowej z logarytmów energii (Hosom, 2003).

```
#kod służący do wyznaczenia współczynników MFCC

import librosa
from librosa.feature import mfcc as extract_mfcc
from librosa.feature import delta as extract_mfcc_deltas
import matplotlib.pyplot as plt

signal, fs = librosa.load('signal.wav', sr = 22050, mono = True)

#MFCC
mfcc = extract_mfcc(signal, sr = fs, n_mfcc = 13)

#pochodne MFCC
delta = extract_mfcc_deltas(mfcc, order = 1)
delta2 = extract_mfcc_deltas(mfcc, order = 2)

fig, (ax0, ax1, ax2) = plt.subplots(nrows = 3, figsize = (10,10))
```

```
mfcc_plot = ax0.imshow(mfcc, cmap = 'plasma',
                             aspect = 'auto',
                             origin = 'lower')
ax0.set_ylabel('Numer filtra')
ax0.set_xlabel('Numer ramki')
fig.colorbar(mfcc_plot, ax = ax0)

delta_plot = ax1.imshow(delta, cmap = 'plasma',
                               aspect = 'auto',
                               origin = 'lower')
ax1.set_ylabel('Numer filtra')
ax1.set_xlabel('Numer ramki')
fig.colorbar(delta_plot, ax = ax1)

delta2_plot = ax2.imshow(delta2, cmap = 'plasma',
                                 aspect = 'auto',
                                 origin = 'lower')
ax2.set_ylabel('Numer filtra')
ax2.set_xlabel('Numer ramki')
fig.colorbar(delta2_plot, ax = ax2)

plt.show()
```



Rysunek 7.6. Melowe współczynniki cepstralne (na górze) oraz ich pierwsza (w środku) i druga pochodna (na dole) wyznaczone z sygnału mowy ciągłej, którego przebieg czasowy przedstawiono na rysunku 7.2

Alternatywnymi dla filtrów melowych są nieco rzadziej używane filtry gammatone (rys. 7.7), których odpowiedź impulsowa dana jest wzorem:

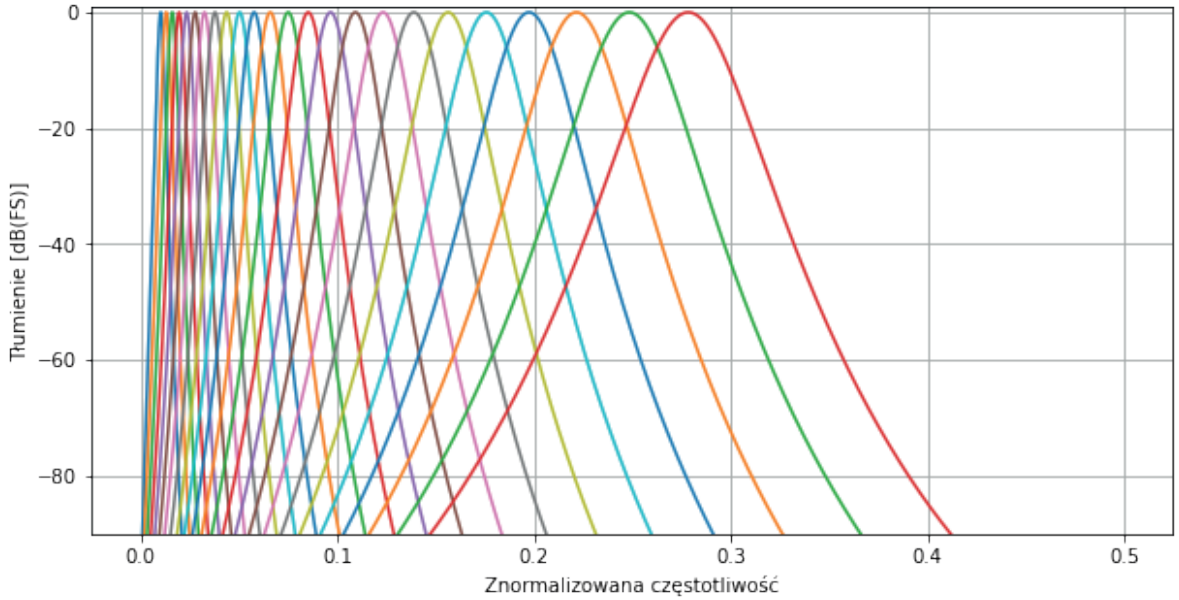
$$g(t) = at^{n-1}e^{-2\pi bt}\cos(2\pi ft + \phi), \tag{7.12}$$

gdzie:

- a – amplituda,
- b – współczynnik tłumienia,
- n – rząd filtra,
- f – częstotliwość,
- ϕ – faza.

Najważniejszą różnicą w stosunku do filtrów melowych jest fakt, że filtry gammatone są asymetryczne, przez co uznawane są za lepszą aproksymację rzeczywistych filtrów występujących w układzie słuchowym.

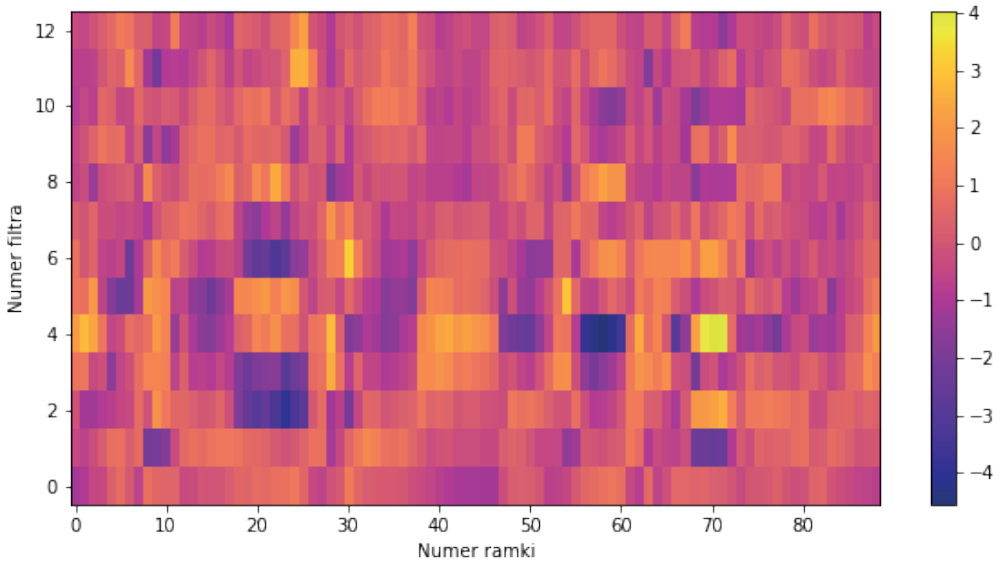
```
#kod służący do wyznaczenia i wyświetlenia banku filtrów gammatone

#potrzebna jest biblioteka pyfilterbank:
#https://github.com/SiggiGue/pyfilterbank
from pyfilterbank.gammatone import GammatoneFilterbank
import scipy

def example_filterbank():
  from pylab import plt
  import numpy as np

  x = scipy.signal.unit_impulse(2000)
  gfb = GammatoneFilterbank(samplerate = 16000,
                            normfreq = 1200,
                            density = 1)

  analyse = gfb.analyze(x)
  imax, slopes = gfb.estimate_max_indices_and_slopes()

  fig, ax = plt.subplots(figsize = (10,5))

  def plotfun(x, y):
    ax.plot(x, 20*np.log10(np.abs(y)**2))

  gfb.freqz(nfft = 2*4096, plotfun = plotfun)
  plt.grid(True)
  plt.xlabel('Znormalizowana częstotliwość')
  plt.ylabel('Tłumienie [dB(FS)]')
  plt.axis('Tight')
  plt.ylim([-90, 1])
  plt.show()

example_filterbank()
```



Rysunek 7.7. Bank filtrów gammatone

Bank filtrów gamma może zostać wykorzystany do wyznaczenia współczynników cepstralnych zwanych GFCC (ang. *Gammatone Frequency Cepstral Coefficients*, rys. 7.8) (Shao i in., 2009). Współczynniki te znajdują zastosowanie głównie w analizie sygnału mowy, w szczególności w algorytmach identyfikacji mówcy (Liu, 2018), są też szeroko stosowane w rozwiązywaniu problemu identyfikacji źródeł dźwięku znajdujących się pod wodą (Wang i in., 2019).

```
#kod służący do wyznaczenia współczynników GFCC

from spafe.features.gfcc import gfcc
import matplotlib.pyplot as plt
from scipy.io.wavfile import read as read_wav

fs, signal = read_wav('signal.wav')

gfcc_feats = gfcc(signal, fs = fs,
                          num_ceps = 13,
                          win_len = 0.025,
                          win_hop = 0.025)

fig, ax = plt.subplots(figsize = (10,5))

gfcc_plot = ax.imshow(gfcc_feats.T, cmap = 'plasma',
                                    aspect = 'auto',
                                    origin = 'lower')
plt.ylabel('Numer filtra')
plt.xlabel('Numer ramki')
plt.colorbar(gfcc_plot)
plt.show()
```



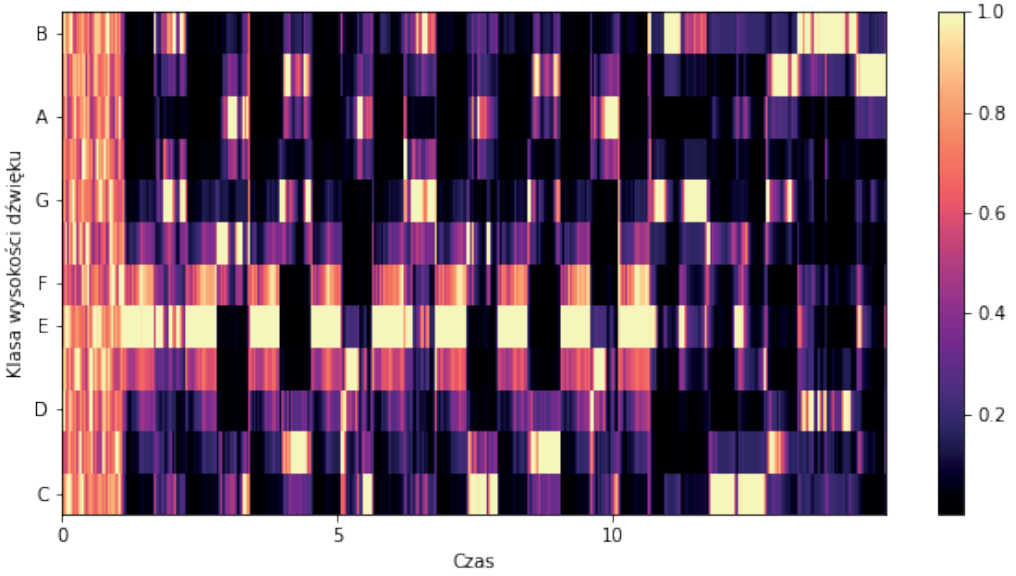
Rysunek 7.8. Współczynniki GFCC wyznaczone z sygnału mowy ciągłej, którego przebieg czasowy przedstawiono na rysunku 7.2

Ostatnie cechy czasowo-częstotliwościowe, które zostaną przedstawione, bazują na teorii muzyki i są wykorzystywane przede wszystkim w analizie i parametryzacji barwy dźwięku. Graficzne przedstawienie tych cech, czyli tzw. chromagram, to wykres wartości klas wysokości w funkcji czasu (rys. 7.9). Klasy wysokości oznaczają zbiór dźwięków o określonej wysokości półtonu należące do różnych oktaw, np. C, c, c^1, c^2 itd. (Yu i in., 2010). Jednym z najważniejszych zastosowań chromagramów jest rozpoznawanie utworów muzycznych.

UWAGA

Wartości na osi *Y* chromagramu wyświetlanego przy użyciu biblioteki librosa są domyślnie oznaczone angielskimi nazwami dźwięków. Dźwięk oznaczony jako „B" to dźwięk, który w języku polskim nazywany jest „h".

```
#kod służący do wyznaczenia chromagramu sygnału

from librosa.feature import chroma_stft
import librosa
import librosa.display
import matplotlib.pyplot as plt

signal, fs = librosa.load('signal.wav', sr = 44100, mono = True)
chroma = librosa.feature.chroma_stft(y = signal, sr = fs)

fig = plt.figure(figsize = (10,5))
chroma_plot = librosa.display.specshow(chroma, y_axis = 'chroma',
                                                x_axis = 'time')
plt.colorbar(chroma_plot)
plt.xlabel('Czas')
plt.ylabel('Klasa wysokości dźwięku')
plt.show()
```



Rysunek 7.9. Chromagram wyznaczony dla fragmentu utworu z baletu „Dziadek do orzechów"

Na koniec należy wspomnieć, że ekstrakcja wyżej opisanych cech nie jest jedynym podejściem stosowanym w analizie sygnałów. Innym równie przydatnym rozwiązaniem jest wykorzystanie metod używanych w analizie szeregów czasowych, w szczególności służących do prognozowania wartości przyjmowanej przez analizowany wskaźnik bądź parametr. Należy jednak zauważyć, że większość powszechnie stosowanych metod opisu szeregów czasowych opartych na estymacji stochastycznej jest wykorzystywana w prognozach średnio- i długoterminowych, lecz uzyskiwane przy ich użyciu rezultaty prognoz chwilowych są często niezadowalające. Rozwiązaniem tego problemu może być użycie metod wykorzystujących tzw. monotoniczną agregację szeregu, która pozwala na uzyskanie dokładnych prognoz w krótkim horyzoncie czasowym dzięki eliminacji z szeregu czasowego zbędnych danych pośrednich oraz wyznaczeniu jego obwiedni ekstremalnej (Skulimowski, 2002). Istotną zaletą tej metody jest fakt, że może być ona z powodzeniem użyta w badaniu nie tylko danych finansowych, do czego pierwotnie została opracowana, ale również innych dyskretnych procesów Markowa. Dzięki temu obszar jej zastosowań można rozszerzyć na wszystkie analizy, w których przyjmuje się, że analizowane zjawisko jest procesem Markowa. Do takich obszarów można zaliczyć m.in. analizę sygnału mowy czy przetwarzanie języka naturalnego (ang. *Natural Language Processing*, NLP). Innym wartym uwagi rozwiązaniem bazującym na metodzie agregacji monotonicznej jest metoda drzew entropijnych, którą można wykorzystać w systemach wspomagania decyzji opartych na sieciach neuronowych oraz ewolucyjnych algorytmach optymalizacyjnych (Skulimowski, 2014).

8 Algorytm *k*-najbliższych sąsiadów

(ang. *k-nearest neighbours, k*-NN)

Zastosowanie: klasyfikacja lub regresja.
Metoda: uczenie nadzorowane.

Algorytm *k*-najbliższych sąsiadów jest uznawany za najprostszy i najbardziej intuicyjny nieparametryczny klasyfikator. Koncepcję tego algorytmu można przedstawić w dwóch punktach:

1. znalezienie sposobu na opisanie podobieństw dwóch różnych próbek,
2. predykcja dla nowej próbki przez przypisanie jej wartości atrybutu decyzyjnego tej próbki, która jest do niej najbardziej podobna w zakresie wartości atrybutów warunkowych.

Przykładem dobrze ilustrującym tę koncepcję jest sytuacja, w której mamy trzech przyjaciół $x^{(1)}$, $x^{(2)}$, $x^{(3)}$ i znamy ich ulubione utwory muzyczne. Nowy znajomy $x^{(4)}$ jest bardzo podobny do $x^{(2)}$, więc wydaje się słuszne, żeby przyjąć, że ulubiony utwór $x^{(4)}$ jest taki sam jak ulubiony utwór $x^{(2)}$.

Koncepcję można rozszerzyć na podobieństwo reprezentowane przez grupę o zadanej liczbie najbardziej podobnych obiektów. Wtedy, w pierwszym kroku należy opisać podobieństwa pomiędzy parami przykładów. Następnie dla nowego przykładu, dla którego wykonywana jest predykcja, należy znaleźć zadaną liczbę obiektów najbardziej do niego podobnych. Ostatnim krokiem jest połączenie wybranych przykładów tak, aby otrzymać jedną wartość atrybutu decyzyjnego, która zostanie przypisana do nowego obiektu. Podstawą przedstawionych koncepcji jest podobieństwo, które musi wyrazić jakaś ustalona metryka. Definicję i przykłady podstawowych metryk zamieszczono w dodatku. Należy jednak nadmienić, że w pewnych bardziej zaawansowanych implementacjach algorytmu *k*-NN stosuje się również miary oparte na współczynnikach korelacji (Jaskowiak i Campello, 2011).

Poniżej przedstawiono kroki algorytmu *k*-najbliższych sąsiadów w przypadku klasyfikacji. Zatem zmienna wyjściowa jest skategoryzowana i zadaniem algorytmu jest przewidywanie klasy dla nowego obiektu na podstawie danych ze zbioru treningowego (Cover i Hart, 1967).

Kroki algorytmu w przypadku klasyfikacji (zmienna wyjściowa skategoryzowana, przewidujemy klasę nowego obiektu)

1. Wybór sposobu mierzenia sąsiedztwa.
2. Wybór *k* – liczby sąsiadów.
3. Znalezienie *k* obiektów najbardziej podobnych (leżących najbliższej) do nowego przykładu, dla którego należy określić wartość atrybutu decyzyjnego *y*.
4. Sprawdzenie, jaka klasa występuje najczęściej wśród znalezionych obiektów (*k*-najbliższych sąsiadów).
5. Decyzja klasyfikacyjna – nowemu obiektowi zostaje przypisana klasa, która wystąpiła najczęściej wśród *k*-najbliższych sąsiadów.

UWAGA

Algorytm można również zastosować w przypadku regresji, czyli gdy zmienna wyjściowa jest ciągła i musimy przewidzieć wartość numeryczną dla nowego obiektu. Wtedy w kroku nr 4 obliczamy wartość średnią, medianę lub inną adekwatną miarę statystyczną dla *k*-najbliższych sąsiadów. Wynik ten uznajemy za wartość atrybutu decyzyjnego, którą należy przypisać nowemu przykładowi.

Pozostaje odpowiedzieć na pytanie o to, jak dobrać parametr *k*. Parametr *k* jest hiperparametrem, co oznacza, że jego wartość musi być podana z góry, w pierwszych krokach wykonywanego algorytmu. Zatem *k* jest podstawową zmienną konfiguracyjną, która będzie miała wpływ na jakość predykcji uzyskanej za pomocą algorytmu *k*-najbliższych sąsiadów.

Zwróćmy uwagę na ogólną zależność pomiędzy wielkością *k* i predykcją:

- mała wartość *k*: duża zmienność predykcji, model podatny na szum w danych,
- duża wartość *k*: duże ryzyko pojawienia się błędu systematycznego objawiającego się systematycznymi przesunięciami przewidywanych wartości,
- optymalna wartość *k*: wartość na tyle duża, że będzie minimalizowane prawdopodobieństwo błędnych klasyfikacji, ale na tyle mała, że *k*-najbliższych sąsiadów będzie dostatecznie bliskimi sąsiadami nowego obiektu.

Tym samym dobór wartości hiperparametru *k* jest bardzo ważny i wpływa na uzyskany model (Hall i in., 2007). Optymalne *k* otrzymuje się przy zastosowaniu metody sprawdzianu krzyżowego.

Sprawdzian krzyżowy dla doboru optymalnej wartości *k*

1. Wstępne ustalenie wartości *k*.
2. Losowy podział danych na *n* rozłącznych zbiorów.
3. Pierwszy zbiór danych zostaje uznany za zbiór testowy, pozostałe (n – 1) zbiorów za zbiór treningowy.
4. Dokonanie predykcji algorytmem *k*-NN dla wszystkich przykładów ze zbioru testowego, porównanie wyników z ich rzeczywistymi wartościami atrybutu decyzyjnego i obliczenie dokładności N_{ov} w przypadku klasyfikacji lub błędu RME w przypadku regresji.
5. Powtórzenie kroku nr 4 jeszcze (n – 1) razy, przyjmując jako zbiory testowe kolejne zbiory nr 2, ..., n.
6. Obliczenie wartości średniej z otrzymanych *n* wskaźników oceny dokonanych predykcji.
7. Powtórzenie kroków nr 2–6 dla różnych *k*.
8. Wybranie tej wartości hiperparametru *k*, dla której uzyskano najlepszą średnią ocenę predykcji.

Zatem sprawdzian krzyżowy oznacza wykonanie prostych kroków, ale wymaga ich wielokrotnego powtórzenia. Tym samym konieczne jest wykonanie wielu obliczeń, co szczególnie przy dużych zbiorach danych skutkuje czasochłonnością tego rozwiązania. Dlatego też dość często rezygnuje się ze znalezienia optymalnego *k* i wartość tego hiperparametru przyjmuje się arbitralnie.

Przykład 8.1. Klasyfikacja poziomu stresu przy użyciu algorytmu *k*-NN

Pod adresem: https://www.kaggle.com/laavanya/stress-level-detection dostępne są dane dotyczące trzech oznak stresu u człowieka oraz ich natężenia w zależności od poziomu stresu. Są nimi:

1) *humidity* – wilgotność skóry związana z intensywnością wydzielania potu podana w miligramach na minutę,
2) *temperature* – temperatura ciała podana w stopniach Fahrenheita,
3) *step count* – liczba kroków wykonanych w ciągu minuty.

Na podstawie wartości tych trzech parametrów nastąpi zaklasyfikowanie stresu do jednego z trzech poziomów: niskiego, umiarkowanego i wysokiego, oznaczonych odpowiednio etykietami 0, 1 i 2.

```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import (accuracy_score, recall_score,
                             precision_score, confusion_matrix,
                             classification_report, f1_score)
#aby zaimportować z jednej biblioteki funkcje znajdujące się w kilku
#linijkach, należy ich nazwy podać po przecinku, umieszczając je w nawiasie

#wczytanie danych
df = pd.read_csv('Stress-Lysis.csv')

#usunięcie wierszy z brakującymi danymi
df.dropna(inplace = True)

#wyświetlenie nazw zmiennych (kolumn)
print(df.columns)

#podział danych na cechy (X) oraz etykiety (y)
columns_to_analyze = ['Humidity', 'Temperature', 'Step count']
X = df[columns_to_analyze].to_numpy()
y = df['Stress Level'].to_numpy()
```

Przed przystąpieniem do uczenia modelu klasyfikacyjnego dane należy podzielić na zbiór uczący, który posłuży do przeprowadzenia procesu uczenia, oraz zbiór testowy, który posłuży do ewaluacji modelu. Jeżeli podział ma zostać przeprowadzony losowo, można w tym celu użyć funkcji train_test_split z biblioteki scikit-learn, do której należy podać następujące argumenty:

- *X* – macierz cech sygnałów o wymiarach $n \times m$, gdzie n – liczba obiektów, m – długość wektora cech pojedynczego obiektu;
- *y* – wektor etykiet;
- train_size – rozmiar zbioru uczącego, zazwyczaj 0,7–0,9;
- test_size – rozmiar zbioru testowego, zazwyczaj 0,1–0,3;
- *stratify* = *y* – parametr zapewniający stratyfikację danych względem wektora *y* (etykiet), czyli podział danych na zbiory zachowujące rozkład klas opisujący cały zbiór danych; pozwala to uniknąć sytuacji, w której wszystkie obiekty należące do jednej klasy znajdą się tylko w jednym zbiorze;
- random_state – stan generatora liczb pseudolosowych; generator zostanie wykorzystany do podziału danych na zbiory; niepodanie wartości tego parametru sprawi, że przy każdym uruchomieniu funkcji stan może być inny, a uzyskiwany podział danych nie będzie powtarzalny.

UWAGA

W przypadku podania wartości parametru train_size można pominąć parametr test_size – zostanie on wyliczony przez funkcję jako 1-train_size. Analogicznie wystarczy podać tylko wartość parametru test_size, wówczas train_size zostanie obliczony jako 1-test_size.

```
#podział danych na zbiór uczący i testowy
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state = 42,
                                                    stratify = y,
                                                    test_size = 0.2)
```

Po podziale danych można przystąpić do inicjalizacji i uczenia klasyfikatora na danych uczących. Podczas inicjalizacji należy do funkcji podać wartości hiperparametrów – w przypadku klasyfikatora *k*-NN najczęściej modyfikowanymi hiperparametrami są:

- *n_neighbors* – liczba sąsiadów, którzy posłużą do określenia klasy obiektów ze zbioru testowego; polecane jest ustalenie nieparzystej liczby sąsiadów, żeby uniknąć sytuacji, w której analizowani sąsiedzi będą równomiernie rozłożeni w kilku klasach;
- *weights* – hiperparametr określający wagi sąsiadów; może przyjąć dwie wartości: 'uniform' oznacza, że wszyscy sąsiedzi będą mieć taką samą wagę (tak samo duży wpływ na ostateczną decyzję o przynależności obiektu do danej klasy), zaś 'distance' oznacza, że wagi zależne będą od odległości sąsiadów od rozważanego obiektu;
- *metric* – metryka wykorzystywana do obliczenia odległości pomiędzy sąsiadami a obiektem, domyślnie metryka Minkowskiego.

Listę wszystkich hiperparametrów można znaleźć w dokumentacji funkcji pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html?highlight=-kneighborsclassifier#sklearn.neighbors.KNeighborsClassifier.

UWAGA

W kontekście algorytmów uczenia maszynowego parametrami nazywa się tylko te parametry, których wartości dobierane są podczas procesu uczenia. Parametry, które mogą zostać zmienione przez użytkownika jeszcze przed rozpoczęciem treningu (czyli zmienne przekazywane do funkcji inicjalizującej model), to tzw. hiperparametry.

ZASADA

Wszystkie algorytmy uczenia maszynowego dostępne w bibliotece scikit-learn uczy się w taki sam sposób: używając metody clf.fit(X_train, y_train). Predykcję klas (w przypadku modeli klasyfikacyjnych) lub wartości ciągłych (w przypadku modeli regresyjnych) wykonuje się za pomocą metody clf.predict(X_test).

ZASADA

Do treningu modeli klasyfikacyjnych i regresyjnych zawsze używa się zarówno macierzy cech, jak i wektora etykiet, natomiast do predykcji wyłącznie macierzy cech. Etykiety zbioru testowego służą wyłącznie do ewaluacji modelu.

```
#inicjalizacja i trening klasyfikatora
knn = KNeighborsClassifier(n_neighbors = 2, weights = 'uniform')
knn.fit(X_train, y_train)
```

Ewaluację nauczonego klasyfikatora prowadzi się z wykorzystaniem zbioru testowego. W tym celu na podstawie cech obiektów testowych (macierzy *X_test*) należy przeprowadzić predykcję, czyli użyć klasyfikatora do przewidzenia klas, do których należą te obiekty. Ocena klasyfikatora powinna być obiektywna i przeprowadzona w sposób powtarzalny – używa się do niej tzw. metryk sukcesu, które za pomocą wartości liczbowych przedstawiają zdolności klasyfikatora do poprawnego przewidywania klasy. Najczęściej używanymi metrykami są dokładność klasyfikacji (ang. *accuracy*), czułość (ang. *recall*), precyzja (ang. *precision*) oraz F1.

Do oceny klasyfikatora służy również macierz pomyłek, w której zestawione są poprawne i niepoprawne predykcje obiektów należących do wszystkich klas. Poprawne predykcje znajdują się na przekątnej macierzy, zaś niepoprawne poza nią.

```
#predykcja etykiet obiektów ze zbioru testowego
preds = knn.predict(X_test)

#wyświetlenie macierzy pomyłek
print('macierz pomyłek:\n', confusion_matrix(y_test, preds))

#obliczenie metryk sukcesu
accuracy = accuracy_score(y_test, preds)
recall = recall_score(y_test, preds, average = 'weighted')
precision = precision_score(y_test, preds, average = 'weighted')
f1 = f1_score(y_test, preds, average = 'weighted')
print('dokładność klasyfikacji = ', accuracy,
      '\n średnia czułość = ', recall,
      '\n średnia precyzja = ', precision,
      '\n średnie F1 = ', f1)

#raport dotyczący wyników klasyfikacji: precyzja, czułość, f1 uzyskane dla
#poszczególnych klas oraz ich liczebność wartości podane z dokładnością
#do dwóch miejsc po przecinku
print(classification_report(y_test, preds))
```

UWAGA

Metryki takie jak czułość, precyzja i F1 są zdefiniowane dla przypadków binarnych. Jeżeli w analizowanym zbiorze danych występują więcej niż dwie klasy, konieczne jest określenie, w jaki sposób wartości metryk zostaną uśrednione dla wszystkich klas. W funkcjach zaimplementowanych w bibliotece scikit-learn możliwy jest wybór pięciu metod uśredniania wyników:

1) 'binary' – wartość domyślna, używana tylko w przypadku danych binarnych,
2) 'macro' – średnia arytmetyczna,
3) 'weighted' – średnia ważona, zalecana w przypadku dużego niezrównoważenia zbioru danych (dużej różnicy w liczebności klas),
4) 'micro' – wartość metryki wyznaczana przez zliczenie wszystkich poprawnych oraz niepoprawnych predykcji na zbiorze testowym, bez rozróżniania na klasy obiektów,
5) 'samples' – wartość metryki wyznaczana dla każdego obiektu, a następnie uśrednienie wszystkich wyników, często jest to równoważne z wyznaczeniem dokładności klasyfikacji.

UWAGA

W przedstawionym przykładzie uzyskano dokładność klasyfikacji równą 100%. Należy jednak pamiętać, że w praktyce tak dobre wyniki są rzadkością.

Przykład 8.2. Optymalizacja hiperparametrów modelu na przykładzie klasyfikatora najbliższych sąsiadów

Dobór wartości hiperparametrów modelu ma kluczowe znaczenie dla jego zdolności predykcyjnych – nieprawidłowo dobrane wartości hiperparametrów mogą spowodować uzyskanie modelu o słabych zdolnościach generalizacyjnych, który nie będzie dawał satysfakcjonujących wyników. Wartości te dobiera się zazwyczaj za pomocą optymalizacji przy użyciu specjalnie zaprojektowanych w tym celu narzędzi. Jedną z bibliotek gwarantujących efektywne pod względem obliczeniowym poszukiwanie optymalnych

wartości hiperparametrów jest Optuna (Akiba i in., 2019). Będzie ona wykorzystywana w przykładach prezentowanych w niniejszej monografii.

Prowadząc optymalizację hiperparametrów przy użyciu Optuny, należy zdefiniować kilka niezbędnych funkcji i zmiennych:

- funkcję *objective*,
- funkcję *get_space*,
- zmienną *model*,
- zmienną *scoring*,
- zmienną *trials*.

Objective, czyli funkcja celu, to kryterium oceny jakości rozwiązania (modelu). Definicja funkcji oraz niezbędne biblioteki przedstawiono w kodzie poniżej.

```
from sklearn.model_selection import (cross_validate, StratifiedKFold)
import numpy as np

def objective(trial, model, get_space, X, y):
    model_space = get_space(trial)
    mdl = model(**model_space)
    #do zmiennej cv należy przypisać sposób, w jaki będą tworzone
    #podzbiory wykorzystywane podczas sprawdzianu krzyżowego,
    #w sytuacji, gdy podział na podzbiory nie jest narzucony
    #i optymalizowany jest model klasyfikacyjny, należy użyć podzbiorów,
    #na które mają zostać podzielone dane

      scores = cross_validate(mdl, X, y, scoring = scoring,
                                         cv = StratifiedKFold(n_splits = 5),
                                         return_train_score = True)

    #funkcja powinna zwracać średnią wartość metryki zdefiniowanej
    #w zmiennej scoring obliczoną dla podzbiorów testowych

    return np.mean(scores['test_accuracy'])
```

Funkcja *get_space* to funkcja określająca przestrzeń hiperparametrów oraz wartości, jakie będą mogły one przyjąć, czyli dopuszczalny obszar rozwiązań. Poniżej przedstawiono kod definiujący tę funkcję dla klasyfikatora *k*-najbliższych sąsiadów.

```
def get_space(trial):
  space = {
           "n_neighbors": trial.suggest_int("n_neighbors",2, 5),
           "metric": trial.suggest_categorical("metric",["mahalanobis"]),
           "weights": trial.suggest_categorical("weights",['uniform',
                                                           'distance'])
         }
  return space
```

Model jest to zmienna określająca wykorzystywany model klasyfikacyjny lub regresyjny, definiowany przez nazwę klasy, w której zaimplementowany jest klasyfikator lub regresor mający zostać poddany optymalizacji.

Scoring jest to metryka służąca do ewaluacji modelu podczas jego optymalizacji. Jej wartość jest wyznaczana i uśredniana dla podzbiorów walidacyjnych w sprawdzianie krzyżowym wykonywanym podczas optymalizacji.

Trials jest to zmienna określająca liczbę prób (iteracji), które zostaną wykonane w celu znalezienia zestawu najlepszych hiperparametrów. Im większa liczba hiperparametrów jest optymalizowana i im więcej ich wartości musi zostać wypróbowanych, tym większa powinna być liczba prób.

Przykładowy kod zawierający definicję zmiennych *model*, *scoring* i *trials* przedstawiono poniżej.

```
from sklearn.metrics import make_scorer, accuracy_score
from sklearn.neighbors import KNeighborsClassifier
scoring = {'accuracy': make_scorer(accuracy_score)}
trials = 20 #liczba prób
model = KNeighborsClassifier
```

Obszar poszukiwań wartości poszczególnych zmiennych definiuje się przez użycie odpowiednich metod. Najczęściej stosowane z nich wymieniono poniżej.

- suggest_int – używana, gdy hiperparametr może przyjmować jedynie wartości całkowite. Należy podać dolną i górną granicę obszaru poszukiwań – jeżeli hiperparametr ma przyjąć konkretną wartość inną niż domyślna, należy podać ją dwukrotnie (jako granicę dolną i górną), np. „n_neighbors": trial.suggest_int(„n_neighbors", 7, 7), w przeciwnym razie funkcja zwróci błąd lub użyta zostanie wartość domyślna.
- suggest_uniform – używana, gdy hiperparametr jest zmienną ciągłą. Należy podać dolną i górną granicę obszaru poszukiwań. Prawdopodobieństwo wylosowania wartości ze zdefiniowanego przedziału jest określone rozkładem jednostajnym (każda wartość ma takie samo prawdopodobieństwo wylosowania).
- suggest_categorical – używana, gdy hiperparametr jest zmienną kategorialną (jakościową). Jeżeli hiperparametr ma przyjąć wartość inną niż domyślna, wystarczy podać ją jednokrotnie, tak jak zrobiono powyżej w przypadku hiperparametru metric.

Pozostałe metody można znaleźć w dokumentacji pakietu Optuna pod adresem: https://optuna.readthedocs.io/en/stable/reference/generated/optuna.trial.Trial.html.

Ostatnią zmienną, którą należy zdefiniować, jest *direction* – zmienna wskazująca pożądany kierunek zmian funkcji celu, czyli określająca, czy funkcja celu ma być maksymalizowana, czy minimalizowana. Kierunek zależy od tego, jaka funkcja celu jest użyta – jeżeli jest to funkcja błędu popełnianego przez model, należy ją minimalizować, natomiast jeżeli jest to funkcja wskazująca na liczbę poprawnych predykcji, należy ją maksymalizować.

Po zdefiniowaniu powyższych funkcji i zmiennych można przejść do optymalizacji modelu. W tym celu należy wykorzystać poniższy kod.

```
import optuna
study = optuna.create_study(direction = 'maximize')
study.optimize(lambda x: objective(x, model, get_space,X_train,y_train),
               n_trials = trials)
```

Po zakończeniu optymalizacji hiperparametrów ich dobrane wartości zapisywane są do zmiennej study.best_params – można je wyświetlić oraz wykorzystać do inicjalizacji i treningu modelu. Po zakoń-

czeniu treningu model należy poddać ewaluacji, aby zobaczyć, czy rezultaty uzyskiwane na zbiorze testowym są zadowalające.

```
print('params: ', study.best_params)
knn = model(**study.best_params)
knn.fit(X_train, y_train)
preds = knn.predict(X_test)
print(confusion_matrix(y_test, preds))
```

Innymi wartymi uwagi metodami i bibliotekami służącymi do optymalizacji hiperparametrów są przykładowo:

- biblioteka TPOT (Olson i in., 2016) wykorzystująca algorytmy genetyczne do poszukiwania rozwiązań Pareto-optymalnych, czyli takich, w których wykorzystywana jest wielokryterialna funkcja celu: http://epistasislab.github.io/tpot/,
- metoda GridSearch zaimplementowana w bibliotece scikit-learn: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html,
- metoda RandomizedSearch zaimplementowana w bibliotece scikit-learn: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html.

Zadanie 8.1. Zastosowanie algorytmu *k*-NN w diagnostyce medycznej: klasyfikacja nowotworów piersi

Ze strony https://www.kaggle.com/uciml/breast-cancer-wisconsin-data pobierz dane dotyczące nowotworów piersi, a następnie wczytaj je do struktury DataFrame. W tabeli znajdzie się numer identyfikacyjny pacjenta (kolumna id), diagnoza (kolumna diagnosis: *M* oznacza zmianę złośliwą, *B* zmianę łagodną) oraz trzydzieści parametrów opisujących nowotwór.

Wyświetl nazwy kolumn tabeli. Ostatnia kolumna powinna mieć nazwę 'Unnamed: 32' – usuń ją z tabeli, np. za pomocą metody drop: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.drop.html.

1. Zamień typ danych zawartych w kolumnie diagnosis na dane kategorialne, używając funkcji astype('category'). Następnie wykorzystaj je do utworzenia wektora etykiet numerycznych – możesz w tym celu skorzystać z funkcji LabelEncoder: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html.
2. Utwórz macierz cech na podstawie kolumn zawierających parametry nowotworów.
3. Przeprowadź trening klasyfikatora *k*-NN oraz jego ewaluację. Dobierz hiperparametry modelu w taki sposób, by zmaksymalizować dokładność klasyfikacji. Co sądzisz o uzyskanych wynikach? Czy są one satysfakcjonujące? Która klasa częściej była źle klasyfikowana?
4. Przeprowadź redukcję wymiarowości przy użyciu analizy składowych głównych opisanej w podrozdziale 9.1. Sprawdź, ile składowych jest potrzebnych, żeby osiągnąć dokładność klasyfikacji nie mniejszą niż 0,9 – w tym celu powtórz proces uczenia oraz ewaluacji modelu. Jaka część zmienności jest wyjaśniana przez te składowe?

9 Przekleństwo wymiarowości

Zgodnie z definicją przedstawioną w rozdziale 3 obiekt w zbiorze treningowym jest wielowymiarową zmienną losową. Bardzo często w problemach rozwiązywanych metodami uczenia maszynowego uwzględnia się dziesiątki, a nawet setki atrybutów warunkowych. W modelach stosowanych w uczeniu maszynowym zwykle każda zmienna jest reprezentowana jako osobny wymiar, co oznacza, że poszczególne obiekty bywają rozpatrywane w przestrzeniach nawet kilkuset wymiarowych. Należy wiedzieć, że wraz ze wzrostem liczby wymiarów w modelu liczba obiektów koniecznych do wiarygodnego oszacowania parametrów rośnie wykładniczo, co jest określane jako tzw. przekleństwo wymiarowości. Brak zapewnienia ilości danych treningowych odpowiedniej do wymiaru przestrzeni, w której rozpatrywane są te obiekty, prowadzi zwykle do przetrenowania modelu. Wynika z tego następująca ważna zasada.

ZASADA

Aby zapewnić taką samą dokładność przewidywania wartości atrybutu decyzyjnego jak w przypadku przestrzeni o mniejszym wymiarze, wymagana liczba przykładów treningowych rośnie wykładniczo wraz ze wzrostem wymiaru przestrzeni cech (Duda i in., 2000).

Zasada wydaje się bardzo prosta i można dojść do wniosku, że wystarczy odpowiednio duży zbiór treningowy, aby poradzić sobie z dowolnie dużą liczbą zmiennych uwzględnianych w modelu. Zauważmy jednak, że wraz ze wzrostem liczby zmiennych wymagana liczba danych rośnie nieliniowo, prowadząc do przekleństwa wymiarowości, ponieważ ilość danych możliwych do pozyskania jest zwykle ograniczona, a obliczenia na dużym zbiorze danych są bardzo czasochłonne.

Odpowiedzią na te problemy jest redukcja wymiaru przestrzeni, z której pochodzą rozpatrywane dane. Można to osiągnąć przez ograniczenie liczby zmiennych opisujących każdy obiekt. Stosowane w tym celu metody dzielą się na dwie grupy:

1) selekcja – wybór istotnych atrybutów warunkowych i odrzucenie pozostałych traktowanych jako nadmiarowe,
2) ekstrakcja cech – stworzenie zadanej liczby nowych zmiennych na podstawie wejściowego zbioru zmiennych.

W przypadku metod opierających się na selekcji eliminowane są te zmienne, których usunięcie spowoduje najmniejszą utratę niesionych informacji. Atrybuty, których odrzucenie należy rozważyć, to atrybuty warunkowe, najsłabiej skorelowane ze zmienną odpowiadającą atrybutowi decyzyjnemu lub atrybuty warunkowe silnie skorelowane z innymi atrybutami warunkowymi. W celu wytypowania zmiennych, których usunięcie zostanie rozważone, najczęściej stosuje się następujące narzędzia:

- współczynnik korelacji liniowej Pearsona,
- współczynnik korelacji rang Kendalla,
- współczynnik korelacji rang Spearmana,
- test zgodności,
- test wariancji Fishera,
- miara redukcji entropii zmiennej wyjściowej przez zmienne wejściowe.

Należy jednak pamiętać, że pomimo braku korelacji pewnej zmiennej wejściowej ze zmienną wyjściową zmienne te mogą być skorelowane nieliniowo lub pośrednio, czyli przez inne zmienne (Szeliga, 2017). Dlatego powyższe narzędzia dostarczają jedynie wstępne informacje o tym, które zmienne należy rozważyć jako najmniej znaczące. Podczas ostatecznej decyzji o usunięciu wybranych atrybutów warunkowych ważną rolę odgrywać będzie wiedza analityka na temat badanego zagadnienia.

Aby zminimalizować ryzyko dużej utraty zmienności przy usuwaniu zmiennych, próbę redukcji wymiaru rozpoczyna się często od metod ekstrakcji cech. Nowy zbiór zmiennych stworzony jest wtedy jako kombinacja początkowych zmiennych wejściowych, a następnie wybrane są tylko te nowe zmienne, które niosą najwięcej informacji, i w ten sposób dane zostają zrzutowane z hiperprzestrzeni do przestrzeni o mniejszej liczbie wymiarów. Bardzo często używanym narzędziem do ekstrakcji cech jest analiza składowych głównych. Metoda ta jest powszechnie stosowana w analizie obrazów czy wyrażeń języka naturalnego (Szeliga, 2017).W algorytmie głównych składowych dąży się do przedstawienia danych w przestrzeni o możliwie małym wymiarze, przy zachowaniu jak największego zróżnicowania w danych. Choć przejście zmiennych wejściowych do niższego wymiaru wiąże się z utratą niewielkiej części informacji, to jednak rekompensatą są zalety tego rozwiązania takie jak łatwiejsze zidentyfikowanie podstawowej struktury danych i skuteczniejsze wykorzystanie docelowych algorytmów uczenia maszynowego.

Podsumowując, należy wymienić następujące zalety przeprowadzenia redukcji wymiaru:

- zmniejszenie rozmiaru danych, co przekłada się na wydajność algorytmów uczenia maszynowego,
- łatwiejsze zidentyfikowanie podstawowej struktury danych,
- zapobieganie przetrenowaniu modelu, czyli nadmiernemu dopasowaniu do danych, co zwykle ma miejsce, gdy dysponuje się zbyt małą liczbą obiektów w zbiorze treningowym w porównaniu z liczbą wymiarów przestrzeni, w której są rozpatrywane te obiekty.

9.1. Analiza składowych głównych

(ang. *Principal Component Analysis*, PCA)

Zastosowanie: redukcja wymiaru.
Metoda: uczenie nienadzorowane.

W analizie głównych składowych zmniejszenie liczby zmiennych wejściowych odbywa się przez rzutowanie obiektów do przestrzeni rozpiętej na ustalonej liczbie wektorów prostopadłych do siebie, tzw. składowych głównych. Wektory te określa się jako odpowiednią kombinację liniową oryginalnych zmiennych wejściowych. Składowe główne konstruuje się w taki sposób, aby były wektorami prostopadłymi względem siebie, czyli aby nie były ze sobą liniowo skorelowane i aby każda składowa reprezentowała inną część informacji ukrytej z oryginalnych zmiennych wejściowych. Drugim warunkiem branym pod uwagę podczas wyznaczania współczynników w kombinacjach liniowych jest maksymalizacja zmienności, ponieważ im większa jest zmienność, tym więcej informacji niesie dana kombinacja zmiennych wejściowych. Do oceny zmienności używa się macierzy kowariancji, która wskazuje, jak różnią się od siebie poszczególne pary zmiennych.

Zatem podczas szukania kolejnych składowych dąży się do:

- jak najlepszego zachowania struktury danych,
- braku korelacji pomiędzy kolejnymi składowymi,
- maksymalizacji zmienności, która nie została wyjaśniona przez poprzednią składową.

Otrzymane składowe główne są uporządkowane począwszy od tych, które niosą najwięcej informacji. Zatem pierwsza główna składowa reprezentuje największą część informacji, druga główna składowa reprezentuje największą część pozostałej informacji, czyli tej, która nie została wyjaśniona przez pierwszą składową i tak dalej (Szeliga, 2017).

Kroki algorytmu (PCA)

1. Weryfikacja założeń:
   – dane pochodzą z rozkładu normalnego,
   – zbiór danych jest jednorodny,
   – zmienne są skorelowane (im wyższa jest korelacja pomiędzy zmiennymi wejściowymi, tym zastosowanie analizy składowych głównych jest bardziej uzasadnione) (Stanisz, 2007).
2. Standaryzacja zmiennych wejściowych.
3. Ustalenie liczby k, gdzie k to wymiar przestrzeni, do której zostaną zrzutowane dane(*).
4. Znalezienie wektora $a_1 = (a_{11}, a_{12}, ..., a_{1d})$, takiego że nowa zmienna $Z_1 = a_{11}X_1 + a_{12}X_2 + ... + a_{1d}X_d = \sum_{i=1}^{d} a_{1i}X_i$ spełnia następujące warunki:
   – osiąga maksimum wariancji,
   – wektor a_1 jest unormowany, czyli $||a_1|| = a_1'a_1 = 1$(**).
5. Niech j:= 2.
6. Znalezienie wektora $a_j = (a_{j1}, a_{j2}, \ldots, a_{jd})$, takiego że zmienna $Z_j = a_{j1}X_1 + a_{j2}X_2 + \ldots + a_{jd}X_d = \sum_{i=1}^{d} a_{ji}X_i$ spełnia następujące warunki:
   – osiąga maksimum wariancji,
   – $a_j'a_{j-1} = 0$ (zmienne Z_j oraz Z_{j-1} nie są skorelowane liniowo),
   – $||a_j|| = a_j'a_j = 1$(***).
7. Sprawdzenie, czy wykonano maksymalną liczbę kroków:
   ◦ jeżeli $j \neq d$, to należy przyjąć j:= j + 1 i wrócić do kroku nr 6,
   • jeżeli $j = d$, to należy przejść do kolejnego kroku.
8. Otrzymane zmienne Z_1, Z_2, ..., Z_d są głównymi składowymi reprezentującymi 100% zmienności. Należy wybrać możliwie najmniejszą liczbę głównych składowych, które reprezentują znaczny odsetek wariancji. W tym celu można się posłużyć kryterium wystarczającej proporcji, kryterium Kaisera lub wykresem osypiska(****).

(*) Zwykle przyjmuje się $k = d$, czyli na początku rozważa się dalej ten sam wymiar, dokonywane jest jedynie przekształcenie danych z zachowaniem 100% wariancji niesionej przez wszystkie dane wejściowe. Dopiero na końcu podejmuje się decyzję, ile z otrzymanych składowych głównych jest koniecznych do skutecznego wychwycenia większości zróżnicowania niesionego przez oryginalne zmienne wejściowe.

(**) Pierwsza składowa główna jest unormowaną kombinacją liniową oryginalnych zmiennych wejściowych, której wariancja z próby jest największa. Wariancja pierwszej składowej głównej wynosi:

$$s^2(Z_1) = \sum_{j=1}^{N}\sum_{i=1}^{d} a_{i1}a_{j1}s_{ij} = a_1'\mathbf{S}a_1. \tag{9.1}$$

Jest to funkcja o d-zmiennych $a_1, a_2, ..., a_d$. Przyjęte założenie warunku normalizującego $||a_1|| = a_1'a_1 = 1$ pozwala na zastosowanie np. metody mnożników Lagrange'a w celu wyznaczenia ekstremum warunkowego funkcji $s^2(Z_1)$. Można wykazać, że unormowanym wektorem maksymalizującym wariancję jest wektor charakterystyczny odpowiadający największej wartości własnej λ_1 macierzy **S** (Morrison, 1990).

(***) Podobnie jak w przypadku pierwszej składowej głównej, można wykazać, że przy warunkach zadanych dla j-tej składowej głównej wektor jest j-tym wektorem własnym macierzy **S**, który odpowiada j-tej wartości własnej, przy czym $\lambda_j < \lambda_{j-1} < ... < \lambda_1$ (Morrison, 1990).

(****) Kryteriami stosowanymi w celu wyboru liczby głównych składowych są:

- kryterium wystarczającej proporcji – należy wybrać tyle składowych głównych, aby sumaryczny poziom wyjaśnionej wariancji był równy co najmniej 75%;
- kryterium Kaisera – odrzucenie składowych głównych, których wartości własne są mniejsze niż 1 (Kaiser, 1960);
- wykres osypiska – wykres przestawiający funkcję przedziałami liniową, której szczególnymi punktami są wartości własne kolejnych składowych głównych; na podstawie wykresu osypiska odrzuca się te składowe główne, dla których wykres się wypłaszcza; dużym zarzutem wobec tej metody jest subiektywizm analityka danych, jednak metoda ta jest często stosowana i uznana za skuteczną (Krzyśko, 2000).

UWAGA

Na przekątnej macierzy kowariancji znajdują się wariancje zmiennych wejściowych, a ich suma odpowiada łącznej zmienności układu zmiennych, którą należy objaśnić jak najmniejszą liczbą głównych składowych.

UWAGA

W przypadku gdy zmienne są wyrażone w różnych jednostkach lub różnią się rzędem wielkości, zamiast macierzy kowariancji należy wykorzystać macierz korelacji. Wyniki mogą się różnić w zależności od tego, która macierz zostanie użyta do oceny zmienności.

Należy jeszcze wspomnieć, że analiza składowych głównych stosowana jest nie tylko w celu redukcji wymiaru, ale również w celu kompresji sygnałów czy eksploracji danych i ich klasteryzacji.

Przykład 9.1. Zastosowanie PCA do wizualizacji danych

Analiza danych często rozpoczynana jest od ich wizualizacji. Dzięki temu możliwe jest zapoznanie się z rozkładem wartości parametrów, przedziałem przyjmowanych przez nie wartości czy też sprawdzenie, czy grupy obiektów wchodzących w skład bazy różnią się między sobą. W przypadku danych opisanych więcej niż trzema parametrami wizualizacja jest jednak utrudniona – konieczne jest zrezygnowanie z prezentacji niektórych parametrów lub przetworzenie danych w sposób umożliwiający zawarcie całej informacji w dwóch lub trzech składowych. Do tego może posłużyć m.in. analiza składowych głównych.

Sposób przeprowadzenia analizy składowych głównych przy użyciu biblioteki scikit-learn przedstawiony zostanie na przykładzie bazy danych *iris* wbudowanej w bibliotekę scikit-learn. W bazie tej zawarte są informacje dotyczące cech budowy kwiatów trzech gatunków irysów. Każdy kwiat opisany jest za pomocą czterech parametrów:

1) długości działki kielicha (ang. *sepal length*),
2) szerokości działki kielicha (ang. *sepal width*),
3) długości płatka (ang. *petal length*),
4) szerokości płatka (ang. *petal width*).

Jest to więc przykład danych, których wizualizacja wymaga użycia algorytmu PCA.

Jak zostało wspomniane, analiza składowych głównych jest wrażliwa na rząd wielkości danych oraz ich jednostki – nieuwzględnienie tego faktu może doprowadzić do uzyskania niezadowalających rezultatów. Stosowanym w praktyce rozwiązaniem tego problemu jest standaryzacja danych, która polega na przeskalowaniu parametrów tak, by były one opisane rozkładem normalnym o wartości średniej równej 0 oraz odchyleniu standardowym równym 1.

```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
iris = datasets.load_iris() #wczytywanie danych
X = iris.data #parametry opisujące obiekty
y = iris.target #przynależność obiektów do klas (gatunków irysów)
X = StandardScaler().fit_transform(X) #standaryzacja danych
```

W celu wyświetlenia i przeanalizowania danych dobrym rozwiązaniem jest dodanie ich do struktury DataFrame, czyli rodzaju tabeli tworzonej przy użyciu biblioteki Pandas. Dzięki wyświetleniu utworzonej tabeli, można zaznajomić się ze strukturą danych, sprawdzić liczbę klas obecnych w danych oraz zakresy wartości, jakie przyjmują poszczególne parametry.

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.hstack((X,y[:, np.newaxis])),
                  columns=['sepal length', 'sepal width',
                           'petal length', 'petal width',
                           'target numerical'])
print(df)
print(np.unique(y))
#unikalne wartości w wektorze y - w ten sposób można sprawdzić, jakie
#klasy znajdują się w wektorze y i jakie są ich etykiety
```

UWAGA

Sprawdzenie liczby klas wyodrębnionych w danych jest zalecane nawet w sytuacji, gdy w dokumentacji bazy danych jest zawarta informacja o klasach. Nie można wykluczyć, że osoba przygotowująca bazę danych mogła popełnić błąd podczas przygotowania etykiet obiektów lub w bazie zawarte są obiekty, które nie powinny się w niej znaleźć. Sprawdzenie danych przed przystąpieniem do ich analizy pozwala zmniejszyć ryzyko powielenia błędów.

Po przeanalizowaniu zawartości tabeli i upewnieniu się, że nie występują w niej więcej niż trzy gatunki irysów, można przystąpić do wyznaczenia składowych głównych.

Inicjalizując obiekt, który posłuży do przeprowadzenia analizy składowych głównych, należy zdefiniować liczbę składowych, która zostanie wyznaczona. Ponieważ w tym przypadku PCA ma posłużyć do wizualizacji danych, wyznaczone zostaną tylko dwie składowe. Następnie zostaną one dodane do nowej tabeli wraz z etykietami obiektów.

```
from sklearn.decomposition import PCA
import pandas as pd
pca = PCA(n_components = 2)
#wyznaczone zostaną tylko dwie pierwsze składowe główne
principal_components = pca.fit_transform(X)

pca_df = pd.DataFrame(data = (np.hstack((principal_components,
                                         y[:, np.newaxis])))),
                      columns = ['PC1', 'PC2', 'target numerical'])
```

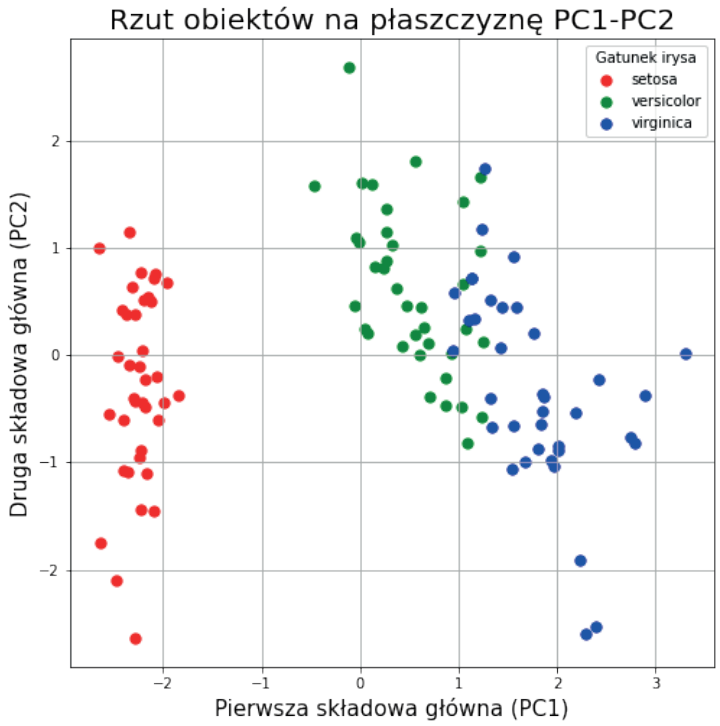
ZASADA

Liczba składowych n_components nie może być większa niż liczba cech opisujących obiekt, czyli w tym przypadku 4. Wpisanie większej wartości spowoduje, że funkcja zwróci błąd.

W tym momencie każdy obiekt w zbiorze danych opisany jest dwoma parametrami (dwiema składowymi głównymi). Umożliwia to wizualizację danych na płaszczyźnie PC1-PC2. W celu zwiększenia czytelności wykresu obiekty należące do różnych klas oznaczone zostaną różnymi kolorami (rys. 9.1).

```
from matplotlib import pyplot as plt
fig = plt.figure(figsize = (8,8)) #określenie wymiarów obrazu
ax = fig.add_subplot(1,1,1) #obraz ma się składać z jednego wykresu
```

```
ax.set_xlabel('Pierwsza składowa główna (PC1)', fontsize = 15)
#podpis osi X
ax.set_ylabel('Druga składowa główna (PC2)', fontsize = 15)
#podpis osi Y
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2', fontsize = 20)
#tytuł wykresu
targets = [0,1,2] #lista etykiet, które znajdują się w zbiorze danych
colors = ['r', 'g', 'b']
#lista kolorów, które posłużą do oznaczenia obiektów należących do
#różnych klas

for target, color in zip(targets,colors):
 indicesToKeep = pca_df['target numerical'] == target
 ax.scatter(pca_df.loc[indicesToKeep, 'PC1'],
            pca_df.loc[indicesToKeep, 'PC2'],
            c = color, s = 50)
ax.legend(targets, title = 'Gatunek irysa') #legenda
ax.grid() #włączenie siatki na tle wykresu
```



Rysunek 9.1. Wizualizacja danych przez rzutowanie obiektów na płaszczyznę utworzoną przez dwie pierwsze składowe główne

UWAGA

Wyznaczenie tylko dwóch składowych głównych jest dobrym wyborem, jeżeli mają one posłużyć do wizualizacji danych. Jeśli jednak składowe główne mają zostać wykorzystane do przeprowadzenia klasyfikacji obiektów, liczbę składowych głównych należy dobrać tak, by łącznie wyjaśniały one dużą część zmienności danych. Wybranie zbyt małej liczby składowych głównych może spowodować nadmierną redukcję wymiarowości i doprowadzić do sytuacji, w której skuteczna klasyfikacja nie będzie możliwa.

Do sprawdzenia, jak dużą część zmienności wyjaśniają poszczególne składowe główne, służy metoda pca.explained_variance_ratio_. W rozważanym przypadku metoda zwróci dwie wartości: 0,73799433 i 0,22216601. Oznaczają one, że pierwsza składowa wyjaśnia około 74% zmienności, a druga składowa około 22% zmienności. Pozostałe około 4% zmienności są wyjaśniane przez dalsze składowe.

Biblioteka scikit-learn umożliwia również sprawdzenie, jakie wagi przypisane zostały poszczególnym cechom podczas wyznaczania składowych głównych – służy do tego metoda pca.components_. Znając wagi, można zapisać wzór, z którego wyliczone zostały wartości poszczególnych składowych głównych. Przykładowo, pierwsza składowa główna PC1 dla zbioru danych *iris* wyznaczana jest w następujący sposób:

$$\begin{aligned} PC1 = {} & 0.53256289 \cdot sepal\_length + (-0.27760892) \cdot sepal\_width + \\ & + 0.57352248 \cdot petal\_length + 0.55711939 \cdot sepal\_length \end{aligned}$$

ZASADA

Jeżeli analiza składowych głównych ma zostać wykorzystana do redukcji wymiarowości danych, które następnie posłużą do przeprowadzenia klasyfikacji, podział na zbiór uczący i testowy powinien zostać przeprowadzony jeszcze przed zastosowaniem algorytmu PCA.

Kod wyznaczający składowe główne wyglądałby wtedy tak.

```
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
#funkcja dzieląca dane na zbiory uczący oraz testowy

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state = 42,
                                                    stratify = y)
#stratify = y oznacza, że dane mają zostać podzielone na podzbiory
#z zachowaniem struktury klas

#pobranie drugiego wymiaru macierzy _train, czyli liczby parametrów
#opisujących obiekt
n_components = X_train.shape[1]

pca = PCA(n_components = n_components)
#wyznaczone zostanie tyle składowych głównych, ile jest parametrów obiektu

principal_components_train = pca.fit_transform(X_train)
principal_components_test = pca.transform(X_test)
```

Zadanie 9.1 Wizualizacja danych przy użyciu analizy składowych głównych

Pod adresem: https://www.kaggle.com/parulpandey/palmer-archipelago-antarctica-penguin-data znajdują się dane dotyczące trzech gatunków pingwinów żyjących na wyspach archipelagu Palmera.

Proszę wykonać następujące kroki.

1. Pobierz pliki z bazą danych i wczytaj dane znajdujące się w pliku penguins_size.csv do struktury DataFrame. Wczytane dane podzielone będą na siedem kolumn:
   - species – gatunek pingwina,
   - island – wyspa, którą zamieszkuje dany osobnik,
   - culmen_lenght_mm – długość dzioba w milimetrach,
   - culmen_depth_mm – grubość dzioba w milimetrach,
   - flipper_length_mm – długość skrzydła w milimetrach,
   - body_mass_g – masa ciała w gramach,
   - sex – płeć osobnika.
2. Na podstawie danych w kolumnie species wygeneruj wektor zawierający etykiety numeryczne osobników.
3. Zastosuj algorytm PCA do wyznaczenia składowych głównych. W tym celu wykorzystaj parametry culmen_lenght_mm, culmen_depth_mm, flipper_length_mm oraz body_mass_g. Pamiętaj o ustandaryzowaniu danych.
4. Sprawdź, jaka część zmienności wyjaśniana jest przez poszczególne składowe główne, oraz wyznacz wzory, za pomocą których można obliczyć wartości składowych.
5. Ponownie zastosuj algorytm PCA, tym razem na danych nieustandaryzowanych.
6. Zwizualizuj dane ustandaryzowane oraz nieustandaryzowane na płaszczyźnie PC1-PC2 oraz w przestrzeni PC1-PC2-PC3. Czy standaryzacja danych znacząco wpłynęła na uzyskane wykresy? W którym przypadku separacja gatunków na wykresach jest lepsza?

9.2. Analiza składowych niezależnych

(ang. *Independent Component Analysis*, ICA)

Zastosowanie: redukcja wymiaru.
Metoda: uczenie nienadzorowane.

W problemach rozwiązywanych za pomocą analizy składowych niezależnych zakłada się, że obserwowany sygnał reprezentowany przez wektor obserwacji jest liniową kombinacją sygnałów pochodzących z kilku źródeł, przy czym sygnały te są statystycznie niezależne oraz nie pochodzą z rozkładu Gaussa.

Zatem jeżeli $X = (x^{(1)}, x^{(2)}, ..., x^{(n)})^T$ jest macierzą obserwacji, to przyjmuje się, że dla $Z = (z^{(1)}, z^{(2)}, ..., z^{(m)})^T$ ukrytych komponentów poszczególnych sygnałów $x^{(j)}$, $j = 1, 2, ..., n$ istnieje pewna macierz A zwana mieszającą taka, że:

$$x^{(j)} = a_{j1}z_1 + a_{j2}z_2 + ... + a_{jm}z_m. \tag{9.2}$$

Wtedy macierz obserwacji można zapisać jako iloczyn macierzy mieszającej i ukrytych komponentów:

$$X = AZ. \tag{9.3}$$

Z równania tego można wyznaczyć macierz komponentów ukrytych:

$$Z = A^{-1}X, \tag{9.4}$$

gdzie macierz A^{-1} zwana jest macierzą separującą i jest to macierz odwrotna do macierzy A.

W klasycznych problemach rozwiązywanych metodą analizy składowych niezależnych znany jest jedynie sygnał zmieszany reprezentowany przez macierz X. Dlatego też, aby móc wyznaczyć macierz mieszającą A, konieczne jest skorzystanie z przyjętych założeń, zgodnie z którymi sygnały źródłowe $z^{(1)}, z^{(2)}, ..., z^{(m)}$:

- mają rozkłady różne od rozkładu normalnego,
- są niezależne statystycznie (Hyvärinen i in., 2001).

Kroki algorytmu (ICA)

1. Weryfikacja założeń:
   – niezależność danych źródłowych,
   – dane źródłowe nie mają rozkładu normalnego,
   – liczba obserwacji jest co najmniej tak duża jak liczba źródeł ukrytych, czyli $n \geq m$.
2. Centracja macierzy obserwacji X.
   W wycentrowanej macierzy $\overline{X}$ średnie poszczególnych zmiennych są równe 0.
3. Transformacja wycentrowanej macierzy obserwacji $\overline{X}$ w tzw. macierz wybieloną $\widetilde{X}$ (ang. *whitening matrix*), której zmienne są parami nieskorelowane oraz wariancja każdej z nich jest równa 1, czyli $X\widetilde{X}^T = 1$.
4. Ewentualna redukcja wymiaru metodą PCA.
5. Wyznaczenie macierzy separującej $\widetilde{A}^{-1}$ tak, aby sygnały źródłowe $z^{(1)}, z^{(2)}, ..., z^{(m)}$ były maksymalnie niezależne. Jest to problem z zakresu optymalizacji, do rozwiązania którego stosowane są:
   – minimalizacja informacji wzajemnej(*) wynikowych komponentów osiągana przez maksymalizację entropii (Comon, 1994);
   – szacowanie przez wyszukiwanie zmiennych o rozkładzie maksymalnie różniącym się od rozkładu normalnego, tzw. maksymalizacja niegaussowości; w tym celu stosuje się kurtozę, jako miarę spłaszczenia rozkładu, lub współczynnik negentropii, który wskazuje, jak podobny do rozkładu normalnego jest dany rozkład prawdopodobieństwa (Bell i Sejnowski, 1995).
6. Obliczenie sygnałów ukrytych na podstawie równania $Z = \widetilde{A}^{-1}\widetilde{X}$.

(*) Informacja wzajemna to miara zależności między dwiema zmiennymi losowymi określająca, o ile poznanie wybranej zmiennej zmniejsza niepewność o innej zmiennej. Zmienne są niezależne, gdy informacja wzajemna jest równa zero. Do oceny informacji wzajemnej używa się zwykle miary entropii.

Analiza składowych niezależnych jest wykorzystywana do rozwiązywania problemu tzw. ślepej separacji źródeł (ang. *Blind Source Separation*, BSS). Szczególnym przypadkiem stosowanej separacji źródeł jest separacja źródeł dźwięku. Najczęściej stosuje się ją do wyodrębniania mowy poszczególnych mówców z sygnału zawierającego sygnał generowany przez kilka osób mówiących równocześnie, czyli tzw. efekt cocktail party (ang. *cocktail party problem*), lub wyodrębniania dźwięków poszczególnych instrumentów grających równocześnie.

Przykład 9.2. Separacja źródeł dźwięku

Działanie algorytmu ICA zostanie pokazane na przykładzie wykorzystującym samodzielnie utworzony miks sygnałów, co pozwoli na ocenę skuteczności jego działania. Przykład został przygotowany na podstawie dokumentacji biblioteki scikit-learn znajdującej się pod adresem: https://scikit-learn.org/stable/auto_examples/decomposition/plot_ica_blind_source_separation.html.

Na początek wygenerowano trzy sygnały: prostokątny, piłokształtny oraz sinusoidalny. Sygnały mają taką samą długość (8 s) oraz częstotliwość próbkowania (250 Hz). Po wygenerowaniu sygnały zostały dodane do jednej tablicy.

```
import numpy as np
from scipy import signal
n_samples = 2000 #liczba próbek
time = np.linspace(0, 8, n_samples)
#wektor czasu o zadanej długości - sygnały trwają 8 s i mają długość
#2000 próbek,więc częstotliwość próbkowania wynosi 250 Hz.
```

```
s1 = np.sin(2 * time) #sinus
s2 = np.sign(np.sin(3 * time)) #sygnał prostokątny
s3 = signal.sawtooth(2 * np.pi * time) #sygnał piłokształtny
S = np.stack((s1, s2, s3)).T
#dodanie sygnałów do jednej macierzy, S.T oznacza transpozycję macierzy S
```

Następnie do sygnałów dodano szum gaussowski. Sygnały rzeczywiste zarejestrowane przez czujnik pomiarowy zawierają w sobie szum, więc dobrze jest to uwzględnić w symulacji. Po dodaniu szumu sygnały zostały ustandaryzowane, a następnie zmiksowane – dzięki temu zaimitowano sytuację, w której sygnały zostały zarejestrowane przez czujniki znajdujące się w różnej odległości od ich źródeł. Sygnał zmieszany wyznaczony został jako iloczyn skalarny macierzy sygnałów i macierzy mieszającej.

```
from sklearn.preprocessing import StandardScaler

np.random.seed(0)
S += 0.2 * np.random.normal(size = S.shape)
#dodanie szumu gaussowskiego do sygnałów

S = StandardScaler().fit_transform(S) #standaryzacja

A = np.array([[1, 1, 1], [0.5, 2, 1.0], [1.5, 1.0, 2.0]])
#macierz mieszająca
X = np.dot(S, A) #iloczyn skalarny macierzy S i A
```

Po zmiksowaniu sygnałów można spróbować odtworzyć sygnały składowe za pomocą algorytmu ICA, podając do funkcji liczbę składowych, na które sygnał zmieszany zostanie zdekomponowany. W tym przypadku wiadomo, że sygnał zmieszany zawiera w sobie trzy sygnały ukryte, więc do funkcji FastICA podany został argument n_components = 3.

```
from sklearn.decomposition import FastICA

ica = FastICA(n_components = 3)
S_reconstructed = ica.fit_transform(X) #rekonstrukcja sygnałów
A_reconstructed = ica.mixing_ #estymata macierzy mieszającej
```

Algorytm ICA posłużył do wyznaczenia dwóch macierzy:
1) S_reconstructed, która zawiera zrekonstruowane sygnały ukryte,
2) A_reconstructed, która jest estymatą macierzy mieszającej.

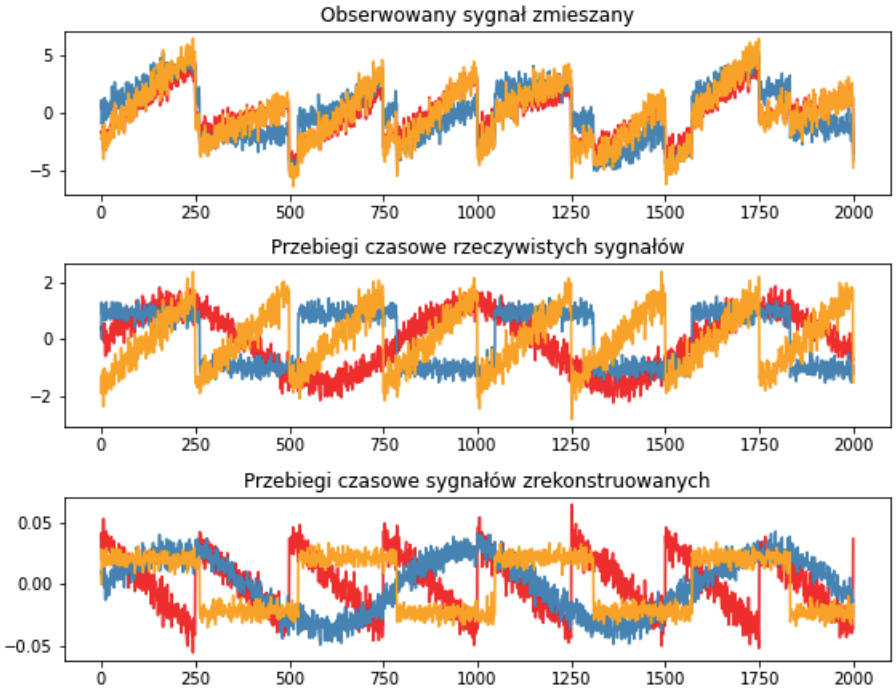
W omawianym przykładzie wiadomo, która macierz mieszająca została użyta, jednak podczas separacji rzeczywistych nagrań nie będzie ona znana i można wtedy jedynie wyestymować wartości, które się w niej znajdują.

UWAGA

Określenie liczby składowych jest konieczne i do jej właściwego doboru potrzebna jest wiedza, z ilu źródeł pochodzą sygnały ukryte. W rzeczywistości należy tę liczbę dobrać, opierając się na dostępnych informacjach o sygnale, np. po przesłuchaniu nagrania lub po zapoznaniu się z opisem przebiegu rejestracji sygnału.

Skuteczność algorytmu ICA w rekonstrukcji sygnałów ukrytych, a tym samym separacji źródeł, można w prosty sposób zweryfikować, porównując wykresy sygnałów rzeczywistych i zrekonstruowanych (rys. 9.2).

```
import matplotlib.pyplot as plt
plt.figure()
models = [X, S, S_reconstructed]|names = [
                              'Obserwowany sygnał zmieszany',
                              'Przebiegi czasowe rzeczywistych sygnałów',
                              'Przebiegi czasowe sygnałów zrekonstruowanych']
colors = ['red', 'steelblue', 'orange']
for ii, (model, name) in enumerate(zip(models, names), 1):
 plt.subplot(4, 1, ii)
 plt.title(name)
 for sig, color in zip(model.T, colors):
   plt.plot(sig, color = color)
plt.tight_layout()
plt.show()
```



Rysunek 9.2. Przebiegi czasowe sygnałów zmieszanych (na górze), rzeczywistych sygnałów ukrytych (w środku) oraz sygnałów ukrytych zrekonstruowanych przy użyciu algorytmu ICA (na dole)

Do sprawdzenia, czy sygnały zrekonstruowane są równe sygnałom oryginalnym z określoną, niewielką tolerancją, można użyć funkcji assert oraz np.allclose. Jeżeli funkcja assert nie zwróci błędu, to sprawdzany warunek jest spełniony. W tym przypadku weryfikowane jest, czy iloczyn skalarny zrekonstruowanych sygnałów i estymaty macierzy miksowania jest równy iloczynowi skalarnemu wygenerowanych sygnałów składowych i macierzy miksowania. Jeżeli warunek jest spełniony, to separacja źródeł dźwięku może zostać uznana na udaną.

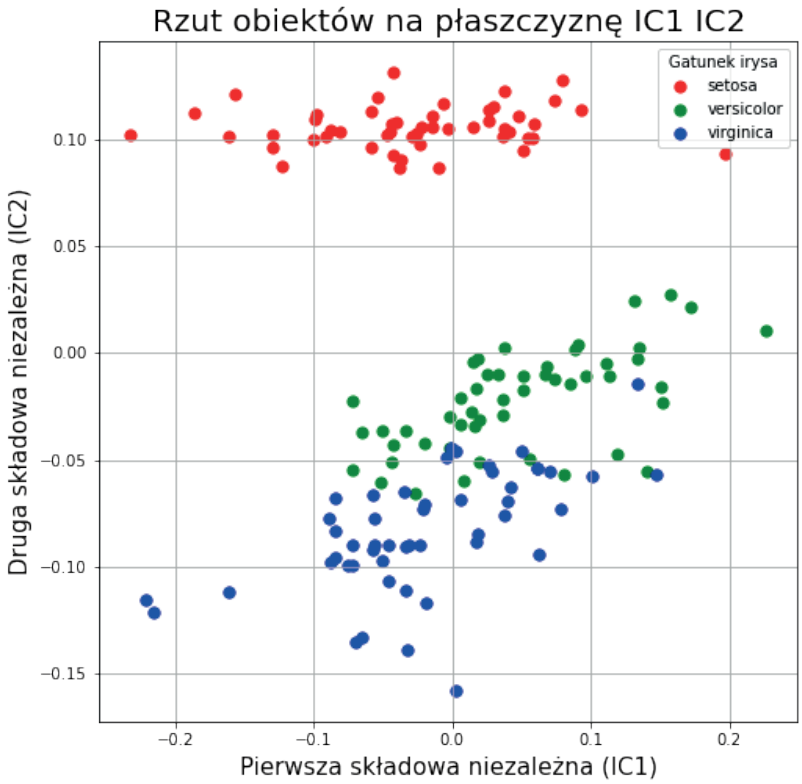
```
assert np.allclose(X,np.dot(S_reconstructed,
                      A_reconstructed.T) + ica.mean_)
```

W omawianym przypadku zarówno wizualna ocena rekonstrukcji sygnałów, jak i metoda wykorzystująca funkcje assert oraz np.allclose wskazują na satysfakcjonujące efekty separacji źródeł.

Przykład 9.3. Redukcja wymiarowości przy użyciu algorytmu ICA

Analiza składowych niezależnych może również posłużyć do redukcji wymiarowości – przykładowe użycie ICA w tym celu zostanie przedstawione na przykładzie bazy danych *iris* opisanej w Przykładzie 8.1.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import FastICA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
labels = iris.target_names
X = StandardScaler().fit_transform(X) #standaryzacja danych

#wyznaczone zostaną tylko dwie składowe niezależne, które posłużą
#do wizualizacji danych na utworzonej przez nie płaszczyźnie
ica = FastICA(n_components = 2)
independent_components = ica.fit_transform(X)
ica_df = pd.DataFrame(
          data = (np.hstack((independent_components, y[:, np.newaxis]))),
          columns = ['IC1', 'IC2', 'numer klasy'])
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pierwsza składowa niezależna (IC1)', fontsize = 15)
ax.set_ylabel('Druga składowa niezależna (IC2)', fontsize = 15)
ax.set_title('Rzut obiektów na płaszczyznę IC1 IC2', fontsize = 20)
targets = [0, 1, 2]
colors = ['r', 'g', 'b']
for target, color in zip(targets, colors):
  indicesToKeep = ica_df['numer klasy'] == target
  ax.scatter(ica_df.loc[indicesToKeep, 'IC1'],
             ica_df.loc[indicesToKeep, 'IC2'],
             c = color,
             s = 50)
ax.legend(labels, title = 'Gatunek irysa')
ax.grid()
plt.show()
```



Rysunek 9.3. Rzut obiektów na płaszczyznę utworzoną przez dwie składowe niezależne.

Porównanie rysunków 9.2 oraz 9.3 uwidacznia, że rezultaty uzyskane przy użyciu ICA różnią się od tych uzyskanych przy użyciu PCA, choć w przypadku analizowanych danych różnice te nie są duże. Oznacza to, że w zależności od rozwiązywanego problemu algorytmy te mogą dawać istotnie różne wyniki i dobrą praktyką jest testowanie różnych technik redukcji wymiarowości w celu znalezienia najlepszego rozwiązania.

UWAGA

Nie ma ściśle określonych reguł pozwalających na wyznaczenie liczby składowych głównych w celu redukcji wymiarowości – ich liczba jest zazwyczaj dobierana metodą optymalizacji pozwalającej zapewnić jak najlepsze działanie modelu, który będzie uczony na danych zredukowanych. Wyodrębnione składowe nie muszą mieć fizycznej interpretacji, ponieważ celem algorytmu nie jest w tym przypadku separacja sygnałów ukrytych, a jedynie zawarcie dużej części informacji w jak najmniejszej liczbie składowych.
Jest możliwe wyznaczenie większej liczby składowych niezależnych niż liczba cech opisujących analizowane obiekty, jednak nie ma to sensu, jeżeli algorytm ma posłużyć do redukcji wymiarowości.

Zadanie 9.2. Wyodrębnianie dźwięków różnych instrumentów muzycznych z nagrania

1. Pobierz bazę nagrań instrumentów muzycznych dostępną pod adresem: https://data.vision.ee.ethz.ch/cvl/ae_dataset/ i wybierz z niej trzy sygnały, np. violin_0.wav, violin_162.wav, rustle_177.wav.
2. Wygeneruj miks sygnałów, a następnie zastosuj algorytm ICA do odtworzenia trzech zaszumionych sygnałów składowych. Sygnały są rzeczywistymi nagraniami, więc zawierają w sobie szum – możesz jednak dodać do nich szum gaussowski, tak jak pokazano w przykładzie powyżej. Zastosuj dowolną macierz mieszającą, by uzyskać sygnał zmieszany zawierający dźwięki wszystkich instrumentów.

3. Wyświetl na wykresie:
   – sygnał obserwowany (zmiksowane sygnały składowe),
   – rzeczywiste sygnały ukryte,
   – zrekonstruowane sygnały ukryte.
   Zwróć uwagę na amplitudy zrekonstruowanych sygnałów wyświetlonych na wykresie. Jeżeli są bardzo małe, należy przeskalować je, używając poniższego kodu.

```
S1_reconstructed = S_reconstructed[:,0] * (
          2 ** 15 - 1)/np.max(np.abs(S_reconstructed[:,0]))
```

UWAGA

Powyższy kod jest odpowiedni, jeżeli macierz S_reconstructed zawiera dane typu *int16*. Jeżeli są to dane innego typu, należy ten kod odpowiednio zmodyfikować.

4. Zapisz zrekonstruowane sygnały do pliku .wav, np. przy użyciu funkcji write z biblioteki scipy znajdującej się pod adresem: https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.wavfile.write.html.
5. Sprawdź, czy odtworzone sygnały brzmią tak samo jak oryginalne.

10 Algorytm *k*-średnich

Zastosowanie: klasteryzacja.
Metoda: uczenie nienadzorowane.

Ogólne informacje dotyczące klasteryzacji można znaleźć w rozdziale 4. W klasteryzacji metodą *k*-średnich dąży się do zmaksymalizowania podobieństwa w skupieniu i maksymalizacji różnic pomiędzy skupieniami. Jest to osiągane w wyniku przenoszenia obiektów między skupieniami aż do momentu zoptymalizowania zmienności wewnątrz skupień i pomiędzy nimi (MacQueen, 1967; Hartigan i Wong, 1979).

Kroki algorytmu *k*-średnich

1. Ustalenie liczby skupień (co odpowiada liczbie klas, na które zostaną podzielone dane). Przyjmuje się tu różne metody, np.:
   – umowny wybór i ewentualne późniejsze zmiany w celu poszukiwania lepszych wyników;
   – wybór oparty na wynikach innych analiz i wiedzy z zakresu badanego zjawiska.
2. Ustalenie warunku zatrzymania (stopu), np.:
   – liczba iteracji;
   – brak przesunięć obiektów pomiędzy skupieniami.
3. Wybór metryki (definicję i przykładowe metryki można znaleźć w dodatku).
4. Ustalenie środków skupień (centroidów):
   – w pierwszej iteracji np.:
     • losowy wybór *k* obserwacji,
     • wybór *k* pierwszych obserwacji,
     • dobór pod kątem zmaksymalizowania odległości skupień;
   – w kolejnych iteracjach np. średnia arytmetyczna współrzędnych punktów należących do danego skupienia.
5. Obliczenie odległości obiektów od środków skupień.
6. Przypisanie obiektów do skupień – dla każdej obserwacji porównujemy jej odległości od każdego ze skupień i przypisujemy tę obserwację do skupienia, do którego środka ma najbliżej.
7. Sprawdzenie warunku zatrzymania:
   ◦ jeżeli warunek zatrzymania nie jest spełniony, to należy wrócić do kroku nr 4,
   • jeżeli warunek zatrzymania jest spełniony, to grupowanie jest zakończone.

UWAGA

Pierwszy krok algorytmu może być poprzedzony redukcją wymiaru.

Przykład 10.1. Podział utworów muzycznych na gatunki

Zastosowanie praktyczne algorytmu *k*-średnich przedstawione zostanie na przykładzie danych dotyczących gatunków muzycznych, możliwych do pobrania ze strony: https://github.com/mdeff/fma.

W bazie danych zawarte są wartości ośmiu cech akustycznych opisujących utwory muzyczne:
1) acousticness – wysoka wartość, jeżeli jest to wersja akustyczna utworu,
2) danceability – określa, czy utwór nadaje się do tańca,
3) energy – wysoka wartość, jeżeli utwór jest energiczny, głośny i szybki,
4) instrumentalness – im wyższa wartość, tym więcej jest partii czysto instrumentalnych,
5) liveness – większa wartość, jeżeli w nagraniu występuje publiczność (nagrania koncertowe),
6) speechiness – im wyższa wartość, tym więcej wypowiedzianych słów zawartych jest w utworze,
7) tempo – tempo utworu wyrażone w BPM (ang. *beats per minute*),
8) valence – im wyższa wartość, tym bardziej pozytywne emocje wzbudza dany utwór.

```
import pandas as pd
import numpy as np

#wczytanie cech utworów
feats_col_list = [0,1,2,3,4,5,6,7,8]
#wczytane będą tylko kolumny z indeksem utworu i cechami akustycznymi

feats_df = pd.read_csv("fma_metadata/echonest.csv",
                       usecols = feats_col_list,
                       low_memory = False,
                       header = 2)
#header - numer wiersza, w którym jest nazwa cechy i który będzie
#nagłówkiem kolumn
#low_memory = False - dzięki temu nie wyświetla się ostrzeżenie,
#że wczytywane są różne typy danych do jednej tabeli

feats_df.rename(columns = {'Unnamed: 0': 'track_id'},
                inplace = True)
#zmiana nagłówka pierwszej kolumny
#inplace = True oznacza, że modyfikowana jest oryginalna tabela
#feats_df,inplace = False oznacza, że utworzona zostanie kopia tabeli,
#która będzie modyfikowana

feats_df.dropna(inplace = True)
#usunięcie wierszy, w których jest NaN (pusta komórka w pliku .CSV)

feats_df.reset_index(drop = True, inplace = True)

#przeprowadzenie nowego indeksowania tabeli - podczas usuwania wierszy
#usunięte zostały też ich indeksy
#drop = True oznacza, że nie będzie tworzona nowa kolumna zawierająca
#stare indeksy

#wczytanie przynależności poszczególnych utworów do gatunków muzycznych
genre_col_list = [0,40]
```

```
genre_df = pd.read_csv("fma_metadata/tracks.csv",
                       usecols = genre_col_list,
                       low_memory = False,
                       header = 1)
genre_df.rename(columns = {'Unnamed: 0': 'track_id'}, inplace = True)
genre_df.dropna(inplace = True)
genre_df.reset_index(drop = True, inplace = True)
```

Przed przystąpieniem do dalszych analiz warto sprawdzić, jakie gatunki muzyczne są dostępne w bazie danych.

```
print(genre_df["genre_top"].unique())
```

Gatunki są określone słownie – takie etykiety nie mogą zostać użyte w większości algorytmów uczenia maszynowego. Należy więc każdemu gatunkowi przypisać wartość liczbową, która będzie służyła jako etykieta. Następnie obie utworzone dotychczas tabele – z cechami akustycznymi oraz gatunkami i etykietami poszczególnych utworów – należy połączyć w taki sposób, aby zachować tylko indeksy utworów, które występują w obu tabelach (how = 'inner'), i żeby dodawane kolumny były posortowane w ten sam sposób.

```
#tworzenie słownika, który posłuży do zamiany etykiet słownych na numeryczne
labels_dict = {value: index for index, value in enumerate(
                                          genre_df["genre_top"].unique())}

#dodanie nowej kolumny do tabeli genre_df i wpisanie do niej etykiet
#numerycznych poszczególnych utworów
genre_df['genre_numeric_label'] = genre_df["genre_top"].map(labels_dict)
all_data_df = pd.merge(feats_df, genre_df, on = 'track_id', how = 'inner')
#argument on określa, względem której kolumny zostaną połączone tabele
```

Każdy utwór opisywany jest przez osiem cech. Jest to zbyt dużo, żeby można je było łatwo zwizualizować na płaszczyźnie i zobaczyć, czy tworzą się skupienia zawierające utwory z poszczególnych gatunków. Żeby rozwiązać ten problem, zostanie przeprowadzona analiza składowych głównych (PCA), a dane zostaną przedstawione na płaszczyźnie PC1-PC2 (tworzonej przez pierwsze dwie składowe główne).

Dane nie będą wykorzystywane do klasyfikacji, więc do PCA można podać wszystkie dane, bez podziału na zbiór uczący i testowy. Po zastosowaniu algorytmu PCA warto sprawdzić, jaka część wariancji jest wyjaśniania przez kolejne składowe główne.

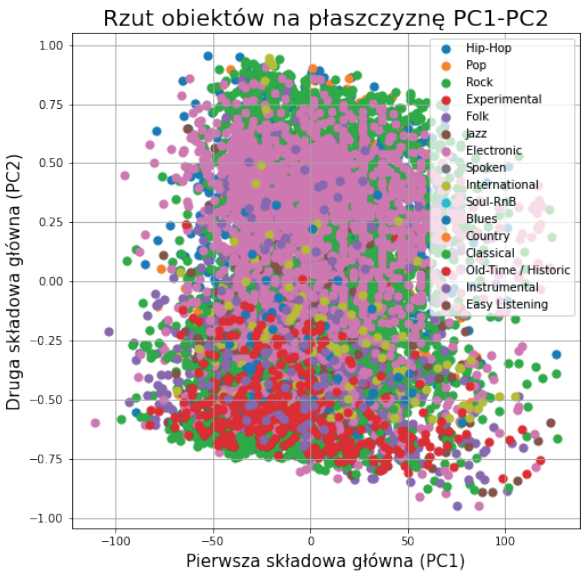
```
from sklearn.decomposition import PCA
pca = PCA(n_components = 8)
#feats_names - wektor zawierający nazwy cech, które będą wykorzystane
#do przeprowadzenia PCA.
feats_names = ["acousticness", "danceability", "energy",
               "instrumentalness", "liveness", "speechiness",
               "tempo", "valence"]
PCA_feats = pca.fit_transform(all_data_df[feats_names])
pca.explained_variance_ratio_
```

Niemal cała wariancja jest wyjaśniana przez zaledwie jedną składową główną. Do tabeli z danymi dodane zostaną jednak dwie składowe, PC1 i PC2, dzięki czemu możliwe będzie narysowanie wykresu na nowej płaszczyźnie.

```
all_data_df['PC1'] = PCA_feats[:,0]
all_data_df['PC2'] = PCA_feats[:,1]
```

Teraz można przystąpić do wizualizacji danych. Na osi *X* będzie PC1, na osi *Y* PC2. Każdy gatunek muzyczny zostanie oznaczony innym kolorem (rys. 10.1).

```
from matplotlib import pyplot as plt

fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
#podpis osi X
ax.set_xlabel('Pierwsza składowa główna (PC1)', fontsize = 15)
#podpis osi Y
ax.set_ylabel('Druga składowa główna (PC2)', fontsize = 15)
#tytuł wykresu
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2', fontsize = 20)
targets = range(0,16) #16, ponieważ tyle jest gatunków muzycznych
for target in targets:
 indicesToKeep = all_data_df['genre_numeric_label'] == target
 ax.scatter(all_data_df.loc[indicesToKeep, 'PC1'],
            all_data_df.loc[indicesToKeep, 'PC2'],
            s = 50) #rozmiar markera
ax.legend(genre_df["genre_top"].unique())
ax.grid()
```



Rysunek 10.1. Podział utworów na gatunki muzyczne przedstawiony na płaszczyźnie utworzonej przez dwie pierwsze składowe główne

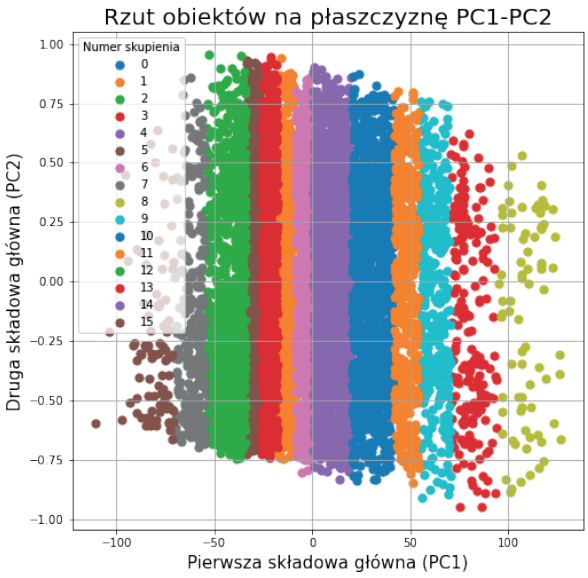
Mimo że składowe PC1 i PC2 wyjaśniają prawie całą wariancję, nie pozwalają one odróżnić od siebie poszczególnych gatunków muzycznych. Na wykresie nakładają się na siebie i nie są widoczne skupienia wyraźnie od siebie oddzielone.

Ciekawym zastosowaniem algorytmu *k*-średnich jest wydzielenie za jego pomocą nowych gatunków muzycznych, których odróżnienie jest możliwe na podstawie analizowanych w tym przykładzie cech akustycznych.

Pierwszym etapem jest określenie liczby skupień (klastrów), czyli nowych gatunków muzycznych. Na początek przyjęta zostanie liczba 16, odpowiadająca liczbie rzeczywistych gatunków muzycznych. Podział obiektów (utworów) na skupienia zostanie przeprowadzony na podstawie wartości dwóch pierwszych składowych głównych, dzięki czemu wizualizacja rezultatów będzie prosta.

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters = 16, random_state = 0, n_init = 10)
#n_init - liczba losowych podziałów na skupienia, które zostaną wykonane;
#ostateczny podział jest wybierany spośród n wykonanych podziałów tak,
#by odległości centroidów od wszystkich obiektów w skupieniu były
#możliwie najmniejsze; algorytm wyznacza skupienia w danych oraz
#przypisuje obserwacjom nowe etykiety (nowe gatunki)

kmeans_labels = kmeans.fit_predict(PCA_feats[:,:2])
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)

#podpis osi X
ax.set_xlabel('Pierwsza składowa główna (PC1)', fontsize = 15)
#podpis osi Y
ax.set_ylabel('Druga składowa główna (PC2)', fontsize = 15)

#tytuł wykresu
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2', fontsize = 20)

targets = range(0,16)
for target in targets:
 indicesToKeep = kmeans_labels == target
 ax.scatter(all_data_df.loc[indicesToKeep, 'PC1'],
            all_data_df.loc[indicesToKeep, 'PC2'],
            s = 50)
ax.legend(targets)
ax.grid()
```

Jak widać na rysunku 10.2, do wydzielenia klastrów wystarczy tylko pierwsza składowa główna, dlatego uzyskano pionowe pasy tworzące oddzielne skupienia. Druga składowa posłużyła praktycznie tylko do tego, by można było rzutować punkty na płaszczyznę i lepiej je zwizualizować. Wynika to z tego, że pierwsza składowa główna wyjaśnia aż 99,96% zmienności. Co więcej, skupienia co prawda są utworzone, ale stykają się ze sobą i bez zastosowania różnych kolorów nie dałoby się ich zauważyć na wykresie i oddzielić od siebie, więc jest to podział bardzo arbitralny. Może to mieć związek ze zbyt dużą liczbą gatunków oraz cech opisujących sygnały, na których użyto algorytmu PCA.



Rysunek 10.2. Skupienia zawierające utwory muzyczne wyznaczone metodą *k*-średnich przedstawione na płaszczyźnie utworzonej przez dwie pierwsze składowe główne

Zadanie 10.1. Podział utworów muzycznych na gatunki – wpływ liczby cech i liczby klastrów na wyniki klasteryzacji

Spróbuj uzyskać lepszą klasteryzację danych przedstawionych w przykładzie powyżej:
- zmniejszając liczbę klastrów, na które dane mają być podzielone przez algorytm *k*-średnich,
- zmniejszając liczbę cech, które podawane są do PCA – wybierz takie, które według Ciebie najlepiej będą się nadawały do odróżniania od siebie gatunków muzycznych (istniejących lub nowych),
- zmniejszając zarówno liczbę cech, jak i liczbę klastrów,
- wykonując standaryzację danych przed użyciem algorytmu PCA.

Porównaj wyniki uzyskane metodami zaproponowanymi powyżej. Czy standaryzacja mocno wpłynęła na uzyskane wyniki, w szczególności na rozmieszczenie klastrów i ich separowalność?

Zadanie 10.2. Podział utworów muzycznych na gatunki – wpływ analizy składowych głównych na wyniki klasteryzacji

Omówiony przykład zakładał tworzenie skupień na podstawie wyznaczonych uprzednio składowych głównych. Alternatywnym rozwiązaniem jest zastosowanie algorytmu *k*-średnich na cechach akustycznych – PCA służy wówczas wyłącznie do wizualizacji danych i nie ma wpływu na uzyskiwane skupienia. Który podział uważasz za lepszy – ten na podstawie cech akustycznych czy składowych głównych? Który z nich pozwala według Ciebie na prostszą interpretację wpływu cech akustycznych na przyporządkowanie utworów do gatunków muzycznych?

11 Algorytmy hierarchiczne

Zastosowanie: klasteryzacja.
Metoda: uczenie nienadzorowane.

Poznana w rozdziale 10 metoda klasteryzacji należy do technik grupowania opartego na podziale, czyli na początku ma miejsce losowy podział na określoną liczbę skupień, a następnie poszukiwane jest rozmieszczenie obiektów, które będzie optymalizować ustaloną funkcję oceny grupowania. Narzędziami alternatywnymi są techniki grupowania hierarchicznego, w których klasteryzacja następuje w wyniku stopniowego łączenia punktów w skupienia lub stopniowego dzielenia tzw. nadskupień. Tym samym w grupie algorytmów hierarchicznych można wyszczególnić metody aglomeracyjne i deaglomeracyjne, zwane również dzielącymi.

11.1. Miara odległości między skupieniami

Należy zauważyć, że w metodach hierarchicznych nie określa się jawnej globalnej funkcji oceny. Obliczane są natomiast lokalne oceny par skupień tworzonego hierarchicznego grupowania i na ich podstawie podejmowana jest decyzja o połączeniu lub rozdzieleniu obiektów, zależnie do przyjętej metody grupowania hierarchicznego. Wybór metody oceny par skupień jest bardzo ważny i ma wpływ na ostateczny wynik grupowania.

Niezależnie od tego, czy stosowana jest metoda aglomeracyjna, czy deaglomeracyjna, w obydwu przypadkach podstawową miarą oceny wykorzystywaną do podjęcia decyzji o grupowaniu lub dzieleniu, jest odległość dwóch skupień. Najczęściej stosowane miary odległości między dwoma skupieniami zdefiniowano poniżej (Sneath i Sokal, 1973).

DEFINICJA

Niech
C_i i C_j – niepuste skupienia,
$d_e(a, b)$ – odległość między a i b w metryce euklidesowej (definicja metryki euklidesowej w dodatku).

Wtedy odległość między dwoma skupieniami $d(C_i, C_j)$, w zależności od wybranej miary, dana jest wzorami:

– pojedyncze połączenie (ang. *single link*)

$$d(C_i, C_j) = \min_{a \in C_i,\, b \in C_j} d_e(a, b), \tag{11.1}$$

– całkowite połączenie (ang. *complete link*)

$$d(C_i, C_j) = \max_{a \in C_i,\, b \in C_j} d_e(a, b), \tag{11.2}$$

– odległość między środkami (ang. *mean distance*)

$$d(C_i, C_j) = d_e(m_i, m_j), \tag{11.3}$$

gdzie m_i i m_j to środki ciężkości odpowiednio skupienia C_i i C_j,

– średnia odległość (ang. *average distance*)

$$d(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{b \in C_j} \sum_{a \in C_i} d_e(a, b). \tag{11.4}$$

11.2. Metody aglomeracyjne

W metodach aglomeracyjnych początkowo każdy obiekt tworzy osobny klaster (skupienie). Następnie dwa najbliższe skupienia (pod względem wybranej metryki odległości) scalane są w jeden klaster i operacja ta jest powtarzana do momentu, aż powstanie jedno duże skupienie zawierające wszystkie dane (Ward, 1963; Hand i in., 2005). Ostatnim krokiem w klasteryzacji metodą aglomeracyjną jest ustalenie liczby poziomów, które prowadzą do uzyskania zadowalającego grupowania.

Kroki algorytmu (metody aglomeracyjne)

1. Wybór miary określającej odległość między dwoma skupieniami.
2. Podział na skupienia przez przydzielenie każdego obiektu w zbiorze treningowym do oddzielnego skupienia. Zatem jeżeli zgodnie z przyjętymi oznaczeniami $X = \{x^{(j)}, j = 1, ..., N\}$, gdzie N to N-elementowy zbiór danych treningowych, to w pierwszym kroku wykonywania algorytmu powstanie N skupień C_j, $j = 1, ..., N$.
3. Niech $t := 1$.
4. Znalezienie dwóch skupień C_k, C_l minimalizujących funkcję odległości między dwoma skupieniami.
5. $C_k = C_k \cup C_l$ oraz C_l zostaje usunięty.
6. Przenumerowanie wszystkich skupień $j = 1, \ldots, N - t$.
7. Jeżeli $N - t \neq 1$, to $t := t + 1$ i powrót do kroku nr 4.
8. Ustalenie liczby poziomów zapewniających zadowalające grupowanie. Obcięcie dendogramu na wybranym poziomie.

11.3. Metody deaglomeracyjne

Metody deaglomeracyjne polegają na postępowaniu przeciwnym niż w metodach aglomeracyjnych, czyli najpierw wszystkie obiekty znajdują się w jednym skupieniu, a następnie zostają stopniowo dzielone na mniejsze zawierające tylko te najbardziej podobne do siebie obiekty, aż do momentu, gdy w każdym skupieniu znajduje się tylko jeden obiekt (Ward, 1963; Hand i in., 2005). Wynikowe skupienia danych otrzymuje się przez obcięcie dendrogramu na wybranym poziomie. Czyli podobnie jak w metodach aglomeracyjnych, ostatnim krokiem klasteryzacji jest ustalenie liczby poziomów, które prowadzą do uzyskania zadowalającego grupowania.

Kroki algorytmu (metody deaglomeracyjne)

1. Wybór miary określającej odległość między dwoma skupieniami.
2. Utworzenie pierwszego skupienia zawierającego wszystkie obiekty ze zbioru treningowego $X = \{x^{(j)}, j = 1, ..., N\}$.
3. Niech $t := 1$.
4. Znalezienie skupienia C_i ze zbioru t skupień takiego, że istnieje podział na dwa skupienia C_k, C_l maksymalizujący funkcję odległości między dwoma skupieniami.
5. Podzielenie wybranego w kroku nr 4 skupienia C_i na skupienia C_k, C_l (C_i zostaje usunięty).
6. Przenumerowanie skupień $i = 1, \ldots, t + 1$.
7. Jeżeli $t + 1 \neq N$, to $t := t + 1$ i powtórzenie kroków 4 oraz 5.
8. Ustalenie liczby poziomów zapewniających zadowalające grupowanie. Obcięcie dendogramu na wybranym poziomie.

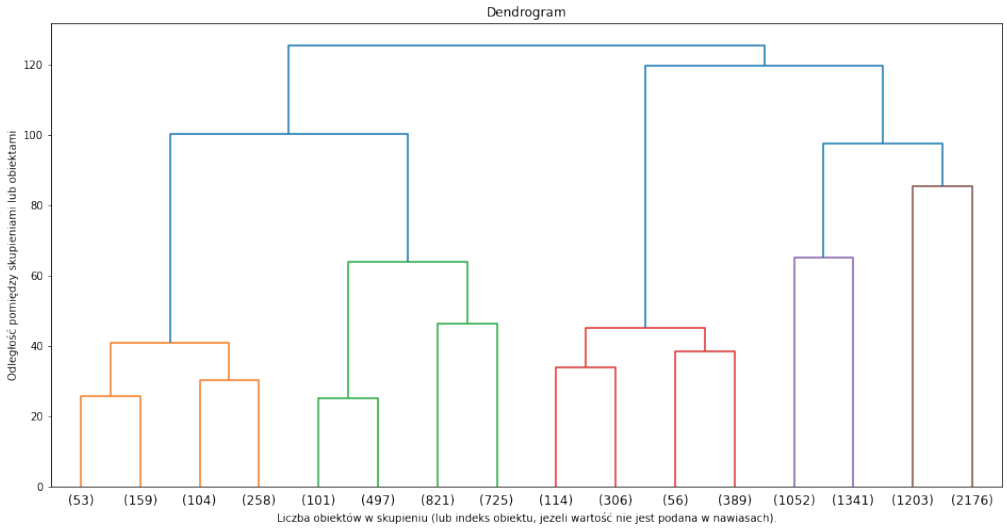
Dużą zaletą algorytmów hierarchicznych jest możliwość przejrzystej wizualizacji kolejnych połączeń na wykresie zwanym dendrogramem, który jest graficzną formą prezentacji hierarchii skupień. Przykładowy dendrogram zostanie przedstawiony w poniższym przykładzie.

Przykład 11.1. Podział utworów muzycznych na gatunki

Przykładowe przeprowadzenie klasteryzacji przy użyciu algorytmów hierarchicznych zostanie przedstawione z wykorzystaniem danych z Przykładu 10.1.

Wyniki klasteryzacji hierarchicznej często przedstawia się w postaci dendrogramów. W Pythonie można to zrobić, używając funkcji plot_dendrogram dostępnej w dokumentacji biblioteki scikit-learn znajdującej się pod adresem: https://scikit-learn.org/stable/auto_examples/cluster/plot_agglomerative_dendrogram.html#sphx-glr-auto-examples-cluster-plot-agglomerative-dendrogram-py.

Otrzymany dendogram zamieszczono na rysunku 11.1.



Rysunek 11.1. Dendrogram przedstawiający podział analizowanych danych na skupienia wyznaczone przy użyciu algorytmu aglomeracyjnego

Dzięki zastosowaniu metod hierarchicznych możliwe jest wygenerowanie etykiet przyporządkowujących obiekty do utworzonych skupień. Może to posłużyć do eksploracji danych i sprawdzenia, jakie skupienia zostaną wyznaczone i które obiekty będą do nich przynależeć. Innym zastosowaniem może być wygenerowanie etykiet, które na dalszych etapach będą służyć do uczenia algorytmów z nadzorem (algorytmów klasyfikacyjnych).

```
model = AgglomerativeClustering(n_clusters=16, affinity = 'euclidean')
#n_clusters - liczba skupień, do których zostaną przyporządkowane
#analizowane obiekty

clusters = model.fit_predict(scaled_feats)

dendrogram_clusters_df = pd.DataFrame()
dendrogram_clusters_df['track_id'] = all_data_df["track_id"]
dendrogram_clusters_df['real genre'] = all_data_df["genre_top"]
dendrogram_clusters_df['dendrogram clusters'] = clusters
```

W celu wygenerowania wektora etykiet należy użyć kodu przedstawionego poniżej. W zmiennej clusters zostaną zapisane numery skupień, do których przyporządkowane są obiekty. W celu sprawdzenia, czy podział na gatunki muzyczne pokrywa się z tym, który uzyskano w analizie aglomeracyjnej, najlepiej będzie dodać obie zmienne do nowej struktury DataFrame.

Dzięki umieszczeniu rezultatów klasteryzacji w strukturze DataFrame możliwe i proste do przeprowadzenia jest filtrowanie gatunków muzycznych. Umożliwia to sprawdzenie, do ilu skupień zostały przyporządkowane utwory danego gatunku i czy wyniki klasteryzacji są zgodne z rzeczywistym podziałem utworów na gatunki muzyczne. Poniżej przedstawiono przykład dla muzyki klasycznej.

```
#wyświetlenie fragmentu tabeli z muzyką klasyczną - w pierwszej kolumnie
#wyświetli się indeks wiersza, w drugiej gatunek, w trzeciej numer
#skupienia
print(dendrogram_clusters_df.loc[dendrogram_clusters_df[
                                    'real genre'] == 'Classical'].head(10))
#head(10) oznacza, że wyświetlone zostanie pierwsze 10 wierszy
#loc użyte na strukturze DataFrame spowoduje zlokalizowanie
#i wyświetlenie tych wierszy, które spełniają zadany warunek;
#sprawdzenie, jakie numery skupień pojawiają się dla utworów klasycznych
dendrogram_clusters_df.loc[dendrogram_clusters_df[
                                  'real genre'] == 'Classical'][
                                           'dendrogram clusters'].unique()
```

Analizując uzyskane wyniki, można stwierdzić, że utwory klasyczne zostały przyporządkowane do siedmiu spośród szesnastu utworzonych skupień.

Zadanie 11.1. Podział utworów muzycznych na gatunki – analiza przyporządkowania obiektów do skupień

Przeprowadź analizę analogiczną do przedstawionej w powyższym przykładzie, tworząc tylko cztery skupienia ze wszystkich obiektów. Sprawdź, do jakich skupień zostały przyporządkowane utwory z wybranego gatunku. Następnie przeprowadź odwrotną analizę – sprawdź, które gatunki zostały przyporządkowane do każdego z czterech skupień. Spróbuj wyznaczyć liczbę utworów każdego gatunku, które zostały przyporządkowane do tych skupień. Czy zostały podzielone równomiernie, czy większość utworów danego gatunku znajduje się w tym samym skupieniu?

UWAGA

W celu uzyskania obiektów spełniających więcej niż jeden warunek równocześnie należy zastosować operatory bitowe (| lub &). Połączenie warunków operatorami logicznymi (*or* lub *and*) spowoduje zwrócenie błędu.

Zadanie 11.2. Podział utworów muzycznych na gatunki – analiza wpływu metryki na wyniki klasteryzacji hierarchicznej

Sprawdź, jak na wyniki wpłyną zmiana metryki odległości na inną niż euklidesowa oraz zmiana metody tworzenia skupień na inną niż metoda Warda. Do wyboru są:

- metryki (affinity): 'euclidean', 'l1', 'l2', 'manhattan', 'cosine',
- metody (linkage): 'ward', 'complete', 'average', 'single'.

UWAGA

Wybierając metodę Warda, można użyć wyłącznie metryki euklidesowej.

12 Regresja liniowa

Zastosowanie: regresja.
Metoda: uczenie nadzorowane.

Problemy rozwiązywane przy użyciu modelu regresji liniowej dotyczą zwykle próby uchwycenia zależności pomiędzy zmiennymi, co pozwala na docelową predykcję wartości atrybutu decyzyjnego dla dowolnego obiektu spoza zbiorów treningowego i testowego. Ponieważ mamy do czynienia z wyraźną zależnością zmiennych, atrybuty warunkowe nazywa się najczęściej zmiennymi niezależnymi, natomiast atrybut decyzyjny zwany jest zwykle zmienną zależną. Więcej szczegółów można znaleźć w rozdziale 4.

Model regresji liniowej opisuje dobrze zmienne, które są zależne liniowo. Wtedy hipotezę w trenowanym modelu można zdefiniować następująco:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_d x_d, \tag{12.1}$$

gdzie θ_i parametryzują przestrzeń funkcji liniowych $X \rightarrow Y$, czyli są to zmienne, których wartości będą poszukiwane na etapie treningu. Niech $x_0 = 1$. Wtedy hipotezę $h_\theta(x)$ można zapisać w następującej postaci:

$$h_\theta(x) = \sum_{i=0}^{d} \theta_i x_i, \tag{12.2}$$

czyli dla j-tego przykładu ze zbioru treningowego ($j = 1, ..., N$) otrzymujemy:

$$h_\theta(x^{(j)}) = \sum_{i=1}^{d} \theta_i x_i^{(j)}, \tag{12.3}$$

co można również zapisać wektorowo:

$$h_\theta(x^{(j)}) = \theta^T x^{(j)}, \tag{12.4}$$

gdzie:

$$\theta = [\theta_0, ..., \theta_d]^T,$$
$$x^{(j)} = [x_0^{(j)}, ..., x_d^{(j)}]^T,$$
$$\forall \; j = 1, ..., N \quad x_0^{(j)} = 1.$$

Rozwiązanie problemu regresji sprowadza się zatem do wyznaczenia takiego wektora θ, dla którego funkcja h najlepiej opisuje zależność liniową atrybutu decyzyjnego od atrybutów warunkowych. Ponieważ model zbudowany na danych empirycznych jest z pewnością obarczony pewnym błędem, możemy zapisać:

$$y^{(j)} = \theta^T x^{(j)} + \epsilon^{(j)}, \tag{12.5}$$

gdzie $\epsilon^{(j)}$ – błąd spowodowany czynnikiem losowym lub czynnikami nieuwzględnionymi w modelu regresji. $\epsilon^{(j)}$ można potraktować jako zmienne niezależne o rozkładzie normalnym $N(0, \sigma^2)$ (Johnson i Wichern, 2007).

Funkcja gęstości prawdopodobieństwa $\epsilon^{(j)}$ dana jest następującym równaniem:

$$p(\epsilon^{(j)}) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\epsilon^{(j)})^2}{2\sigma^2}}. \tag{12.6}$$

Zatem funkcja gęstości prawdopodobieństwa zmiennej $y^{(j)}$ z daną zmienną $x^{(j)}$ sparametryzowaną przez θ jest postaci:

$$p(y^{(j)} \mid x^{(j)}; \theta) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y^{(j)} - \theta^T x^{(j)})^2}{2\sigma^2}}. \tag{12.7}$$

Wtedy $p(y|X; \theta)$ jest funkcją θ przy ustalonych X i y, przy czym:

$$X = \begin{bmatrix} 1 & x_1^{(1)} & \dots & x_d^{(1)} \\ 1 & x_1^{(2)} & \dots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(N)} & \dots & x_d^{(N)} \end{bmatrix}. \tag{12.8}$$

Zatem $p(y|x; \theta) = L(\theta; X, y) = L(\theta)$ jest funkcją wiarygodności, natomiast poszukiwane θ to będą te argumenty, dla których funkcja wiarygodności osiąga maksimum, czyli wartości atrybutu decyzyjnego są najbardziej prawdopodobne przy zadanym zbiorze treningowym (McCullagh i Nelder, 1989). Z niezależności $\epsilon^{(j)}$ wynika następująca postać funkcji wiarygodności:

$$L(\theta) = \prod_{j=1}^{N} p(y^{(j)} \mid x^{(j)}; \theta) = \prod_{j=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y^{(j)} - \theta^T x^{(j)})^2}{2\sigma^2}}. \tag{12.9}$$

Funkcja $L(\theta)$ przyjmuje maksimum dla tych samych argumentów co funkcja $l(\theta) = \ln(L(\theta))$. Zatem można przyjąć, że $l(\theta)$ jest złożeniem funkcji logarytmicznej z wyjściową postacią powyższego równania. Wtedy stosując elementarne własności funkcji logarytmicznej, funkcję $l(\theta)$ można przekształcić do postaci:

$$\begin{aligned} l(\theta) = \ln(L(\theta)) &= \ln\left(\prod_{j=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y^{(j)} - \theta^T x^{(j)})^2}{2\sigma^2}} \right) = \\ &= \ln\left(\frac{N}{\sqrt{2\pi}\sigma} \prod_{j=1}^{N} e^{-\frac{(y^{(j)} - \theta^T x^{(j)})^2}{2\sigma^2}} \right) = \\ &= \ln\left(\frac{N}{\sqrt{2\pi}\sigma} \right) + \ln\left(\prod_{j=1}^{N} e^{-\frac{(y^{(j)} - \theta^T x^{(j)})^2}{2\sigma^2}} \right) = \\ &= \ln\left(\frac{N}{\sqrt{2\pi}\sigma} \right) + \sum_{j=1}^{N} \ln\left(e^{-\frac{(y^{(j)} - \theta^T x^{(j)})^2}{2\sigma^2}} \right) = \\ &= \ln\left(\frac{N}{\sqrt{2\pi}\sigma} \right) + \sum_{j=1}^{N} -\frac{(y^{(j)} - \theta^T x^{(j)})^2}{2\sigma^2} = \\ &= \ln\left(\frac{N}{\sqrt{2\pi}\sigma} \right) - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{j=1}^{N} (y^{(j)} - \theta^T x^{(j)})^2 \end{aligned} \tag{12.10}$$

Ponieważ $(y^{(j)} - \theta^T x^{(j)})^2 = (-\theta^T x^{(j)} + y^{(j)})^2 = (-(\theta^T x^{(j)} - y^{(j)}))^2 = (\theta^T x^{(j)} - y^{(j)})^2$, możemy zapisać, że:

$$l(\theta) = \ln \frac{N}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{j=1}^{N} (y^{(j)} - \theta^T x^{(j)})^2. \tag{12.11}$$

Z powyższego równania wynika, że aby zmaksymalizować $l(\theta)$, należy zminimalizować $\sum_{j=1}^{N}(\theta^T x^{(j)} - y^{(j)})^2/2$. Rozwiązywany problem znalezienia θ_i maksymalizujących prawdopodobieństwo $p(y|X;\theta)$ sprowadza się zatem do znalezienia θ_i, które minimalizują wyrażenie $\sum_{j=1}^{N}(\theta^T x^{(j)} - y^{(j)})^2/2$ zwane funkcją kosztu $J(\theta)$ (Johnson i Wichern, 2007). Jako że $\theta^T x^{(j)}$ jest wektorową postacią hipotezy $h_\theta(x^{(j)})$, wzór funkcji kosztu przyjmie następującą postać:

$$J(\theta) = \frac{1}{2}\sum_{j=1}^{N}(h_\theta(x^{(j)}) - y^{(j)})^2. \tag{12.12}$$

W celu znalezienia argumentu minimum funkcji kosztu $J(\theta)$ można posłużyć się różnymi metodami, np. metodą analityczną lub metodą gradientu prostego.

12.1. Metoda analityczna

Niech

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \cdot \\ \cdot \\ \cdot \\ y^{(N)} \end{bmatrix}, \quad X = \begin{bmatrix} 1 & x_1^{(1)} & \dots & x_d^{(1)} \\ 1 & x_1^{(2)} & \dots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(N)} & \dots & x_d^{(N)} \end{bmatrix} \text{ oraz } \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \cdot \\ \cdot \\ \cdot \\ \theta_d \end{bmatrix}.$$

Wtedy można zapisać wektorowo:

$$\begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(N)}) - y^{(N)} \end{bmatrix} = \begin{bmatrix} \theta^T x^{(1)} \\ \vdots \\ \theta^T x^{(N)} \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{bmatrix} = X\theta - \boldsymbol{y}. \tag{12.13}$$

Zatem funkcja kosztu przyjmuje postać:

$$J(\theta) = \frac{1}{2}\sum_{j=1}^{N}(h_\theta(x^{(j)}) - y^{(j)})^2 = \frac{1}{2}(X\theta - \boldsymbol{y})^T(X\theta - \boldsymbol{y}) = \frac{1}{2}(\theta^T X^T X\theta - \theta^T X^T \boldsymbol{y} - \boldsymbol{y}^T X\theta + \boldsymbol{y}^T \boldsymbol{y}) \tag{12.14}$$

Aby wyznaczyć argument minimum funkcji kosztu, należy znaleźć miejsce zerowe gradientu funkcji kosztu. Po zastosowaniu podstawowych własności macierzy i śladu macierzy (opisano w dodatku) gradient funkcji kosztu można zapisać następująco:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(\theta^T X^T X\theta - \theta^T X^T \boldsymbol{y} - \boldsymbol{y}^T X\theta + \boldsymbol{y}^T \boldsymbol{y}) = \\
&= \frac{1}{2}\nabla_\theta \mathrm{tr}(\theta^T X^T X\theta - \theta^T X^T \boldsymbol{y} - \boldsymbol{y}^T X\theta + \boldsymbol{y}^T \boldsymbol{y}) = \\
&= \frac{1}{2}\nabla_\theta \mathrm{tr}(\theta^T X^T X\theta) - \mathrm{tr}((\boldsymbol{y}^T X\theta)^T - \mathrm{tr}(\boldsymbol{y}^T X\theta) + \mathrm{tr}(\boldsymbol{y}^T \boldsymbol{y})) = \\
&= \frac{1}{2}\nabla_\theta \mathrm{tr}(\theta^T X^T X\theta) - 2\mathrm{tr}(\boldsymbol{y}^T X\theta) + \mathrm{tr}(\boldsymbol{y}^T \boldsymbol{y}) = \\
&= \frac{1}{2}[\nabla_\theta \mathrm{tr}(\theta^T X^T X\theta) - 2\nabla_\theta \mathrm{tr}(\boldsymbol{y}^T X\theta) + \nabla_\theta \mathrm{tr}(\boldsymbol{y}^T \boldsymbol{y})] = \\
&= \frac{1}{2}[2X^T X\theta - 2X^T \boldsymbol{y}] = \\
&= X^T X\theta - X^T \boldsymbol{y}
\end{aligned} \tag{12.15}$$

Zatem przy założeniu, że $X^T X$ jest nieosobliwa, poszukiwany wektor wartości θ_i, $i = 1, ..., d$ wyznaczany jest zgodnie ze wzorem:

$$J(\theta) = 0 \Leftrightarrow X^T X\theta - X^T y = 0 \Leftrightarrow X^T X\theta = X^T \boldsymbol{y} \Leftrightarrow \theta = (X^T X)^{-1} X^T \boldsymbol{y} \tag{12.16}$$

Równanie to wydaje się bardzo proste i można by uznać, że nie ma potrzeby stosowania innych metod. Jednak okazuje się, że w przypadku często stosowanych, bardzo dużych zbiorów treningowych, np. z milionem obiektów opisanych tysiącami atrybutów warunkowych, powyższa droga do rozwiązania przestaje być efektywna.

12.2. Regularyzacja Tichonowa

Zastosowanie metody analitycznej może skutkować uzyskaniem wektora θ zawierającego duże wartości bezwzględne. W praktyce oznacza to duże wagi modelu regresyjnego, które sprawiają, że wyjście modelu y jest wrażliwe na niewielkie zmiany wartości wejściowych x_i. Jest to zjawisko niekorzystne, mogące prowadzić do uzyskania dużego błędu generalizacji oraz przeuczenia modelu.

Kolejną dość problematyczną kwestią jest możliwość wystąpienia liniowej zależności zmiennych objaśniających. Ryzyko wystąpienia korelacji zmiennych jest szczególnie wysokie w sytuacji, kiedy liczba zmiennych objaśniających jest większa od liczby obserwacji. Macierz $X^T X$ może wówczas nie być odwracalna, co z kolei oznacza, że znalezienie jednoznacznego rozwiązania metodą analityczną nie będzie możliwe.

Jednym z najczęściej stosowanych rozwiązań powyższych problemów jest tzw. regularyzacja Tichonowa, zwana również regularyzacją L2. Polega ona na wprowadzeniu do funkcji kosztu dodatkowego parametru regularyzacyjnego $\alpha \geq 0$, który reprezentuje karę nakładaną na model, gdy wagi są zbyt duże (Goodfellow i in., 2018). Funkcja kosztu przyjmuje wówczas postać:

$$J(\theta) = \frac{1}{2}(X\theta - y)^T (X\theta - y) + \alpha\theta^T\theta. \tag{12.17}$$

Wektor θ wyznaczany jest ze wzoru:

$$\theta = (X^T X + \alpha I)^{-1} X^T y, \tag{12.18}$$

gdzie I oznacza macierz jednostkową o wymiarach $N \times N$.

Stopień uwzględnienia kary zależy od wartości parametru α:
- gdy $\alpha = 0$, to model będzie zwykłym modelem regresji liniowej,
- gdy $\alpha = \infty$, to wszystkie wagi będą dążyć do 0.

Wszystkie wartości parametru regularyzacji α dobierane są w wyniku optymalizacji ze sprawdzianem krzyżowym.

UWAGA

Parametr regularyzacji α bywa również oznaczany jako λ.

UWAGA

Regresja liniowa z regularyzacją Tichonowa nosi nazwę **regresji grzbietowej** (ang. *ridge regression*). Oprócz podstawowej wersji, w której wyznaczane współczynniki θ_i są wyrażone liczbami, istnieje również tzw. **bayesowska regresja grzbietowa** (ang. *bayesian ridge regression*). W jej przypadku zamiast konkretnych wartości liczbowych estymowane są rozkłady prawdopodobieństwa wartości współczynników θ_i.

12.3. Metoda gradientu prostego

Problem znalezienia wektora θ na podstawie gradientu funkcji kosztu można rozwiązać metodą gradientu prostego. Aby zrozumieć, na czym polega algorytm gradientu prostego, należy przypomnieć, że gradient funkcji wielu zmiennych to wektor, którego kierunek pokrywa się z kierunkiem, w którym funkcja zmienia się najszybciej, a jego zwrot wskazuje, w którą stronę funkcja rośnie. Zgodnie z wyprowadzonym wcześniej wzorem gradient funkcji kosztu można przedstawić w postaci:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{N}(h_\theta(x^{(i)}) - y^{(i)})^2. \tag{12.19}$$

Korzystając z obliczeń przedstawionych w metodzie analitycznej, gradient funkcji kosztu można zapisać wzorem:

$$\nabla J(\theta) = X^T X\theta - X^T \boldsymbol{y}. \tag{12.20}$$

Dla tak określonego gradientu funkcji kosztu można zastosować algorytm gradientu prostego w celu znalezienia poszukiwanego wektora θ, który będzie minimalizować funkcję kosztu. W metodzie tej wykonuje się kroki o ustalonej długości α w kierunku przeciwnym do gradientu funkcji celu w punkcie startowym danej iteracji.

Kroki algorytmu (metoda gradientu prostego)

1. Ustalenie kryterium stopu, np.:
   – maksymalna liczba iteracji t_{max},
   – gradient bliski zero $|\nabla J(\theta^{(i)})| < \epsilon$, gdzie ϵ jest z góry ustalone, np. $\epsilon = 10^{-3}$,
   – brak poprawy funkcji celu: $d(\theta^{(t+1)}, \theta^{(t)}) < \epsilon$, gdzie ϵ jest z góry ustalone.
2. Ustalenie wielkości kroku $\alpha^{(*)}$.
3. Przyjęcie startowej wartości parametrów $\theta^{(0)}$ (punkt startowy).
4. Ustalenie $i := 0$.
5. Obliczenie w punkcie startowym wartości funkcji kosztu $J_i = J(\theta^{(i)})$ i jej gradientu $g_i = g(\theta^{(i)}) = \nabla J(\theta^{(i)})$
6. Wyznaczenie kierunku poszukiwań: $d = -g_i^{(**)}$.
7. Wykonanie z punktu $\theta^{(i)}$ kroku o długości α w wyznaczonym kierunku d: $\theta^{(i+1)} := \theta^{(i)} + \alpha d$.
8. Obliczenie w nowym punkcie wartości funkcji kosztu $J_{i+1} = J(\theta^{(i+1)})$ i jej gradientu $g_{i+1} = \nabla J(\theta^{(i+1)})$.
9. Sprawdzenie, czy $J_{i+1} < J_i$:
   ◦ jeżeli tak, to dla $i := i + 1$ wykonać ponownie kroki 6–8,
   • jeżeli nie, to sprawdzić spełnienie kryterium stopu – gdy nie jest spełnione, należy zmniejszyć krok α i wrócić do kroku 7.

(*) Wybór α jest arbitralny. α to hiperparametr, którego wartość ma wpływ na efekty stosowanego algorytmu:
– jeżeli α jest zbyt małe, to algorytm będzie wykonywany bardzo długo ze względu na zbyt małą zbieżność,
– jeżeli α jest zbyt duże, to możliwe jest pominięcie minimum i funkcja zacznie rosnąć.

(**) Zwrot gradientu wskazuje na wzrost funkcji, a w rozwiązywanym zadaniu optymalizacyjnym poszukiwane jest minimum, czyli interesuje nas, gdzie maleją wartości funkcji. Dlatego należy zastosować znak przeciwny do znaku gradientu.

UWAGA

Szeroko stosowany w zadaniach optymalizacyjnych algorytm gradientowy, wykorzystywany w celu znalezienia ekstremum funkcji, jest wrażliwy na minima lokalne. Oznacza to, że z danego punktu w przestrzeni parametrów algorytm prowadzi do najbliższego minimum lokalnego. W przypadku regresji liniowej nie stanowi to jednak problemu, ponieważ jej funkcja kosztu ma tylko jedno minimum globalne.

Przykład 12.1. Regresja liniowa – metoda najmniejszych kwadratów

Pod adresem: https://www.kaggle.com/datasets/mirichoi0218/insurance dostępne są dane dotyczące wysokości kosztów procedur medycznych pokrytych przez firmę ubezpieczeniową.

W skład bazy danych wchodzi siedem zmiennych:

1) age – wiek osoby poddanej procedurze medycznej,
2) sex – płeć,
3) bmi – wskaźnik masy ciała,
4) children – liczba dzieci,
5) smoker – określenie, czy osoba pali papierosy („yes"), czy nie („no"),
6) region – jeden z czterech regionów geograficznych Stanów Zjednoczonych („northwest", „northeast", „southwest", „southeast"),
7) charges – wysokość kosztów pokrytych przez ubezpieczyciela.

Regresja liniowa może posłużyć do przewidywania wysokości kosztów, które ubezpieczyciel będzie musiał ponieść w przypadku osób opisanych danym zestawem parametrów. To może z kolei posłużyć do określenia ceny ubezpieczenia zdrowotnego, która zostanie zaproponowana danej osobie.

Przed przystąpieniem do przeprowadzenia regresji liniowej należy zastanowić się, czy wszystkie dostępne dane można poddać analizie. Pierwszą rzeczą, na którą należy zwrócić uwagę, jest typ danych – model regresji liniowej przyjmuje wyłącznie dane liczbowe, natomiast w analizowanych danych występują trzy zmienne kategorialne: określające płeć, fakt palenia/niepalenia oraz region. Konieczne jest więc zakodowanie ich za pomocą wartości liczbowych – w tym celu można wykorzystać funkcję OrdinalEncoder zaimplementowaną w bibliotece scikit-learn, która w sposób automatyczny przekoduje dane wyrażone słownie na dane numeryczne. Drugą kwestią do przemyślenia jest to, czy model dopasowany do wszystkich posiadanych danych będzie możliwy do interpretacji. Przedstawienie płci w postaci zmiennej dychotomicznej, przyjmującej wartość 0 lub 1 można uznać za logiczne – będzie to wówczas zmienna binarna określająca występowanie danej cechy (płci) u konkretnej osoby. Podobnie wygląda sytuacja ze zmienną określającą, czy osoba jest palaczem, czy nie. Natomiast przedstawienie regionu w postaci zmiennej przyjmującej cztery poziomy liczbowe (0, 1, 2, 3) jest już mniej logiczne – dlaczego 0 miałoby oznaczać np. region południowo-wschodni, a nie północno-wschodni? Przypisanie konkretnych wartości do regionów geograficznych musiałoby zostać przeprowadzone w sposób arbitralny, a przyjęcie innego systemu kodowania znacząco wpłynęłoby na uzyskiwany model regresyjny. Bezpieczniejszym wyborem jest więc pominięcie takich zmiennych i nieuwzględnianie ich podczas treningu i ewaluacji modelu.

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.metrics import (mean_absolute_error,
                             explained_variance_score,
                             max_error)
from sklearn.preprocessing import OrdinalEncoder

df = pd.read_csv('insurance.csv')
df.drop(columns = ['region'], inplace = True)

X_col_names = df.columns[:-1]
X = df[X_col_names].to_numpy()
y = df['charges'].to_numpy()
```

```
enc = OrdinalEncoder()
#kodowanie zmiennych nienumerycznych za pomocą wartości numerycznych
X[:,1] = enc.fit_transform(X[:,1].reshape(-1, 1)).squeeze()
X[:,4] = enc.fit_transform(X[:,4].reshape(-1, 1)).squeeze()
#reshape(-1, 1) - dodanie wymiaru jednostkowego konieczne, ponieważ
#funkcja OrdinalEncoder przyjmuje dane dwuwymiarowe, więc z wektora
#należy utworzyć macierz o wymiarach długość_wektora × 1
#squeeze() - usunięcie nadmiarowych wymiarów jednostkowych

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state = 42,
                                                    test_size = 0.2)
#jeżeli dane mają być użyte przez model regresyjny, nie należy
#przeprowadzać stratyfikacji (używać stratify = y)
```

W celu dopasowania do danych modelu regresji liniowej należy użyć klasy LinearRegression z biblioteki scikit-learn. Domyślnie zostanie dopasowany model liniowy, w którym wyraz wolny jest różny od zera. Jeżeli ma zostać dopasowany model z zerowym wyrazem wolnym, należy podczas inicjalizacji modelu podać do klasy argument *fit_intercept* = *False*.

Modele regresyjne często oceniane są za pomocą dwóch metryk:

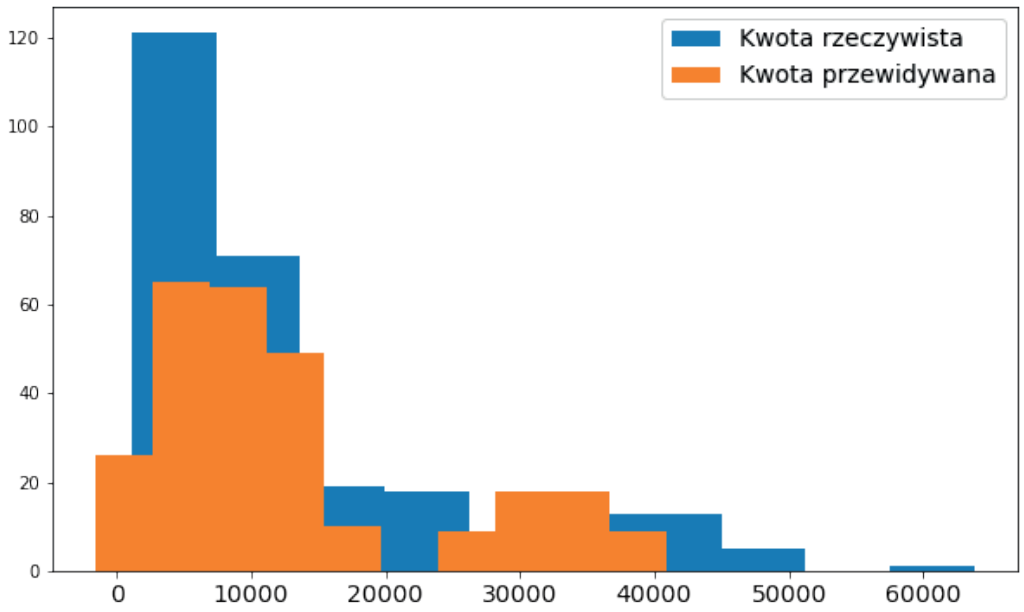
1) MAE (ang. *Mean Absolute Error*) – średni błąd bezwzględny określający, o jaką wartość średnio myli się model (równanie (5.3)),
2) MSE (ang. *Mean Squared Error*) – błąd średniokwadratowy (równanie (5.1)).

Możliwe jest również proste wyznaczenie maksymalnego popełnianego przez model błędu – w tym celu należy użyć funkcji max_error.

```
reg = LinearRegression()
reg.fit(X_train, y_train)
preds = reg.predict(X_test)
print(mean_absolute_error(y_test, preds))
print(max_error(y_test, preds))
```

Błędy popełniane przez model są stosunkowo duże – średni absolutny błąd predykcji (MAE) wynosi około 4213, natomiast maksymalny popełniany błąd aż około 22 820. W celu dokładniejszej analizy popełnianych błędów można wyświetlić histogramy kwot rzeczywistych oraz przewidywanych przez uzyskany model (rys. 12.1).

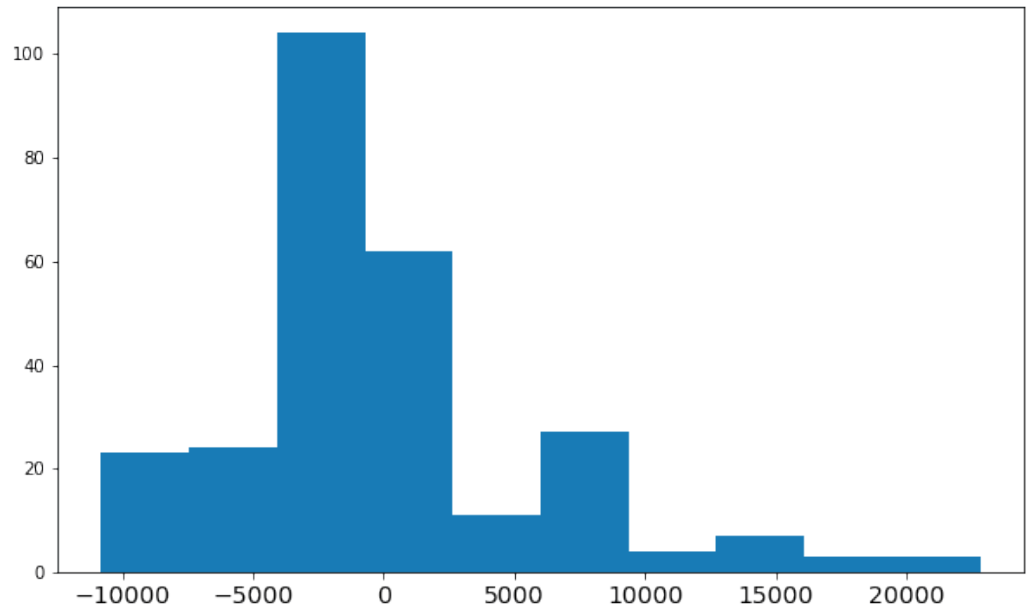
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 1, figsize = (10,6))
plt.xticks(fontsize = 14)
ax.hist(y_test, label = 'Kwota rzeczywista')
ax.hist(preds, label = 'Kwota przewidywana')
ax.legend(fontsize = 14)
plt.show()
```



Rysunek 12.1. Histogramy rzeczywistych kosztów procedur medycznych oraz kosztów przewidzianych przez model

Aby zobaczyć, jaki jest rozkład różnic pomiędzy kwotą rzeczywistą a przewidywaną, można również wyświetlić histogram różnic (rys. 12.2).

```
fig, ax = plt.subplots(1, 1, figsize = (10,6))
plt.xticks(fontsize = 14)
ax.hist(y_test-preds)
plt.show()
```



Rysunek 12.2. Histogram różnic pomiędzy rzeczywistymi kosztami procedur medycznych a kosztami przewidywanymi przez model

Analizując przedstawione histogramy, można zauważyć, że najczęściej popełniane błędy odpowiadają stosunkowo niewielkim kwotom i częściej dotyczą sytuacji, w której model zawyżył koszty procedur medycznych. Największe różnice pomiędzy kwotami rzeczywistymi a przewidywanymi występują natomiast w sytuacji, gdy model zaniżył koszty.

Przykład 12.2. Metoda stochastycznego zejścia gradientowego

Metoda gradientu prostego wiąże się z wysokimi kosztami obliczeniowymi oraz czasowymi, zwłaszcza w przypadku dużych zbiorów danych. Zamiast niej w praktyce często stosowana jest metoda tzw. stochastycznego zejścia gradientowego (ang. *Stochastic Gradient Descent*, SGD).

Metoda SGD różni się od metody gradientu prostego tym, że nie korzysta się w niej z gradientu wyliczonego na całym zbiorze danych uczących – zamiast tego wartość gradientu jest estymowana i aktualizowana na podstawie losowo wybranego jednego elementu ze zbioru danych (Ketkar, 2017). W bibliotece scikit-learn metoda SGD służąca do estymacji współczynników regresji liniowej zaimplementowana jest w klasie SGDRegressor i znajduje się pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html.

UWAGA

Istnieje również implementacja metody stochastycznego zejścia gradientowego służąca do prowadzenia klasyfikacji – wówczas należy użyć klasy SGDClassifier dostępnej pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html.

```
from sklearn.linear_model import SGDRegressor

reg = SGDRegressor()
reg.fit(X_train, y_train)
preds = reg.predict(X_test)
mean_absolute_error(y_test, preds)
```

Powyższy kod inicjalizuje model SGDRegressor z domyślnymi wartościami hiperparametrów, wśród których wyróżnić można m.in.:

- loss – funkcję kosztu,
- penalty – rodzaj regularyzacji,
- alpha – wartość parametru regularyzacji,
- max_iter – maksymalną liczbę przejść zbioru danych uczących przez model podczas jego treningu,
- learning_rate – parametr szybkości uczenia oznaczający wielkość kroku wykonywanego podczas poszukiwania minimum lokalnego funkcji kosztu.

Pełną listę hiperparametrów można znaleźć w dokumentacji klasy SGDClassifier.

Optymalizację hiperparametrów modelu można przeprowadzić np. przy użyciu pakietu Optuna – poniżej zamieszczono przykładowy kod realizujący optymalizację wybranych hiperparametrów.

```
import optuna
from sklearn.model_selection import cross_validate, Kfold
from sklearn.metrics import mean_absolute_error, make_scorer
```

```
#metryka, która posłuży do określenia funkcji celu
scoring = {'mae': make_scorer(mean_absolute_error)}
#model, który będzie optymalizowany
model = SGDRegressor
#definicja przestrzeni hiperparametrów:
#hiperparametry, które zostaną poddane optymalizacji oraz przedziały
#wartości, w których będą poszukiwane wartości optymalne
def get_space(trial):
  space = {"alpha": trial.suggest_uniform("alpha", 0, 1),
           "loss": trial.suggest_categorical("loss",
                                    ['squared_loss', 'huber',
                                     'epsilon_insensitive',
                                     'squared_epsilon_insensitive']),
           "penalty": trial.suggest_categorical("penalty",
                                    ['l1', 'l2', 'elasticnet'])}
  return space
trials = 500 #liczba prób
#definicja funkcji celu
def objective(trial, model, get_space, X, y):
  model_space = get_space(trial)
  mdl = model(**model_space)
  #walidacja będzie przeprowadzana z użyciem 5-krotnego sprawdzianu
  #krzyżowego
  scores = cross_validate(mdl, X, y, scoring = scoring,
                                    cv = KFold(n_splits = 5),
                                    return_train_score = True)
  return np.mean(scores['test_mae'])
#inicjalizacja optymalizacji
study = optuna.create_study(direction = 'minimize')
#przeprowadzenie optymalizacji
study.optimize(lambda x: objective(x, model, get_space,
                                          X_train,
                                          y_train),
               n_trials = trials)
#wyświetlenie dobranych wartości hiperparametrów
print('params: ', study.best_params)
#trening modelu o optymalnych hiperparametrach
sgd = model(**study.best_params)
sgd.fit(X_train, y_train)
#predykcja na zbiorze testowym
preds = sgd.predict(X_test)
#zapisanie nauczonego modelu do pliku:
#pickle.dump(sgd, open('sgd_model', 'wb+'))
#wyświetlenie błędu absolutnego uzyskanego na zbiorze testowym
print('test MAE: ', mean_absolute_error(y_test, preds))
```

Przykład 12.3. Regresja grzbietowa

Regresja grzbietowa może zostać przeprowadzona przy użyciu klasy Ridge zaimplementowanej w bibliotece scikit-learn. Inicjalizację i trening modelu o hiperparametrach przyjmujących wartości domyślne prowadzi się z wykorzystaniem poniższego kodu.

```
from sklearn.linear_model import Ridge
reg = Ridge()
reg.fit(X_train, y_train)
preds = reg.predict(X_test)
mean_absolute_error(y_test, preds)
```

W przypadku regresji grzbietowej najczęściej optymalizacji poddaje się trzy hiperparametry:

1) alpha – wartość parametru regularyzacji,
2) solver – metoda wykorzystywana do znalezienia współczynników regresji,
3) max_iter – maksymalna liczba iteracji, po której metoda sprzężonego gradientu wykorzystywana do poszukiwania rozwiązania powinna osiągnąć zbieżność, jeżeli zbieżność nie zostanie osiągnięta, procedura zostanie przerwana z najlepszym uzyskanym dotychczas wynikiem.

Pełna lista hiperparametrów modelu znajduje się w dokumentacji klasy Ridge pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html.

```
import optuna
from sklearn.model_selection import cross_validate, KFold
from sklearn.metrics import mean_absolute_error, make_scorer

#metryka, która posłuży do określenia funkcji celu
scoring = {'mae': make_scorer(mean_absolute_error)}

#model, który będzie optymalizowany
model = Ridge

#definicja przestrzeni hiperparametrów:
#hiperparametry, które zostaną poddane optymalizacji, oraz przedziały
#wartości, w których będą poszukiwane wartości optymalne
def get_space(trial):
  space = {"alpha": trial.suggest_uniform("alpha", 0, 1),
           "solver": trial.suggest_categorical("solver", ['svd',
                                                          'cholesky',
                                                          'sparse_cg'])}
  return space

trials = 500 #liczba prób

#definicja funkcji celu
def objective(trial, model, get_space, X, y):
  model_space = get_space(trial)
  mdl = model(**model_space)
```

```
    #walidacja będzie przeprowadzana z użyciem 5-krotnego sprawdzianu krzyżowego
    scores = cross_validate(mdl, X, y, scoring = scoring,
                cv = KFold(n_splits = 5),
                return_train_score = True)

    return np.mean(scores['test_mae'])

#inicjalizacja optymalizacji
study = optuna.create_study(direction = 'minimize')
#przeprowadzenie optymalizacji
study.optimize(lambda x: objective(x, model, get_space,
                                   X_train,
                                   y_train),
                n_trials = trials)
#wyświetlenie dobranych wartości hiperparametrów
print('params: ', study.best_params)

#trening modelu o optymalnych hiperparametrach
ridge = model(**study.best_params)
ridge.fit(X_train, y_train)
#predykcja na zbiorze testowym
preds = ridge.predict(X_test)
#zapisanie nauczonego modelu do pliku:
#pickle.dump(ridge, open('ridge_model', 'wb+'))
#wyświetlenie błędu absolutnego uzyskanego na zbiorze testowym
print('test MAE: ', mean_absolute_error(y_test, preds))
```

Bayesowską regresję grzbietową wykonuje się przy użyciu klasy BayesianRidge dostępnej pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.BayesianRidge.html.

Zadanie 12.1. Przewidywanie cen mieszkań przy użyciu regresji liniowej

1. Pod adresem: https://www.kaggle.com/quantbruce/real-estate-price-prediction znajdują się dane dotyczące cen mieszkań w Singapurze. Pobierz je i wczytaj do struktury DataFrame. Uzyskana tabela będzie zawierać osiem kolumn:
   1) No – numer porządkowy,
   2) X1 transaction date – data kupna,
   3) X2 house age – wiek budynku w latach,
   4) X3 distance to the nearest MRT station – odległość od najbliższej stacji metra,
   5) X4 number of convenience stores – liczba sklepów znajdujących się w okolicy,
   6) X5 latitude – szerokość geograficzna,
   7) X6 longitude – długość geograficzna,
   8) Y house price of unit area – cena za jednostkę powierzchni.
2. Usuń zbędne kolumny z tabeli, a następnie podziel dane na zbiór uczący i testowy w stosunku 0,8 : 0,2.
3. Przeprowadź trening modelu regresyjnego (regresji liniowej) oraz jego ewaluację. O jaką kwotę średnio myli się uzyskany model? Ile wynosi największy popełniany błąd?
4. Zastanów się, czy można przeprowadzić optymalizację modelu liniowego inicjalizowanego funkcją LinearRegression. Uzasadnij swoją odpowiedź.

Zadanie 12.2. Wyznaczanie częstotliwości tonu podstawowego na podstawie parametrów sygnału mowy przy użyciu regresji grzbietowej

Pod adresem: https://osf.io/cwquj/ znajduje się korpus mowy zawierający nagrania samogłosek o przedłużonej fonacji emitowanych z różną wysokością głosu.

1. Pobierz nagrania i przygotuj dane:
   – wygeneruj wektor zawierający częstotliwości tonu podstawowego wyznaczone na podstawie nagrań,
   – wygeneruj macierz parametrów akustycznych opisujących sygnał mowy zawarty w nagraniach, w tym celu użyj zestawu cech ComParE_2016 biblioteki openSMILE oraz zdefiniuj poziom ekstrahowanych cech jako Functionals, w ten sposób wyznaczone zostaną parametry statystyczne cech sygnału,
   – podziel dane na zbiór uczący i testowy,
   – ustandaryzuj dane,
   – przeprowadź redukcję wymiarowości, zmniejszając liczbę cech opisujących jeden obiekt do 100, w tym celu użyj funkcji SelectKBest, która pozwala wybrać określoną liczbę cech mających największy wpływ na działanie modelu: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html.
   Podpowiedź: przykładowa ekstrakcja cech przy użyciu biblioteki openSMILE została przedstawiona w rozdziale 7.
2. Przeprowadź trening i optymalizację hiperparametrów modelu regresji grzbietowej, który pozwoli przewidzieć wartość częstotliwości tonu podstawowego na podstawie parametrów statystycznych sygnałów. Jako metrykę stosowaną do ewaluacji modelu podczas optymalizacji wybierz:
   – błąd średniokwadratowy (MSE),
   – średni błąd bezwzględny (MAE).
3. Przeprowadź ewaluację uzyskanych modeli. Czy metryka zastosowana podczas optymalizacji ma wpływ na uzyskane wyniki?
4. Spróbuj zmodyfikować funkcję celu (*objective*) oraz przestrzeń hiperparametrów (*get_space*) w taki sposób, by optymalizacji poddać również liczbę cech wybieranych przez funkcję *SelectKBest*.
   Podpowiedź: potrzebna będzie funkcja *make_pipeline* dostępna pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.make_pipeline.html.

13 Uogólnione modele liniowe

(ang. *Generalized Linear Models,* GML*)*

Zastosowanie: regresja lub klasyfikacja.
Metoda: Uczenie nadzorowane.

Zadania rozwiązywane przy użyciu uogólnionych modeli liniowych można określić ogólnie jako oszacowanie wartości y na podstawie ustalonych cech x. Mamy tu do czynienia z uczeniem nadzorowanym, przy czym niektóre z modeli służą klasyfikacji, natomiast inne są algorytmami regresji (McCulloch i Searle, 2000).

Aby zrozumieć definicję uogólnionych modeli liniowych, należy najpierw przytoczyć definicję rodziny wykładniczej rozkładów prawdopodobieństwa.

DEFINICJA (rodzina wykładnicza)

Rodzina rozkładów prawdopodobieństwa $\{P_\theta;\theta \in \Theta\}$ jest rodziną wykładniczą, gdy rozkłady P_θ mają względem pewnej miary na przestrzeni obiektów X funkcje prawdopodobieństwa lub funkcję gęstości prawdopodobieństwa postaci:

$$p_\theta(y) = \frac{h(y)\cdot e^{\sum_{l=1}^{L}\eta_l(\theta)T_l(y)}}{e^{\kappa(\theta)}} = z(y)\cdot e^{(\eta^T T(y)-\kappa(\theta))}, \tag{13.1}$$

gdzie:

- η – parametr naturalny/kanoniczny dystrybucji,
- $T(y)$ – statystyka wystarczająca,
- $e^{\kappa(\theta)}$ – czynnik normalizujący,
- L – liczba parametrów, przy czym gdy $L = 1$, to:

$$p_\theta(y) = \frac{z(y)\cdot e^{\eta(\theta)T(y)}}{e^{\kappa(\theta)}} = z(y)\cdot e^{(\eta(\theta)T(y)-\kappa(\theta))}. \tag{13.2}$$

Uogólnione modele liniowe zdefiniowane są zestawem trzech warunków. Założenia modelu, który spełnia warunki uogólnionego modelu liniowego, są następujące (Nelder i Wedderburn, 1972).

Założenie nr 1. Zmienna y przy zadanych x i θ, tj. $(y \mid x; \theta)$, podlega rozkładowi wykładniczemu o parametrze η.

Założenie nr 2. Celem jest estymacja wartości oczekiwanej $T(y)$ przy danym x.

Założenie nr 3. Parametr naturalny η jest związany z wejściem x liniowo, czyli $\eta = \theta^T x$.

Analizując powyższe założenia, możemy wykazać, że regresja liniowa jest przykładem uogólnionego modelu liniowego.

13.1. Regresja liniowa jako uogólniony model liniowy

Wiadomo, że w regresji liniowej y jest zmienną ciągłą oraz przy zadanych atrybutach warunkowych x i parametrach θ (zdefiniowane w rozdziale 12) ma rozkład normalny, czyli możemy zapisać:

$$(y \mid x; \theta) \sim N(\mu, \sigma). \tag{13.3}$$

Można sprawdzić, że z założeń uogólnionych modeli liniowych wynika, że model regresji liniowej jest przykładem uogólnionych modeli liniowych.

Ad założenie nr 1
Należy sprawdzić, czy zmienna y przy zadanych x i θ podlega rozkładowi wykładniczemu. Wiadomo, że zmienna y przy zadanych x i θ ma rozkład Gaussa. Dla uproszczenia można przyjąć $\sigma = 1$, ponieważ w estymacji parametrów regresji liniowej nie używa się wariancji. Zatem:

$$p(y;\mu) = \frac{1}{\sqrt{2\pi}} e^{\frac{-(y-\mu)^2}{2}} = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(y^2-2y\mu+\mu^2)} = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}y^2} e^{y\mu-\frac{1}{2}\mu^2}. \tag{13.4}$$

Następnie możemy określić:

$$z(y) = \frac{1}{\sqrt{2\pi}} e^{\frac{-1}{2}y^2} \tag{13.5}$$

$$\eta(\mu) = \mu \tag{13.6}$$

$$T(y) = y \tag{13.7}$$

$$\kappa(\mu) = \frac{1}{2}\mu^2 \tag{13.8}$$

Oznacza to, że rozkład Gaussa należy do rodziny rozkładów wykładniczych i spełnione jest założenie nr 1.

Ad założenie nr 2
Zgodnie z drugim założeniem uogólnionych modeli liniowych w regresji liniowej powinno się estymować wartość oczekiwaną $T(y)$, gdy dany jest x. Z założenia nr 1 wiadomo, że $T(y) = y$, czyli:

$$h_\theta(x) = E(y|x;\theta) = \mu. \tag{13.9}$$

Ponadto z złożenia nr 1 wiadomo też, że $\eta(\mu) = \mu$, czyli:

$$h_\theta(x) = E(y|x;\theta) = \mu = \eta. \tag{13.10}$$

Ad założenie nr 3
Trzecie założenie determinuje postać parametru naturalnego dystrybucji $\eta = \theta^T x$. Po wstawieniu do równania uzyskanego we wcześniejszym założeniu otrzymujemy:

$$h_\theta(x) = \theta^T x\,, \tag{13.11}$$

zgodnie z tym, jak przyjęliśmy w rozdziale 12.

Tym samym wykazaliśmy, że regresja liniowa jest przykładem uogólnionych modeli liniowych (McCullagh i Nelder, 1989).

14 Regresja logistyczna

Zastosowanie: klasyfikacja binarna.
Metoda: uczenie nadzorowane.

Regresja logistyczna, pomimo nazwy sugerującej rozwiązywanie problemów regresji, jest w rzeczywistości klasyfikatorem binarnym. Tym samym zmienna zależna (atrybut decyzyjny) y może przyjąć tylko dwie wartości, tj. $y \in Y = \{0, 1\}$, gdzie Y to przestrzeń, z której pochodzą dane wyjściowe. Zatem $(y|x; \theta)$ ma rozkład Bernoulliego $B(\phi)$, gdzie $\phi = P(y = 1)$.

Zatem:

$$p(y = 1, \phi) = \phi, \tag{14.1}$$

$$p(y = 0, \phi) = 1 - \phi. \tag{14.2}$$

Regresja logistyczna podobnie jak regresja liniowa może być rozpatrywana jako szczególny przypadek uogólnionych modeli liniowych. Przeanalizujmy założenia uogólnionych modeli liniowych (opisane w rozdziale 13) w kontekście wprowadzonych wstępnie informacji dotyczących regresji logistycznej (McCullagh i Nelder, 1989).

Ad założenie nr 1
Wiadomo, że zmienna y ma rozkład Bernoulliego. Pozostaje wykazać, że rozkład ten należy do rodziny rozkładów wykładniczych. Można tego dokonać przez zastosowanie podstawowych własności funkcji logarytmicznej.

$$\begin{aligned} p(y; \phi) &= \phi^y (1 - \phi)^{(1-y)} = \\ &= e^{\log(\phi^y (1-\phi)^{1-y})} = \\ &= e^{\log(\phi^y) + \log((1-\phi)^{1-y})} = \\ &= e^{y \cdot \log(\phi) + (1-y) \cdot \log(1-\phi)} = \\ &= e^{y \cdot \log(\phi) + \log((1-\phi) - y \cdot \log(1-\phi))} = \\ &= e^{y(\log(\phi) - \log(1-\phi) + \log(1-\phi))} = \\ &= e^{y \cdot \log\left(\frac{\phi}{1-\phi}\right) + \log(1-\phi)} \end{aligned} \tag{14.3}$$

Wtedy:

$$z(y) = 1, \tag{14.4}$$

$$T(y) = y, \tag{14.5}$$

$$\eta(\phi) = \log\left(\frac{\phi}{1-\phi}\right), \tag{14.6}$$

$$\kappa(\phi) = -\log(1 - \phi). \tag{14.7}$$

Zatem rozkład Bernoulliego może być rozpatrywany jako rozkład wykładniczy.

Ad założenie nr 2
W rozpatrywanym modelu powinno się estymować wartość oczekiwaną $T(y)$ przy znanym x. Z założenia nr 1 wiadomo, że $T(y) = y$, czyli:

$$h_\theta(x) = E(y|x; \theta) = \phi. \tag{14.8}$$

Z założenia nr 1 wiadomo też, że $\eta(\phi) = \log(\phi/(1 - \phi))$, czyli $\phi = 1/(1 + e^{-\eta})$. Podstawiając ϕ do powyższego wzoru, otrzymujemy:

$$h_\theta(x) = E(y \mid x; \theta) = \phi = \frac{1}{1+e^{-\eta}}. \tag{14.9}$$

Ad założenie nr 3
Trzecie założenie determinuje postać parametru naturalnego dystrybucji $\eta = \theta^T x$, stąd:

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}. \tag{14.10}$$

Jak widać, dzięki założeniom uogólnionych modeli liniowych uzyskaliśmy postać funkcji $h_\theta(x)$. W przypadku regresji liniowej znaliśmy tę funkcję z góry i sprawdzenie, czy można wyprowadzić jej postać również z uogólnionych modeli liniowych, było jedynie formalnością. Jednak nie zawsze postać funkcji $h_\theta(x)$ jest oczywista i w takich właśnie przypadkach jest przydatna znajomość uogólnionych modeli liniowych, o ile oczywiście rozkład zmiennej y można przedstawić w postaci rodziny wykładniczej.

Znając funkcję $h_\theta(x)$ dla regresji logistycznej, do dokończenia zadania, czyli otrzymania modelu dokonującego klasyfikacji, należy – podobnie jak w przypadku regresji liniowej – znaleźć najlepsze θ spełniające równanie (14.10). Skoro wiemy, że $(y|x; \theta) \sim B(\phi)$, a z założenia nr 2 wynika, że $h_\theta(x) = \phi$, to możemy zapisać:

$$P(y = 1 \mid x; \theta) = h_\theta(x) \tag{14.11}$$

oraz

$$P(y = 0 \mid x; \theta) = 1 - h_\theta(x). \tag{14.12}$$

Ostatecznie funkcja będzie dana wzorem:

$$P(y|x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y}. \tag{14.13}$$

Niech $(x^{(j)}, y^{(j)})_{j = 1, \ldots, N}$ są niezależnymi obiektami w zbiorze treningowym. Wtedy funkcja wiarygodności przyjmuje postać:

$$L(\theta) = P(y \mid X; \theta) = \prod_{j=1}^{N} P(y^{(j)} \mid x^{(j)}; \theta) = \prod_{j=1}^{N} (h_\theta(x^{(j)}))^{y^{(j)}} (1 - h_\theta(x^{(j)}))^{1-y^{(j)}}, \tag{14.14}$$

gdzie $h_\theta(x) = 1/(1 + e^{-\theta^T x})$ (Johnson i Wichern, 2007).

Następnie, podobnie jak w rozdziale 12, wyznaczamy tzw. funkcję log-wiarygodności $l(\theta)$ daną wzorem:

$$l(\theta) = \ln(L(\theta)) = \sum_{j=1}^{N} (y^{(j)} \ln(h_\theta(x^{(j)})) + (1 - y^{(j)}) \ln(1 - h_\theta(x^{(j)}))). \tag{14.15}$$

Aby znaleźć rozwiązanie problemu metodą analityczną, należy rozwiązać ze względu na θ równanie, w którym gradient funkcji log-wiarygodności zeruje się, czyli:

$$\nabla l(\theta) = 0. \tag{14.16}$$

Poszukiwane wartości parametrów θ są jednak zwykle wyznaczane numerycznie iteracyjną metodą najmniejszych kwadratów z doregulowaniem wag (Mardia i in., 1979).

14.1. Ocena jakości modelu regresji logistycznej

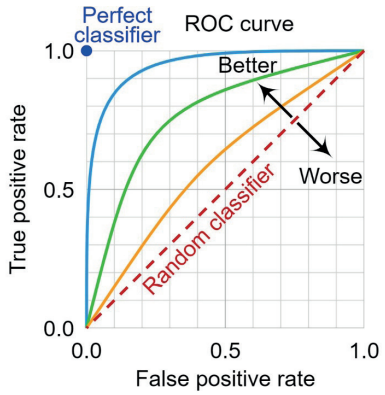
W rozdziale 5 opisano standardowe metody oceny poszczególnych typów modeli, w szczególności klasyfikatorów. Jednak regresja logistyczna należy do szczególnej grupy klasyfikatorów, gdzie na wyjściu wytrenowanego modelu zamiast wybranej klasy otrzymujemy prawdopodobieństwo przynależności obiektu do klasy pozytywnej ($y = 1$). Implikuje to konieczność zastosowania szczególnych metod oceny wytrenowanego modelu, bazujących na prawdopodobieństwie skorygowanym.

Przykładem takiego wskaźnika jest tzw. **strata logarytmiczna** (ang. *log loss*), zwana również binarną entropią krzyżową. Parametr ten jest ujemną średnią logarytmiczną skorygowanych prawdopodobieństw, którą można wyprowadzić z równania funkcji log-wiarygodności regresji logistycznej. Otrzymamy wówczas:

$$logloss = -\frac{1}{N}\sum_{j=1}^{N}(y^{(j)}\ln(h_\theta(x^{(j)})) + (1-y^{(j)})\ln(1-h_\theta(x^{(j)}))). \tag{14.17}$$

Stosuje się tutaj ujemną średnią, ponieważ prawdopodobieństwo przyjmuje wartości z przedziału [0, 1] i jego logarytm ma wartość ujemną. Natomiast dzięki przemnożeniu średniej z logarytmów przez wartość –1 otrzymuje się dodatnią stratę logarytmiczną, co upraszcza interpretację, gdyż im mniejsza będzie strata logarytmiczna, tym predykcje modelu będą bliższe wartości rzeczywistej.

Oprócz straty logarytmicznej do oceny jakości klasyfikatora zwracającego prawdopodobieństwa przynależności do klas używa się także metryki **ROC AUC**, zdefiniowanej jako pole pod krzywą ROC. Na początek należy jednak zrozumieć, czym jest krzywa ROC (ang. *Receiver Operating Characteristic*) i jak wygląda. Krzywa ROC obrazuje zdolności diagnostyczne modelu w zależności od przyjętego progu dyskryminacyjnego. Wykres przedstawia wartości TPR (czułość) w zależności od wartości FPR = 1 – TNR (1 – specyficzność). Idealny model to taki, w którym krzywa ROC dla FPR = 0 przyjmuje wartość TPR = 1, a następnie wykres przebiega równolegle do osi *X*. Im bardziej przebieg uzyskanej krzywej różni się od takiego przebiegu, tym gorszy model. Przykładowe krzywe ROC przedstawiono na rysunku 14.1.



Rysunek 14.1. Przykładowe krzywe ROC dla różnych klasyfikatorów (MartinThoma, 2022)

Wskaźnik ROC_AUC co do wartości jest równy polu pod wykresem krzywej ROC. Zatem w idealnym przypadku ROC_AUC wynosi 1. Jeżeli wskaźnik ten wynosi 0,5, to utworzony model prawdopodobnie dokonuje predykcji w sposób losowy (prawdopodobieństwo przynależności do klasy „0" wynosi 0,5 i do klasy „1" również 0,5). Taki model nie nadaje się do użytku. Natomiast uzyskanie wskaźnika ROC_AUC = 1 jest w przypadku danych rzeczywistych bardzo trudne i nie zawsze jest możliwe. Dążymy jednak do tego, by jego wartość była jak największa.

Krzywą ROC otrzymaną dla konkretnych danych testowych można znaleźć w Przykładzie 14.1.

UWAGA

Każdy model zwracający losowe predykcje ma ROC_AUC = 0,5, natomiast jest możliwe, że model o ROC_AUC = 0,5 nie zwraca losowych predykcji.

Przykład 14.1. Przewidywanie prawdopodobieństwa przeżycia katastrofy na przykładzie pasażerów Titanica

Pod adresem: https://www.kaggle.com/datasets/azeembootwala/titanic znajdują się dane dotyczące pasażerów Titanica, które posłużą do zademonstrowania działania algorytmu regresji logistycznej. Dane są już w odpowiedni sposób przetworzone tak, by mogły zostać użyte do przeprowadzenia treningu i ewaluacji modelu – dane poddano standaryzacji, dane brakujące zostały usunięte lub uzupełnione, a zmienne kategorialne zostały zakodowane za pomocą zmiennych numerycznych przyjmujących dwie wartości: 0 i 1.

Na początek dane zostaną wczytane do dwóch struktur DataFrame – osobnych dla zbioru uczącego i testowego. Po ich wyświetleniu można zauważyć, że w tabeli znajduje się 15 kolumn:

1) Unnamed: 0 – kolumna bez nadanej nazwy (nazwa utworzyła się automatycznie podczas wczytywania danych do struktury DataFrame), w której znajduje się numer porządkowy wiersza,
2) PassengerId – numer identyfikacyjny pasażera,
3) Survived – zmienna określająca, czy dany pasażer przeżył katastrofę (1), czy nie (0),
4) Sex – płeć pasażera (0 – kobieta, 1 – mężczyzna),
5) Age – wiek znormalizowany do przedziału (0, 1),
6) Fare – wysokość opłaty za bilet,
7) Pclass_1 – zmienna określająca, czy pasażer podróżował w klasie 1,
8) Pclass_2 – zmienna określająca, czy pasażer podróżował w klasie 2,
9) Pclass_3 – zmienna określająca, czy pasażer podróżował w klasie 3,
10) Family_size – liczba członków rodziny znajdujących się na pokładzie statku, ustandaryzowana do przedziału (0, 1),
11) Title_1 – zmienna określająca, czy pasażer był żonaty,
12) Title_2 – zmienna określająca, czy pasażerka była mężatką,
13) Title_3 – zmienna określająca, czy pasażer był kawalerem,
14) Title_4 – zmienna określająca, czy pasażerka była panną,
15) Emb_1, Emb_2, Emb_3 – zmienne binarne określające port (Cherbourg, Queenstown lub Southampton), w którym pasażer wsiadł na pokład.

Analiza dwóch pierwszych zmiennych (liczba porządkowa i numer identyfikacyjny pasażera) nie miałaby sensu, ponieważ nie są to zmienne określające cechy pasażera, dlatego zostaną one usunięte z danych przed przystąpieniem do uczenia modelu.

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (log_loss,
                             roc_auc_score,
                             confusion_matrix,
                             classification_report,
                             make_scorer)
import matplotlib.pyplot as plt
from sklearn.model_selection import (cross_validate, StratifiedKFold)
import optuna

train_df = pd.read_csv('train_data.csv')
test_df = pd.read_csv('test_data.csv')

train_df.drop(columns = ['Unnamed: 0', 'PassengerId'], inplace = True)
test_df.drop(columns = ['Unnamed: 0', 'PassengerId'], inplace = True)
```

```
columns = train_df.columns
X_train = train_df[columns[1:]].to_numpy()
X_test = test_df[columns[1:]].to_numpy()
y_train = train_df['Survived'].to_numpy()
y_test = test_df['Survived'].to_numpy()
```

Na początek model regresji logistycznej zostanie zainicjalizowany z wartościami domyślnymi hiperparametrów. Po przeprowadzeniu treningu zostanie on poddany ewaluacji – w wyniku zastosowania metody predict uzyskuje się przynależność obiektów do klas, możliwe jest więc wyznaczenie macierzy pomyłek oraz obliczenie wartości metryk sukcesu.

```
lr = LogisticRegression()
lr.fit(X_train, y_train)
preds = lr.predict(X_test)
print(confusion_matrix(y_test, preds))
print(classification_report(y_test, preds))
```

Aby uzyskać prawdopodobieństwo przynależności do klas, a nie bezpośrednio predykcję samej klasy, należy użyć metody predict_proba. Prawdopodobieństwo przynależności obiektów do klasy 1. będzie znajdować się w kolumnie 1. uzyskanej macierzy predykcji.

UWAGA

Regresja logistyczna w bibliotece scikit-learn jest klasyfikatorem, a nie regresorem, dlatego po użyciu predict_proba dla każdego obiektu uzyskuje się wektor dwóch wartości: prawdopodobieństwo przynależności do klasy 0 oraz do klasy 1. Tak samo jest w przypadku innych modeli klasyfikacyjnych, na których użyje się metody predict_proba. W przypadku modeli regresyjnych otrzymuje się tylko jedną wartość, ponieważ regresory mają tylko jedno wyjście.

```
preds = lr.predict_proba(X_test)
print(roc_auc_score(y_test, preds[:,1]))
print(log_loss(y_test, preds[:,1]))
```

Wartość pola pod krzywą ROC (ROC_AUC) uzyskana na zbiorze testowym wynosi około 0,89, zaś strata logarytmiczna wynosi 0,3728477052627943. Można to uznać za całkiem dobre rezultaty – ROC_AUC jest znacząco większa od 0,5. Strata logarytmiczna jest stosunkowo duża, co świadczy o tym, że zdolności generalizacyjne modelu nie są bardzo dobre. Jednak biorąc pod uwagę, że nie przeprowadzono optymalizacji hiperparametrów modelu, nie można powiedzieć, że jest to zły wynik.

Dla porównania wartość straty logarytmicznej zostanie obliczona ręcznie, zgodnie ze wzorem (14.17). Należy zauważyć, że prawdopodobieństwo skorygowane oblicza się zgodnie ze schematem:

1. jeżeli obiekt należy do klasy 1, to prawdopodobieństwo skorygowane wynosi tyle, ile wyznaczone przez regresor (y_pred),
2. jeżeli obiekt należy do klasy 0, to prawdopodobieństwo skorygowane wynosi 1 – (y_pred).

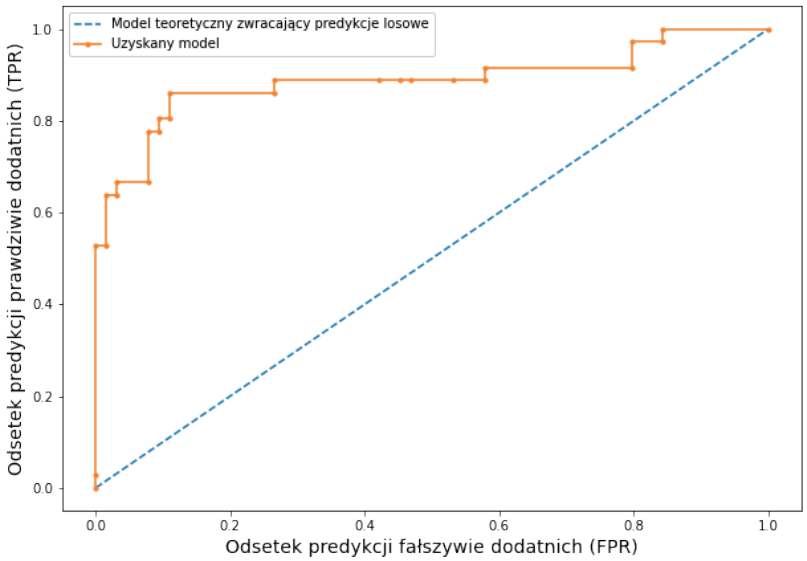
```
corrected_prob = []
for y_pred, y_true in zip(preds[:,1], y_test):
```

```
    if y_true == 0:
      corrected_prob.append(1-y_pred)
    else:
      corrected_prob.append(y_pred)

corrected_prob = np.array(corrected_prob)
log_prob = np.log(corrected_prob + 1e-15)
logloss = -(np.mean(log_prob))
print(logloss)
```

W wyniku obliczeń uzyskano wartość straty logarytmicznej wynoszącą 0,3728477052627926, czyli niemal identyczną jak wyznaczona wcześniej. Drobne różnice pomiędzy tą wartością oraz wartością uzyskaną przy użyciu funkcji log_loss z biblioteki scikit-learn wynikają z przybliżeń numerycznych. W praktyce różnice te nie mają znaczenia, dlatego wyznaczone wartości metryk podaje się z mniejszą dokładnością (zazwyczaj do dwóch lub czterech miejsc po przecinku).

Krzywa ROC, na podstawie której wyznaczono wartość metryki ROC_AUC, może zostać narysowana za pomocą poniższej funkcji. Niebieską linią przerywaną zostanie zaznaczony model zwracający predykcje losowe (rys. 14.2).

```
def plot_roc_curve(y, preds, image_path = None):
  import matplotlib.pyplot as plt
  from sklearn.metrics import roc_curve

  fpr, tpr, thresholds = roc_curve(y, preds)
  plt.plot([0,1], [0,1], linestyle = '--', label = 'Model teoretyczny
  zwracający predykcje losowe')
  plt.plot(fpr, tpr, marker = '.', label = 'Uzyskany model')
  plt.xlabel('Odsetek predykcji prawdziwie dodatnich (TPR)')
  plt.ylabel('Odsetek predykcji fałszywie dodatnich (FPR)')
  plt.legend()
```



Rysunek 14.2. Krzywa ROC uzyskana dla zbioru testowego

Przebieg uzyskanej krzywej ROC potwierdza to, co można stwierdzić na podstawie wartości ROC_AUC – model daje znacząco lepsze rezultaty niż model zwracający predykcje losowe.

Predykcje otrzymane przy użyciu metody predict_proba są zmiennymi ciągłymi, a więc nie można z takich wyników wyliczyć standardowych metryk sukcesu. Żeby to zrobić, należy zamienić predykcje na liczby całkowite wyrażające przynależność do klas. Zatem trzeba przyjąć próg, poniżej którego będą wartości oznaczające przyporządkowanie do klasy 0, a powyżej – do klasy 1. Są różne sposoby wyznaczania takiego progu. Poniżej przedstawiono kod funkcji implementującej metodę, w której wartość progowa wyznaczana jest jako największa średnia geometryczna TPR i TNR = (1 – FPR) dla całego zbioru danych (Liu, 2012). Jest to metoda, która zapewnia najwęższe przedziały ufności dokładności klasyfikacji oraz największą stabilność tej metryki w porównaniu z innymi często stosowanymi sposobami (Unal, 2017).

Po wyznaczeniu progu należy przeprowadzić dychotomizacje predykcji ciągłych. Można to zrobić, wykorzystując poniższy kod.

```
preds_train = lr.predict_proba(X_train)
preds_test = lr.predict_proba(X_test)

th = get_threshold(y_train, y_test, preds_train[:,1], preds_test[:,1])
th = get_threshold(y_train, y_test, preds_train[:,1], preds_test[:,1])

bool_preds = preds[:,1] > th

print(confusion_matrix(y_test, bool_preds))
print(classification_report(y_test, bool_preds))
```

Porównując macierze pomyłek uzyskane w wyniku zastosowania metod predict oraz predict_proba z progowaniem, można zauważyć, że pierwsza metoda dała nieco lepsze rezultaty – liczba poprawnie zaklasyfikowanych obiektów z klasy 0 jest o jeden większa. Ma to związek z dobraniem wartości progowej – zmieniając wartości progu, możemy przy użyciu tego samego modelu uzyskać różne wartości metryk sukcesu, zwłaszcza czułości i swoistości. Im większa wartość progu, tym więcej obiektów zostanie przydzielonych do klasy 0 i na odwrót – im mniejsza wartość progu, tym więcej obiektów zostanie przydzielonych do klasy 1.

Model regresji logistycznej można poddać optymalizacji mającej na celu dobór wartości hiperparametrów gwarantujących maksymalizację wybranej metryki. Poniżej przedstawiono kod, który prowadzi do maksymalizacji uzyskiwanej przez model wartości pola pod krzywą ROC. Innymi często wybieranymi metrykami są dokładność klasyfikacji oraz F1. Hiperparametrami, które poddano optymalizacji, są:

- C – parametr określający siłę regularyzacji, czyli ograniczenia wartości współczynników (wag) tego modelu, które pozwala zmniejszyć ryzyko przeuczenia modelu i nadmiernego dopasowania do danych;
- solver – algorytm wykorzystywany do optymalizacji wag;
- max_iter – maksymalna liczba iteracji (prób), które będą wykonane, by wyżej wymieniony algorytm osiągnął warunek zatrzymania (możliwe pod warunkiem, że jest zbieżny); jeżeli algorytm nie jest zbieżny lub optymalne wagi nie zostaną uzyskane podczas zdefiniowanej liczby iteracji, optymalizacja jest przerywana i wybierany jest najlepszy uzyskany wynik.

UWAGA

Optymalizacja hiperparametrów modelu może mieć również na celu minimalizację funkcji celu – wówczas metryką służącą do oceny optymalizowanego modelu jest miara popełnianego błędu. W przypadku modeli klasyfikacyjnych jest to zazwyczaj strata logarytmiczna, natomiast w przypadku modeli regresyjnych błąd średniokwadratowy lub średni błąd bezwzględny.

```
scoring = {'roc_macro': make_scorer(roc_auc_score)}

def objective(trial, model, get_space, X, y):
  model_space = get_space(trial)

  mdl = model(**model_space)
  scores = cross_validate(mdl, X, y, scoring = scoring,
                                     cv = StratifiedKFold(n_splits = 5),
                                     return_train_score = True)

  return np.mean(scores['test_roc_macro'])

def get_space(trial):
  space = {"C": trial.suggest_uniform("C", 0, 2),
           'max_iter': trial.suggest_int('max_iter', 100, 1000),
           "solver": trial.suggest_categorical("solver",
                                               ["lbfgs", "liblinear"])}
  return space

trials = 15 #liczba prób
model = LogisticRegression
study = optuna.create_study(direction = 'maximize')
study.optimize(lambda x: objective(x, model, get_space,
                                           X_train,
                                           y_train),
               n_trials = trials)

print('params: ', study.best_params)
lr = model(**study.best_params)
lr.fit(X_train, y_train)
preds = lr.predict(X_test)
print('test ROC_AUC: ', roc_auc_score(y_test, preds))
```

W przedstawionym przykładzie optymalizacja hiperparametrów modelu regresji logistycznej nie przyniosła poprawy rezultatów – metryki wyliczone za pomocą funkcji classification_report, czyli dokładność klasyfikacji, czułość, precyzja i F1 mają takie same wartości jak uzyskane przez model o domyślnych wartościach hiperparametrów. Może to mieć związek z niewielką liczbą iteracji wykonanych w procesie optymalizacji – wykonano jedynie 15 prób, więc maksimum globalne funkcji celu mogło nie zostać osiągnięte.

Zadanie 14.1. Przewidywanie prawdopodobieństwa zachorowania na cukrzycę typu 2

1. Ze strony https://www.kaggle.com/mathchi/diabetes-data-set pobierz dane i wczytaj je do struktury DataFrame. Uzyskasz tabelę składającą się z dziewięciu kolumn:
   - Pregnancies – liczba przebytych ciąż,
   - Glucose – stężenie glukozy dwie godziny po wykonaniu testu obciążenia glukozą,
   - BloodPressure – rozkurczowe ciśnienie krwi podane w milimetrach słupa rtęci,

- SkinThickness – grubość fałdu skórnego na ramieniu podana w milimetrach,
- Insulin – stężenie insuliny dwie godziny po wykonaniu testu obciążenia glukozą,
- BMI – współczynnik masy ciała,
- DiabetesPedigreeFunction – funkcja określająca ryzyko zachorowania na cukrzycę na podstawie wywiadu rodzinnego,
- Age – wiek,
- Outcome – przynależność do klas: 0 – nie zdiagnozowano cukrzycy, 1 – zdiagnozowano cukrzycę.

2. Podziel dane na zbiór uczący i testowy.
3. Przeprowadź trening klasyfikatora opartego na regresji logistycznej i jego ewaluację. Zastosuj wstępnie domyślne wartości hiperparametrów.
4. Przeprowadź optymalizację modelu – zdefiniuj funkcję celu *objective* tak, by maksymalizować pole pod krzywą ROC (ROC_AUC).
5. Przeprowadź trening modelu o optymalnych hiperparametrach. Predykcję na zbiorze testowym przeprowadź, używając metody predict oraz predict_proba. W przypadku drugiej metody wyznacz punkt odcięcia na krzywej ROC w celu dychotomizacji uzyskanych predykcji.
6. Wyznacz wartości metryk sukcesu i przeanalizuj wyniki – czy optymalizacja znacząco poprawiła skuteczność modelu? Czy wartości metryk uzyskane z użyciem predykcji wyliczonych metodą predict oraz predict_proba są takie same?
7. Zastanów się, która metryka jest istotniejsza w przypadku danych analizowanych w zadaniu: czułość czy precyzja? Dlaczego?

15 Metody wektorów nośnych

Zastosowanie: klasyfikacja.
Metoda: uczenie nadzorowane.

Maszyna wektorów nośnych jest narzędziem służącym do klasyfikacji obiektów i jej zadaniem jest wyznaczenie granicy decyzyjnej w postaci hiperpowierzchni. Położenie obiektów względem tej hiperpłaszczyzny przekłada się na decyzję klasyfikacyjną. Maszyna wektorów nośnych została opracowana w odpowiedzi na problemy innych, mniej uniwersalnych narzędzi, jakimi są klasyfikator maksymalizujący margines i klasyfikator wektorów nośnych. Zatem aby dobrze zrozumieć działanie maszyny wektorów nośnych, należy poznać najpierw algorytmy, na bazie których powstała.

Wszystkie trzy algorytmy wykorzystują pojęcie marginesu, rozumianego jako odstęp między granicą decyzyjną a najbliższym obiektem. W każdym z algorytmów margines należy zmaksymalizować. Algorytmy różnią się natomiast postacią funkcji określającej granicę decyzyjną. Klasyfikator maksymalizujący margines pozwala znaleźć hiperpłaszczyznę rozdzielającą obiekty na dwie klasy, zatem może być zastosowany jedynie w przypadku danych separowalnych liniowo, co znacznie ogranicza możliwości wykorzystania tego algorytmu. Klasyfikator wektorów nośnych, zwany też klasyfikatorem łagodnego marginesu, bazuje na tej samej koncepcji liniowej separacji obiektów, z tą różnicą, że stosowany jest tzw. margines miękki (ang. *soft margin*). W tym przypadku dopuszcza się, aby niektóre obiekty leżały w pasie marginesu, a nawet zostały błędnie zaklasyfikowane. W ten sposób algorytm maksymalizacji marginesu można zastosować również w przypadku danych, które nie są separowalne liniowo. Jednak w rozwiązaniu tym używa się nadal liniowej granicy decyzyjnej, co w przypadku danych wyraźnie nieliniowo separowanych będzie skutkować bardzo dużym błędem w klasyfikacji. Odpowiedzią na ten problem jest maszyna wektorów nośnych, w której stosuje się mechanizm przekształcający klasyfikator liniowy w klasyfikator o nieliniowej granicy decyzyjnej. W szczególnych przypadkach można rozszerzyć działanie maszyny wektorów nośnych, stosując margines miękki.

Wymienione algorytmy najłatwiej zrozumieć, wykorzystując geometryczną interpretację, w której obiekty w zbiorze treningowym są punktami w przestrzeni wejść z etykietami $y^{(j)} \in \{-1, 1\}$, $j = 1, ..., N$. Zbiór treningowy jest zatem postaci:

$$X = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(N)}, y^{(N)}) \in R^d \times \{-1, 1\}\}. \tag{15.1}$$

Klasyfikacja obiektów wymaga znalezienia granicy decyzyjnej, która w przypadku klasyfikatora liniowego jest hiperpłaszczyzną wyrażoną równaniem:

$$w^T x + w_0 = 0, \tag{15.2}$$

gdzie $w = [w_1, ..., w_d]^T$.

Dla tak określonej granicy decyzyjnej funkcja, na podstawie której dokonywana będzie klasyfikacja, jest postaci:

$$h_{w,w_0}(x) = \begin{cases} 1, & \text{dla} \quad w^T x + w_0 \geq 0 \\ -1, & \text{dla} \quad w^T x + w_0 < 0 \end{cases}. \tag{15.3}$$

15.1. Margines

Rozpatrując klasyfikację za pomocą ustalonej granicy decyzyjnej, możemy określić tzw. margines:

$$\epsilon = \min_{j=1,\ldots,N} \epsilon^{(j)}, \tag{15.4}$$

gdzie $\epsilon^{(j)}$ jest odległością obiektu $x^{(j)}$ od hiperpłaszczyzny rozdzielającej.

Wartość $\epsilon^{(j)}$ dla obiektu $x^{(j)}$ odpowiada zatem długości wektora prostopadłego do hiperpłaszczyzny i łączącego punkt $x^{(j)}$ z hiperpłaszczyzną. Stosując metodę mnożników Lagrange'a, można udowodnić poniższe twierdzenie o odległości punktu od hiperpłaszczyzny (Jurlewicz i Skoczylas, 2001).

TWIERDZENIE o odległości punktu od hiperpłaszczyzny

Rozważmy punkt $x^{(j)} = (x_1^{(j)}, x_2^{(j)}, x_d^{(j)})$ w przestrzeni euklidesowej oraz hiperpłaszczyznę daną równaniem $w^T x + \mathrm{w}_0 = 0$, gdzie $w^T = [w_1, \ldots, w_d]$.

Odległość punktu $x^{(j)}$ od tej hiperpłaszczyzny wynosi:

$$\epsilon^{(j)} = \frac{|w^T x^{(j)} + w_0|}{||w||}. \tag{15.5}$$

Z postaci funkcji $h_{w,\, w_0}(x)$ wynika, że $w^T x^{(j)} + w_0 \geq 0$ dla x należących do klasy $y = 1$, natomiast $w^T x^{(j)} + w_0 < 0$ dla x należących do klasy $y = -1$. Zatem w tym przypadku odległość punktu $x^{(j)}$ od pewnej hiperpłaszczyzny można zapisać jako:

$$\epsilon^{(j)} = y^{(j)} \frac{w^T x^{(j)} + w_0}{||w||}. \tag{15.6}$$

Zatem wyznaczenie marginesu wiąże się ze znalezieniem płaszczyzny leżącej pomiędzy obszarami klas, której odległość od najbliższych do niej obiektów z każdej klasy jest największa. Wtedy też obiekty te nazywane są wektorami nośnymi, czy inaczej wektorami wspierającymi lub podpierającymi (ang. *support vectors*).

15.2. Klasyfikator maksymalizujący margines

(ang. *Maximal Margin Classifier*, MMC)

Zadaniem algorytmu maksymalizującego margines jest wyznaczenie szczegółowego równania hiperpłaszczyzny wyznaczającej podział na klasy. Jak wskazuje nazwa algorytmu, warunkiem, który musi spełnić poszukiwana hiperpłaszczyzna, jest maksymalizacja marginesu dla ustalonego zbioru treningowego X. Zatem problem optymalizacyjny można zapisać jako:

$$\max_{w, w_0} \epsilon$$

pod warunkiem, że dla $j = 1, \ldots, N$

$$\epsilon^{(j)} = y^{(j)} \left(\frac{w^T}{||w||} x^{(j)} + \frac{w_0}{||w||} \right) \geq \epsilon,$$

co zapewnia spełnienie warunku z definicji marginesu, czyli $\epsilon = \min_{j=1,\ldots,N} \epsilon^{(j)}$.

UWAGA

Koncepcja znalezienia hiperpłaszczyzny takiej, która będzie maksymalizować margines, bazuje na rozumowaniu, że wynik klasyfikacji nowego obiektu jest tym bardziej prawdopodobny, im obiekt leży dalej od granicy decyzyjnej.

Powyższy warunek można również zapisać jako:

$$y^{(j)}(w^T x^{(j)} + w_0) \geq \epsilon \cdot ||w||. \tag{15.7}$$

Niech $\epsilon \cdot ||w|| = E$. Wtedy problem optymalizacyjny można zapisać jako:

$$\max_{w,w_0} \frac{E}{||w||}, \tag{15.8}$$

pod warunkiem, że dla $j = 1, ..., N$

$$y^{(j)}(w^T x^{(j)} + w_0) \geq E. \tag{15.9}$$

Należy zauważyć, że przeskalowanie w oraz w_0 nie zmieni wyniku optymalizacji. Zatem niech $E = 1$. Wtedy zadanie optymalizacyjne przyjmuje postać:

$$\max_{w,w_0} \frac{1}{||w||}$$

pod warunkiem, że dla $j = 1, ..., N$

$$y^{(j)}(w^T x^{(j)} + w_0) \geq 1. \tag{15.10}$$

Maksymalizację $1/||w||$ można zastąpić znalezieniem minimum $||w||$, co z kolei jest równoważne minimum $||w||^2$, czyli:

$$\max_{w,w_0} \frac{1}{||w||} = \min_{w,w_0} ||w|| = \min_{w,w_0} \frac{1}{2}||w||^2. \tag{15.11}$$

Wtedy problem

$$\min_{w,w_0} \frac{1}{2}||w||^2 \tag{15.12}$$

pod warunkiem, że dla $j = 1, ..., N$

$$y^{(j)}(w^T x^{(j)} + w_0) - 1 \geq 0, \tag{15.13}$$

jest wypukłym kwadratowym problemem optymalizacyjnym, co oznacza, że funkcja celu ma dokładnie jedno minimum, czyli jest to minimum globalne. Do rozwiązania problemu służy metoda mnożników Lagrange'a, w wyniku zastosowania której otrzymuje się optymalną, maksymalizującą margines hiperpłaszczyznę daną równaniem $w^T x + w_0 = 0$, gdzie:

$$w = \sum_{j \in S} \alpha_j y^{(j)} x^{(j)}, \tag{15.14}$$

$$w_0 = \frac{(\boldsymbol{w}^T x^* - \boldsymbol{w}^T x^{**})}{2}, j \in S, \tag{15.15}$$

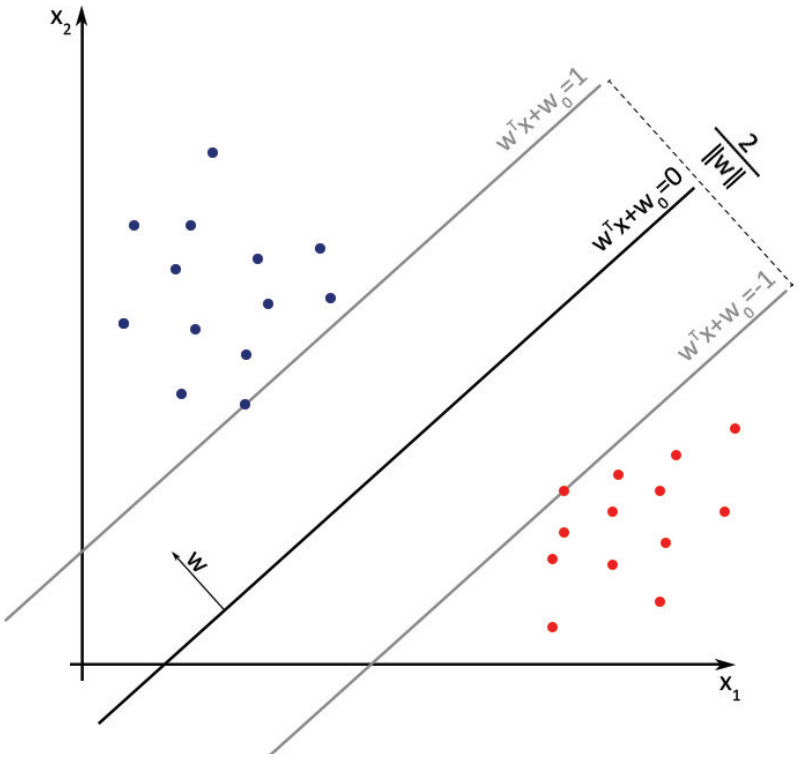
gdzie:

S – zbiór wektorów nośnych dla hiperpłaszczyzny,
α_j – wyznaczone mnożniki Lagrange'a takie, że $\alpha_j > 0 \Leftrightarrow j \in S$,
x^* i x^{**} – dowolne wektory nośne odpowiednio klasy $y = 1$ oraz $y = -1$.

Współczynnik w_0 umiejscawia hiperpłaszczyznę dzielącą na klasy w środku pomiędzy hiperpłaszczyznami wyznaczonymi przez wektory nośne każdej z klas (Koronacki i Ćwik, 2015). Przy tak przyjętych oznaczeniach otrzymujemy następującą postać reguły decyzyjnej otrzymanego klasyfikatora liniowego:

$$f(x) = \operatorname{sgn}\left(\sum_{i \in S} \alpha_j y^{(j)} x^{(j)} x + w_0 \right). \tag{15.16}$$

Na rysunku 15.1 przedstawiono przykładową hiperpłaszczyznę i margines dla obiektów określonych za pomocą dwóch atrybutów decyzyjnych i o wartościach atrybutu decyzyjnego ze zbioru {−1, 1}.



Rysunek 15.1. Przykładowa hiperpłaszczyzna i margines dla obiektów o dwóch atrybutach warunkowych i wartościach atrybutu decyzyjnego ze zbioru {−1, 1}

15.3. Klasyfikator wektorów nośnych

(ang. *Support Vector Classifier*, SVC)

W większości przypadków dane nie są liniowo separowalne, czyli nie istnieje hiperpłaszczyzna rozdzielająca. Zatem w przypadkach tych algorytm maksymalizacji marginesu nie przyniesie rozwiązania. Dlatego też stosuje się klasyfikator wektorów nośnych, w którym hiperpłaszczyznę należy wyznaczyć tak, aby rozdzielała większość obiektów, a nie wszystkie obiekty. Warunek ten zapewnia tzw. margines miękki. Niech $C \geq 0$ oznacza parametr regulujący (ang. *tuning parameter*) oraz δ_j parametry luzu (ang. *slack parameters*), dzięki którym dopuszczone są pojedyncze błędne decyzje klasyfikacyjne, czyli nieliczne obiekty mogą zostać po błędnej stronie hiperpłaszczyzny i marginesu. Można wtedy zapisać problem optymalizacyjny, którego rozwiązanie prowadzi do znalezienia hiperpłaszczyzny rozdzielającej, jako (Cortes i Vapnik, 1997):

$$\max_{w, w_0, \delta_1, \ldots, \delta_N} \epsilon, \tag{15.17}$$

Warunki takiego zapisu problemu optymalizacyjnego są następujące:

$$y^{(j)}(w^T x^{(j)} + w_0) \geq \epsilon(1 - \delta_j), \tag{15.18}$$

$$||w|| = 1, \tag{15.19}$$

$$\delta_j \geq 0, \tag{15.20}$$

$$\sum_{i=1}^{N} \delta_j \leq C. \tag{15.21}$$

Rozwiązanie powyższego problemu optymalizacyjnego prowadzi do uzyskania hiperpłaszczyzny. Następnie, podobnie jak w klasyfikatorze maksymalizującym margines, zaklasyfikowanie nowego obiektu odbywa się na podstawie znaku wartości, jaką przyjmuje hiperpłaszczyzna w punkcie reprezentowanym przez ten obiekt.

UWAGA

Parametry δ_j informują o położeniu obserwacji $x^{(j)}$ względem hiperpłaszczyzny i marginesu:
- gdy $\delta_j = 0$, obiekt $x^{(j)}$ leży po właściwej stronie marginesu,
- gdy $\delta_j > 0$, $x^{(j)}$ naruszył margines,
- gdy $\delta_j > 1$, $x^{(j)}$ leży po złej stronie hiperpłaszczyzny.

UWAGA

Parametr C określony we wzorze (15.21) ogranicza od góry sumę wszystkich δ_j, czyli determinuje maksymalną liczbę naruszeń marginesu oraz błędnych klasyfikowań po błędnej stronie hiperpłaszczyzny, w szczególności:
- gdy $C = 0$, żaden obiekt nie może naruszyć granicy marginesu, zatem problem sprowadza się do klasyfikatora maksymalizującego margines,
- gdy $C > 0$, dopuszcza się co najwyżej C obserwacji, które zostaną błędnie sklasyfikowane.

Zatem wraz ze wzrostem wartości parametru C klasyfikator jest mniej wrażliwy na naruszanie marginesu i błędne klasyfikacje. Natomiast gdy wartość parametru C maleje, opracowywany klasyfikator jest bardziej restrykcyjny.

Parametr C jest hiperparametrem, który należy określić w pierwszych krokach wykonywania algorytmu. Wartość tego parametru może zostać dobrana przez sprawdzian krzyżowy, opisany na przykładzie doboru hiperparametru w rozdziale 8.

15.4. Maszyna wektorów nośnych

(ang. *Support Vector Machine*, SVM)

W celu ułatwienia zrozumienia, jak działa maszyna wektorów nośnych, przedstawiony zostanie najpierw ogólny mechanizm przekształcania klasyfikatora liniowego w klasyfikator o nieliniowej granicy decyzyjnej. Automatyzacja tego procesu jest realizowana przez maszynę wektorów nośnych (Boser i in., 1992).

Niech $X = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(N)}, y^{(N)}) \in R^d \times \{-1, 1\}\}$ będzie zbiorem o nieliniowej granicy klas. Zatem hiperpłaszczyzna, którą można otrzymać w wyniku zastosowania przedstawionych do tej pory narzędzi, nie będzie skutecznym klasyfikatorem. W przypadku danych separowalnych nieliniowo obiekty można spróbować analizować w wyższym wymiarze. Uzyskuje się to przez dodanie zmiennych, które są wynikiem zastosowania pewnych funkcji na pierwotnych zmiennych wejściowych. Bardzo często wykorzystuje się w tym celu funkcje wielomianowe. Prostym przykładem wykorzystania funkcji wielomianowych jest zdefiniowanie dodatkowych zmiennych jako $x_1^2, x_2^2, ..., x_d^2$.

Niech płaszczyzna odpowiadająca granicy decyzyjnej będzie wyrażona równaniem:

$$\sum_{i=1}^{d} v_i x_i^2 + \sum_{i=1}^{d} w_i x_i + w_0 = 0. \tag{15.22}$$

Wtedy problem optymalizacyjny określony wcześniej dla klasyfikatora wektorów nośnych przyjmie postać:

$$\max_{w_0, w_1, \ldots, w_d, v_1, \ldots, v_d, \delta_1, \ldots, \delta_N} \epsilon, \tag{15.23}$$

pod następującymi warunkami:

$$y^{(j)} \left(\sum_{i=1}^{d} v_i (x_i^{(j)})^2 + \sum_{i=1}^{d} w_i x_i^{(j)} + w_0 \right) \geq \epsilon(1 - \delta_j), \tag{15.24}$$

$$\sum_{i=1}^{d} v_i + \sum_{i=1}^{d} w_i = 1, \tag{15.25}$$

$$\delta_j \geq 0, \tag{15.26}$$

$$\sum_{i=1}^{N} \delta_j \leq C. \tag{15.27}$$

UWAGA

Jako nowych zmiennych można użyć dowolnych wielomianów wybranego stopnia zbudowanych z pierwotnych zmiennych wejściowych, co oznacza, że można również uwzględnić interakcje zmiennych postaci $x_i x_j$, gdzie $i \neq j$. Często stosuje się również inne funkcje. Należy jednak pamiętać, że zbyt duża liczba cech niekoniecznie przełoży się na lepiej dobraną granicę decyzyjną. W szczególności przeniesienie danych do zbyt wysokiego wymiaru wiąże się z przekleństwem wymiarowości (opisano w rozdziale 9) i ryzykiem przetrenowania modelu, czyli zbyt dużego dopasowania kształtu hiperpowierzchni dyskryminacyjnej do danych. Wynikiem przetrenowania modelu jest słabe uogólnienie i mała skuteczność w klasyfikowaniu nieznanych wcześniej danych.

Powyższą ideę analizowania zadanego problemu w przestrzeni o wyższym wymiarze można uogólnić. Niech $\phi : R^d \rightarrow R^m$ jest funkcją wektorową przenoszącą zmienne wejściowe w wyższy wymiar ($m > d$) w ten sposób, że dodatkowe zmienne są funkcjami zmiennych wejściowych. Wtedy dotychczasowy problem optymalizacyjny dany równaniami (15.17)–(15.21) możemy zapisać jako:

$$\max_{w, w_0, \delta_1, \ldots, \delta_N} \epsilon. \tag{15.28}$$

pod następującymi warunkami:

$$y^{(j)} (w^T \phi(x^{(j)}) + w_0) \geq \epsilon(1 - \delta_j), \tag{15.29}$$

$$||w|| = 1, \tag{15.30}$$

$$\delta_j \geq 0, \tag{15.31}$$

$$\sum_{i=1}^{N} \delta_j \leq C. \tag{15.32}$$

Okazuje się jednak, że rozwiązaniem powyższego problemu optymalizacyjnego jest nadal równanie postaci $f(x) = \text{sgn}\left(\sum_{i \in S} \alpha_j y^{(j)} x^{(j)} x + w_0 \right)$, przy czym zamiast iloczynów skalarnych $x^{(j)} x$ należy obliczyć iloczyny skalarne $\phi(x^{(j)})\phi(x)$. Etap ten może zostać usprawniony przez zastosowanie wybranej funkcji jądra.

UWAGA

Funkcja jądra jest symetryczną i nieujemnie określoną funkcją dwóch zmiennych $K: R^d \times R^d \to R$ postaci $K(x^{(j)}, y^{(k)}) = \phi$, spełniającą założenia twierdzenia Mercera (Mercer, 1909). Jądro interpretowane jest jako ilościowe określenie podobieństwa dwóch wektorów.

Przykładowymi funkcjami jądra stosowanymi w maszynie wektorów nośnych są:

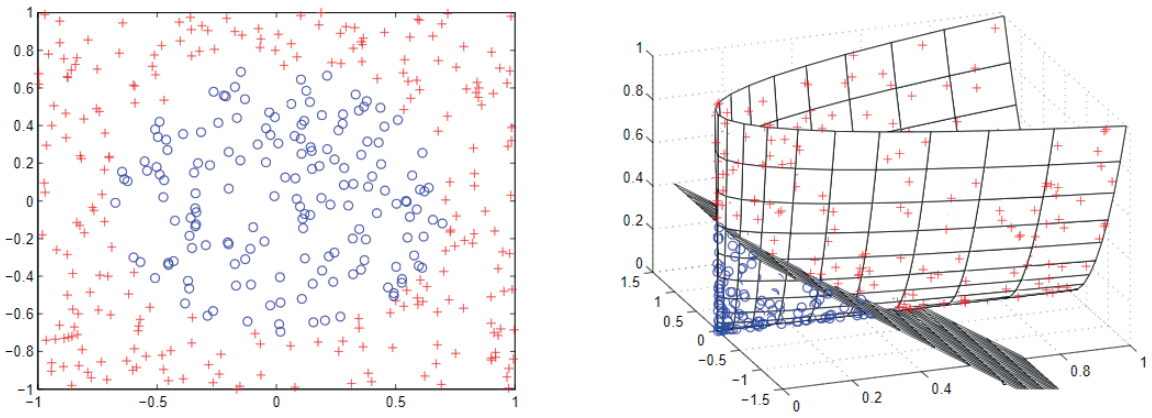
- **jądro liniowe** (ang. *linear kernel*): $K(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^T\boldsymbol{y}$,
- **jądro wielomianowe** (ang. *polynomial kernel*): $K(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}^T\boldsymbol{y} + 1)^m$, $m \in \mathbb{N}$,
- **jądro gaussowskie** (ang. *Gaussian kernel/radial basis function kernel/RBF kernel*): $K(\boldsymbol{x}, \boldsymbol{y}) = e^{-\frac{\|x-y\|^2}{2\sigma^2}}$, gdzie $\sigma > 0$.

Decydując się na jedną z funkcji jądrowych, można zastosować tzw. **sztuczkę jądra** (ang. *kernel trick*). Pozwala ona zrezygnować z wyznaczania postaci funkcji ϕ i obliczania iloczynu skalarnego $\phi(x^{(j)})\phi(x)$, ponieważ regułę decyzyjną można zapisać jako:

$$f(x) = \operatorname{sgn}\left(\sum_{i \in S} \alpha_j y^{(j)} K(x^{(j)}, x) + w_0\right). \tag{15.33}$$

Dzięki sztuczce jądra otrzymujemy optymalną hiperpowierzchnię dyskryminacyjną w rozszerzonej przestrzeni cech $\phi(x)$ bez znajomości tej przestrzeni. Podstawą tego podejścia jest twierdzenie Mercera, które gwarantuje istnienie tej przestrzeni (Mercer, 1909).

Na rysunku 15.2 przedstawiono sztuczkę jądra. W przykładzie wykorzystano wielomianową funkcję wektorową $\phi: R^2 \to R^3$: $\phi(x_1, x_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$. W ten sposób dane określone w dwuwymiarowej przestrzeni cech zostały przeniesione do trójwymiarowej przestrzeni cech, gdzie dane stają się separowalne liniowo, jak na poniższym rysunku. W rozwiązaniu analitycznym do uzyskania reguły decyzyjnej wystarczyła znajomość funkcji jądra $K(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y}) = (\boldsymbol{x}T\boldsymbol{y})^2$ i zastosowanie sztuczki jądra zgodnie ze wzorem (15.33).



Rysunek 15.2. Przykładowe nieliniowo separowalne dane treningowe przed zastosowaniem i po zastosowaniu sztuczki jądra (Machine Learner, 2022)

15.5. Maszyna wektorów nośnych w klasyfikacji wieloklasowej

Do tej pory opisane algorytmy wykorzystujące wektory nośne dotyczyły wyłącznie klasyfikacji binarnej, czyli atrybut decyzyjny mógł przyjąć jedną z dwóch wartości $Y = \{-1, 1\}$. Jednak maszynę wektorów nośnych można również stosować w przypadku klasyfikacji wieloklasowych. Najczęściej wykorzystuje się wówczas zwielokrotnienie klasyfikatorów binarnych, w których zestawiane są pary klas lub pojedyncze klasy z całą resztą. Są to algorytmy typu jeden kontra jeden (ang. *one-versus-one*) lub jeden kontra wszystkie (ang. *one-versus-all*). W literaturze można również znaleźć inne podejścia łączenia binarnych maszyn wektorów nośnych w celu dokonania klasyfikacji wieloklasowej (Vapnik, 1998).

Kroki algorytmu (jeden kontra wszystkie)

Niech $X = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(N)}, y^{(N)}) \in R^d \times \{1, 2, ..., p\}\}$ jest zbiorem treningowym, przy czym wartości atrybutu decyzyjnego rozważanych danych należą do zbioru p-elementowego, czyli p oznacza liczbę klas i $p > 2$.

1. Niech $i := 1$.
2. Niech $A = \{(x^{(j)}, y^{(j)}) : y^{(j)} = i\}$ oraz $B = X - A$.
3. Nadpisanie wartości atrybutu decyzyjnego w zbiorze A wartością 1, a w zbiorze B wartością −1.
4. Potraktowanie zbioru $A \cup B$ jako treningowego i zbudowanie maszyny wektorów nośnych, w wyniku czego otrzymana zostaje reguła $f_i(x)$.
5. Sprawdzenie, czy osiągnięto maksymalną liczbę kroków:
   - jeżeli $i < p$, to $i := i + 1$ i powrót do kroku nr 2,
   - jeżeli $i = p$, to przejście do kolejnego kroku.
6. Wyznaczenie reguły decyzyjnej klasyfikacji wieloklasowej jako:

$$f(x) = \text{argmax}_{i = 1, ..., p} f_i(x). \tag{15.34}$$

Kroki algorytmu (jeden kontra jeden)

Niech $X = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(N)}, y^{(N)}) \in R^d \times \{1, 2, ..., p\}\}$ jest zbiorem treningowym, przy czym wartości atrybutu decyzyjnego rozważanych danych należą do zbioru p-elementowego, czyli p oznacza liczbę klas i $p > 2$.

1. Niech $Z_i = \{(x^{(j)}, y^{(j)}): y^{(j)} = i\}$.
2. Niech $G = \left\{ G_n = Z_k \cup Z_l : k \neq l,\ k, l = 1, 2, ... p,\ n = 1, 2, ..., \binom{p}{2} \right\}$.

   Wyznaczamy wszystkie możliwe pary $\{Z_k, Z_l\}$, $k \neq l$ (liczba takich kombinacji dwuelementowych to oczywiście $|G| = \binom{p}{2}$) i kolejne zbiory G_n są sumą zbiorów w danej parze. W ten sposób powstaje $\binom{p}{2}$ zbiorów treningowych, z których w każdym występują tylko dwie klasy obiektów, zatem możliwe jest przeprowadzenie klasyfikacji binarnej. Należy również zauważyć, że każda z p klas występuje w takich parach $p - 1$ razy.
3. Niech $n := 1$.
4. Potraktowanie zbioru G_n jako treningowego i zbudowanie maszyny wektorów nośnych, w wyniku czego otrzymana zostaje reguła decyzyjna $f_n(x)$.

   UWAGA

   Aby wykorzystać maszynę wektorów nośnych, należy zamienić wartości atrybutów decyzyjnych w zbiorze G_n na wartości 1 i −1. Jednak aby przejść do kolejnych kroków, trzeba w wynikowej regule decyzyjnej $f_n(x)$ powrócić do oryginalnego oznaczenia klas $k, l \in \{1, 2, ..., p\}$.

5. Sprawdzenie, czy osiągnięto maksymalną liczbę kroków:
   - jeżeli $n < \binom{p}{2}$, to $n := n + 1$ i powrót do kroku nr 4,
   - jeżeli $n = \binom{p}{2}$, to przejście do kolejnego kroku.
6. Określenie reguły decyzyjnej klasyfikacji wieloklasowej: nowy obiekt przypisujemy do tej klasy, do której został najczęściej zaklasyfikowany wśród $\binom{p}{2}$ klasyfikatorów binarnych.

Przykład 15.1. Automatyczny dobór leków na podstawie parametrów fizjologicznych pacjenta

Pod adresem: https://www.kaggle.com/datasets/prathamtripathi/drug-classification znajdują się dane, które zostaną wykorzystane do zademonstrowania działania algorytmu SVM. W skład danych wchodzi sześć zmiennych:

1) Age – wiek pacjenta,
2) Sex – płeć pacjenta,
3) Blood Pressure Levels (BP) – ciśnienie krwi,
4) Cholesterol Levels – poziom cholesterolu we krwi,
5) Na to Potassium Ratio – stosunek stężenia sodu do stężenia potasu we krwi,
6) Drug type – lek przepisany pacjentowi.

Analizując wartości pierwszych pięciu zmiennych, którymi są parametry fizjologiczne, można spróbować określić, który lek z dużym prawdopodobieństwem okaże się skuteczny u danego pacjenta.

Przed przystąpieniem do uczenia modelu należy odpowiednio przygotować dane. Zostaną one wczytane do struktury DataFrame, a wiersze zawierające brakujące dane zostaną usunięte. Zarówno etykiety, jak i niektóre parametry obiektów są zmiennymi kategorialnymi, konieczne jest więc ich zakodowanie za pomocą zmiennych numerycznych. Zostanie do tego wykorzystana funkcja OrdinalEncoder, która posłuży do zakodowania wartości parametrów, oraz funkcja LabelEncoder, która posłuży do zakodowania etykiet. Następnie dane zostaną podzielone na zbiór uczący i testowy oraz poddane standaryzacji.

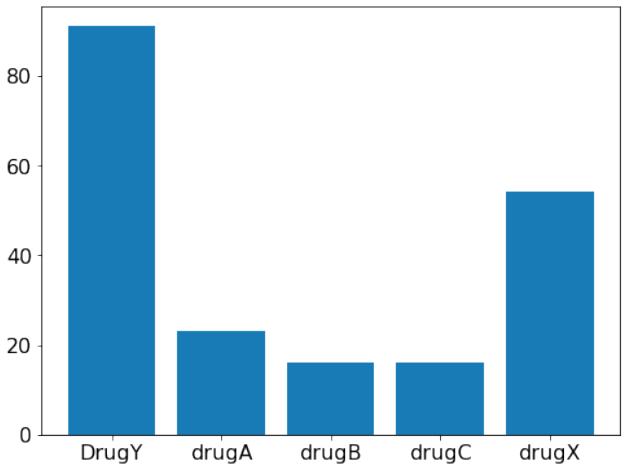
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import (StandardScaler,
                                   OrdinalEncoder,
                                   LabelEncoder)
from sklearn.model_selection import train_test_split

#wczytanie danych i usunięcie brakujących danych
df = pd.read_csv('drug200.csv')
df.dropna(inplace = True)

#utworzenie wektora etykiet i macierzy parametrów
columns = df.columns
data = df[columns[:-1]].to_numpy()
labels = df[,Drug'].to_numpy()

#zakodowanie zmiennych kategorialnych za pomocą zmiennych liczbowych
enc = OrdinalEncoder()
data = enc.fit_transform(data)
label_enc = LabelEncoder()
labels = label_enc.fit_transform(labels)

#podział danych na zbiór uczący i testowy
X_train, X_test, y_train, y_test = train_test_split(data, labels,
                                                    stratify = labels,
                                                    random_state = 42)
```

```
#standaryzacja danych
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Kolejnym krokiem jest sprawdzenie liczebności poszczególnych klas. Od tego, czy analizowane dane są zrównoważone pod względem liczebności klas, czy też są niezrównoważone, zależy dobór metryk sukcesu pozwalających dobrze ocenić jakość nauczonego klasyfikatora.

Liczebność klas zostanie sprawdzona przy użyciu funkcji Counter zaimplementowanej w bibliotece collections, która zwraca liczbę obiektów przynależących do poszczególnych klas. Dodatkowo zostanie wyświetlony wykres obrazujący liczebność klas (rys. 15.3).

```
from collections import Counter
import matplotlib.pyplot as plt

print(Counter(labels))
plt.figure(figsize = (8, 6))
plt.hist(labels,
         bins = np.arange(len(np.unique(labels))+1)-0.5,
         rwidth = 0.8)
drugs_names = label_enc.inverse_transform(np.unique(labels))
plt.xticks(np.unique(labels),
           drugs_names, ha = 'center',
           fontsize = 16)
plt.yticks(fontsize = 16)
plt.show()
```



Rysunek 15.3. Liczebność klas występujących w analizowanym zbiorze danych

Jak można zauważyć, dysproporcje w liczebności klas są dość duże – leki B i C występują zaledwie 16 razy, podczas gdy lek Y występuje 91 razy. Należy to uwzględnić podczas treningu i ewaluacji modelu, w szczególności na etapie optymalizacji hiperparametrów. Nieprawidłowy dobór metryki może skutkować uzyskaniem modelu, który będzie cechował się bardzo małą czułością w stosunku do mniej licznych klas.

Jako pierwszy przeprowadzony zostanie trening modelu o domyślnych wartościach hiperparametrów. W celu inicjalizacji klasyfikatora wektorów nośnych należy skorzystać z klasy SVC w bibliotece scikit-learn. Trening oraz predykcja na zbiorze testowym prowadzone są przy użyciu metod fit oraz predict.

```
from sklearn.svm import SVC
from sklearn.metrics import (confusion_matrix, classification_report)

svm = SVC()
svm.fit(X_train_scaled, y_train)
preds = svm.predict(X_test_scaled)
print(classification_report(y_test, preds))
print(confusion_matrix(y_test, preds))
```

Analizując wyniki klasyfikacji uzyskane za pomocą funkcji classification_report, można zauważyć, że istotnie najmniej skuteczna klasyfikacja dotyczy leku B – w tym przypadku czułość wynosi zaledwie 0,5, czyli połowa obiektów ze zbioru testowego należących do tej klasy została niepoprawnie zaklasyfikowana. Warto również zauważyć, że z powodu bardzo małej liczebności tej klasy w zbiorze testowym znalazły się zaledwie cztery obiekty. Zatem formułowanie wniosków na podstawie przeprowadzonych analiz powinno być ostrożne – parametry cechujące obiekty należące do tej klasy mogą nie odzwierciedlać dobrze rozkładu parametrów w całej populacji.

Następnie przeprowadzona zostanie optymalizacja hiperparametrów modelu SVC. Optymalizacji poddane zostaną cztery hiperparametry:

1) *C* – parametr określający siłę regularyzacji;
2) max_iter – maksymalna liczba iteracji (prób), które będą wykonane, by algorytm optymalizujący wagi osiągnął warunek zatrzymania (możliwe pod warunkiem, że jest zbieżny); jeżeli algorytm ten nie jest zbieżny lub optymalne wagi nie zostaną uzyskane podczas zdefiniowanej liczby iteracji, optymalizacja jest przerywana i przyjmowany jest najlepszy uzyskany wynik;
3) kernel – funkcja jądra użyta przez algorytm SVM;
4) degree – stopień wielomianu w sytuacji, gdy używana jest wielomianowa funkcja jądra.

Pozostałe hiperparametry modelu można znaleźć w dokumentacji klasy SVC pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

Do ewaluacji modelu podczas optymalizacji została wybrana metryka F1 (opisana w rozdziale 5) uwzględniająca zarówno precyzję, jak i czułość modelu, a zatem pozwalająca dobrze go ocenić w sytuacji, gdy zbiór danych jest niezrównoważony. Ponieważ w analizowanych danych występują więcej niż dwie klasy, konieczne było określenie sposobu uśredniania metryki F1 uzyskiwanej dla poszczególnych klas. Rekomendowaną metodą uśredniania przy dużej dysproporcji liczebności klas jest wyznaczanie średniej ważonej, dlatego zdecydowano o jej użyciu.

```
import optuna
from sklearn.model_selection import (cross_validate, StratifiedKFold)
from sklearn.metrics import make_scorer, f1_score

scoring = {'F1': make_scorer(f1_score, average = 'weighted')}
#funkcja celu
def objective(trial, model, get_space, X, y):
  model_space = get_space(trial)

  mdl = model(**model_space)
```

```
  scores = cross_validate(mdl, X, y, scoring = scoring,
                                     cv = StratifiedKFold(n_splits = 5),
                                     return_train_score = True)

  return np.mean(scores['test_F1'])

#przestrzeń hiperparametrów
def get_space(trial):
  space = {"C": trial.suggest_uniform("C", 0, 2),
           "max_iter": trial.suggest_int("max_iter", 100, 1000),
           "kernel": trial.suggest_categorical("kernel",
                                               ["linear", "poly",
                                                "rbf", "sigmoid"]),
           "degree": trial.suggest_int("degree", 2,5)}
  return space

trials = 120 #liczba prób

model = SVC

#optymalizacja
study = optuna.create_study(direction = 'maximize')
study.optimize(lambda x: objective(x, model, get_space,
                                           X_train_scaled,
                                           y_train),
               n_trials = trials)

#dobrane wartości hiperparametrów
print('params: ', study.best_params)
#trening modelu o optymalnych wartościach hiperparametrów
svm = model(**study.best_params)
svm.fit(X_train_scaled, y_train)

#ewaluacja modelu
preds = svm.predict(X_test_scaled)
print(classification_report(y_test, preds))
print(confusion_matrix(y_test, preds))
```

Po przeprowadzeniu optymalizacji hiperparametrów modelu wyniki na zbiorze testowym uległy poprawie – lek A został poprawnie zaklasyfikowany we wszystkich przypadkach, podczas gdy wcześniej jeden obiekt był zaklasyfikowany nieprawidłowo. Również w przypadku najmniej licznych klas można zauważyć poprawę – lek B został poprawnie zaklasyfikowany w trzech z czterech przypadków, co oznacza poprawę czułości modelu o 25 punktów procentowych. Gorsze rezultaty uzyskano jedynie w przypadku leku X – po przeprowadzeniu optymalizacji jeden obiekt przestał być prawidłowo klasyfikowany. Jednak biorąc pod uwagę wzrost dokładności klasyfikacji oraz poprawę dotyczącą pozostałych leków, można uznać, że optymalizacja hiperparametrów pozwoliła uzyskać model o zauważalnie lepszych zdolnościach klasyfikacyjnych.

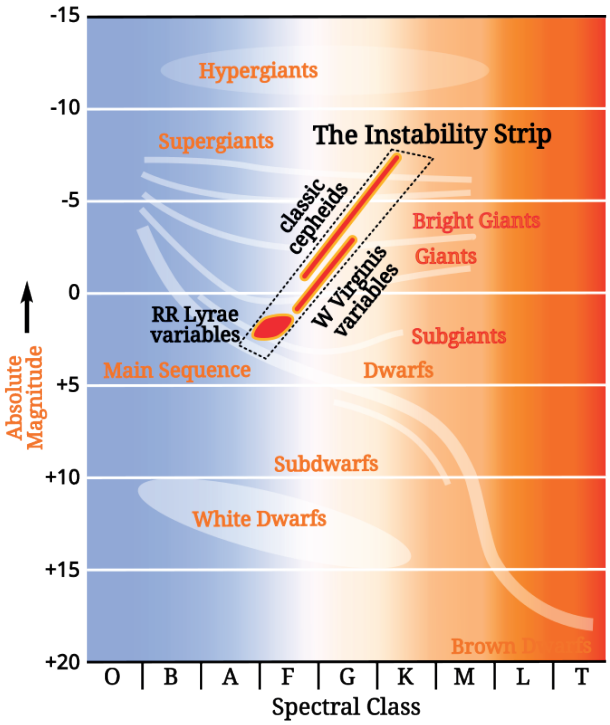
Zadanie 15.1. Klasyfikacja gwiazd ze względu na klasę jasności

Pod adresem: https://www.kaggle.com/datasets/deepu1109/star-dataset znajdują się dane dotyczące gwiazd należących do różnych klas jasności. Każda gwiazda opisana jest przez sześć parametrów:

1) Absolute Temperature (K) – temperatura,
2) Relative Luminosity(L/Lo) – jasność względna obliczona w odniesieniu do jasności Słońca,
3) Relative Radius (R/Ro) – promień względny obliczony w odniesieniu do promienia Słońca,
4) Absolute Magnitude (Mv) – absolutna wielkość gwiazdowa,
5) Star Color – kolor,
6) Spectral Class – typ widmowy.

Klasy jasności poszczególnych gwiazd zapisane są w kolumnie star type i są oznaczone numerami (rys. 15.4):

0 – brązowy karzeł (ang. *brown dwarf*),
1 – czerwony karzeł (ang. *red dwarf*),
2 – biały karzeł (ang. *white dwarf*),
3 – gwiazda ciągu głównego (ang. *main sequence*),
4 – nadolbrzym (ang. *supergiant*),
5 – hiperolbrzym (ang. *hypergiant*).



Rysunek 15.4. Podział gwiazd ze względu na klasę jasności oraz typ widmowy
(źródło: https://upload.wikimedia.org/wikipedia/commons/c/c7/HR-diag-instability-strip.svg, dostęp: 20.04.2022)

W celu przeprowadzenia klasyfikacji gwiazd ze względu na klasę jasności wykonaj następujące kroki.

1. Pobierz dane i wczytaj je do struktury DataFrame.
2. Przygotuj dane do przeprowadzenia treningu i ewaluacji klasyfikatora SVC: usuń dane brakujące, zakoduj zmienne kategorialne za pomocą zmiennych numerycznych, podziel dane na zbiory uczący i testowy oraz ustandaryzuj dane.
3. Przeprowadź trening i ewaluację klasyfikatora o domyślnych wartościach hiperparametrów.
4. Przeprowadź optymalizację hiperparametrów modelu, a następnie jego trening i ewaluację. Jakiej metryki należy użyć do ewaluacji modelu w trakcie optymalizacji i dlaczego?
5. Przeanalizuj wyniki – czy optymalizacja poprawiła zdolności predykcyjne modelu?

16 Drzewa klasyfikacyjne i lasy losowe

Zastosowanie: klasyfikacja.
Metoda: uczenie nadzorowane.

16.1. Drzewa klasyfikacyjne

Drzewa klasyfikacyjne nazywane są również drzewami decyzyjnymi. Posługując się nomenklaturą z zakresu teorii grafów, drzewo klasyfikacyjne można określić jako skierowany, acykliczny i spójny graf, który ma tylko jeden wierzchołek zwany korzeniem. Przyjęło się, że korzeń sytuowany jest na górze, tak więc drzewo rośnie z góry na dół. W korzeniu znajduje się cały zbiór treningowy, w którym obiekty określone są za pomocą zmiennej zależnej (atrybut decyzyjny) oraz d zmiennych niezależnych (atrybuty warunkowe, inaczej cechy). Załóżmy, że zmienna zależna ma p klas. Zatem:

$$X = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(N)}, y^{(N)}) \in R^d \times \{1, 2, ..., p\}\}. \tag{16.1}$$

Korzeń jest szczególnym przypadkiem węzła decyzyjnego. W węzłach decyzyjnych dokonuje się podziału zbioru zgodnie z przyjętym dla niego kryterium podziału. W korzeniu dokonuje się pierwszego takiego podziału. Kryteria podziału są ustalonymi warunkami wyrażonymi zwykle zdaniami logicznymi dotyczącymi zmiennych niezależnych, a ich zastosowanie ma doprowadzić do podziału zbioru. Gałęzie to odcinki łączące zbiór, którego podział jest dokonywany w danym węźle decyzyjnym, ze zbiorami, które są wynikiem tego podziału i które stają się nowymi węzłami decyzyjnymi. W nowo powstałych węzłach decyzyjnych niezależnie ustala się nowe kryteria podziału. Podziały są dokonywane tak długo, aż w danych węzłach decyzyjnych znajdują się głównie obiekty tej samej klasy. Jeżeli nie jest dokonywany dalszy podział danego węzła, to taki węzeł nazywa się liściem. W tabeli 16.1 zestawiono najważniejsze elementy drzewa wraz z określeniem pełnionych funkcji.

Tabela 16.1. Struktura drzewa decyzyjnego

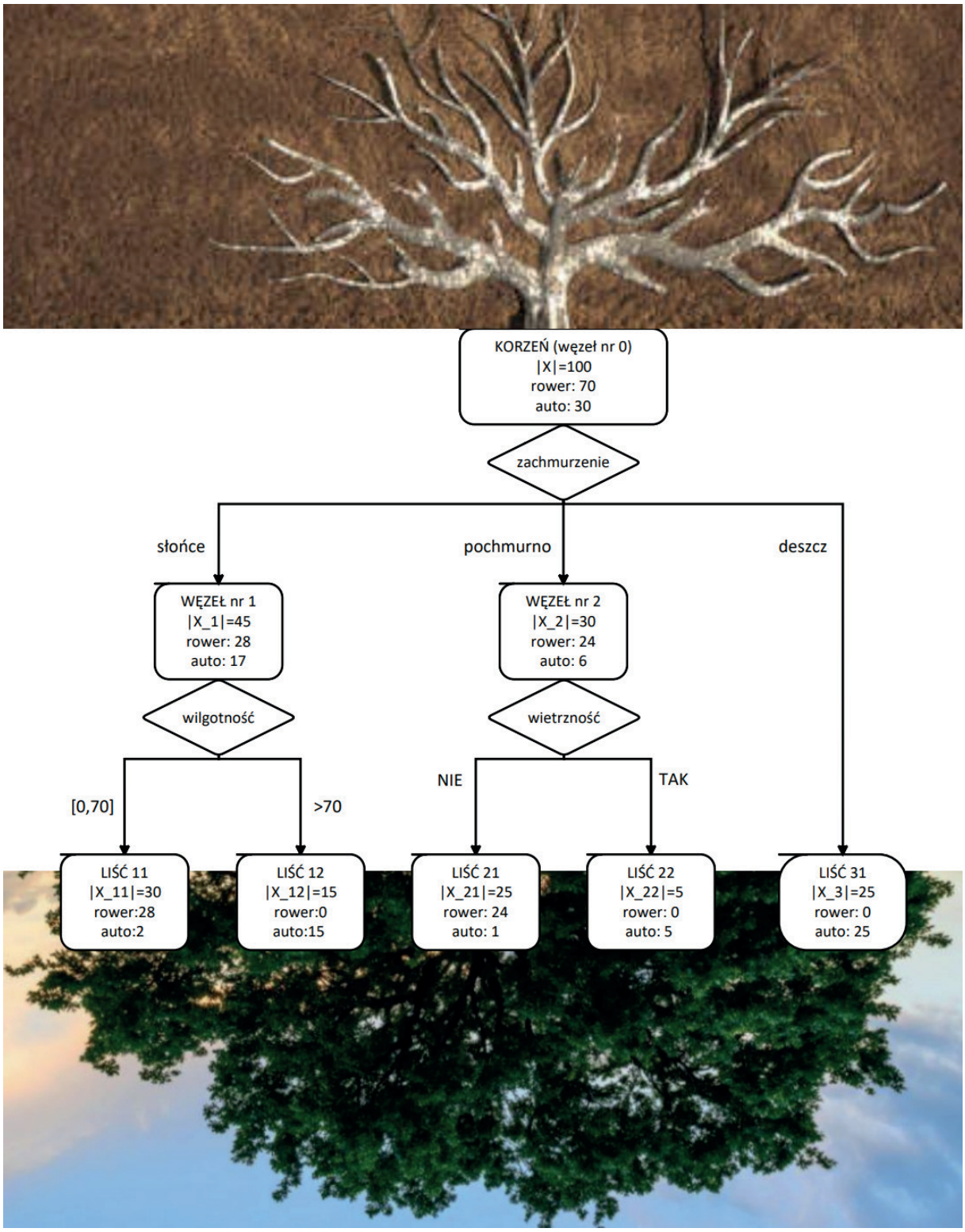
Element drzewa	Funkcja
Węzeł decyzyjny	punkt podziału ze względu na wybrane kryterium podziału
Kryterium podziału	warunek dla wybranej cechy, według którego dokonywany jest podział w danym węźle
Korzeń	pierwszy węzeł decyzyjny
Liście	węzły końcowe

W drzewach klasyfikacyjnych używa się również nazewnictwa określającego hierarchię pomiędzy węzłami decyzyjnymi. Węzeł, z którego gałęzie prowadzą do innych węzłów, nazywany jest rodzicem, a połączone z nim poniżej węzły nazywane są jego dziećmi. Dzieci też mogą być rodzicami, a ich dzieci zwane są potomkami wcześniejszych węzłów, z którymi są połączone przez swoich rodziców (Koronacki i Ćwik, 2015).

Na rysunku 16.1 przedstawiono przykładowe drzewo klasyfikacyjne. Problem, który został zamodelowany tym drzewem, to klasyfikacja obiektów opisanych za pomocą trzech zmiennych niezależnych – zachmurzenie, wilgotność oraz wietrzność. Zmienna zależna jest binarna i przyjmuje jedną z dwóch

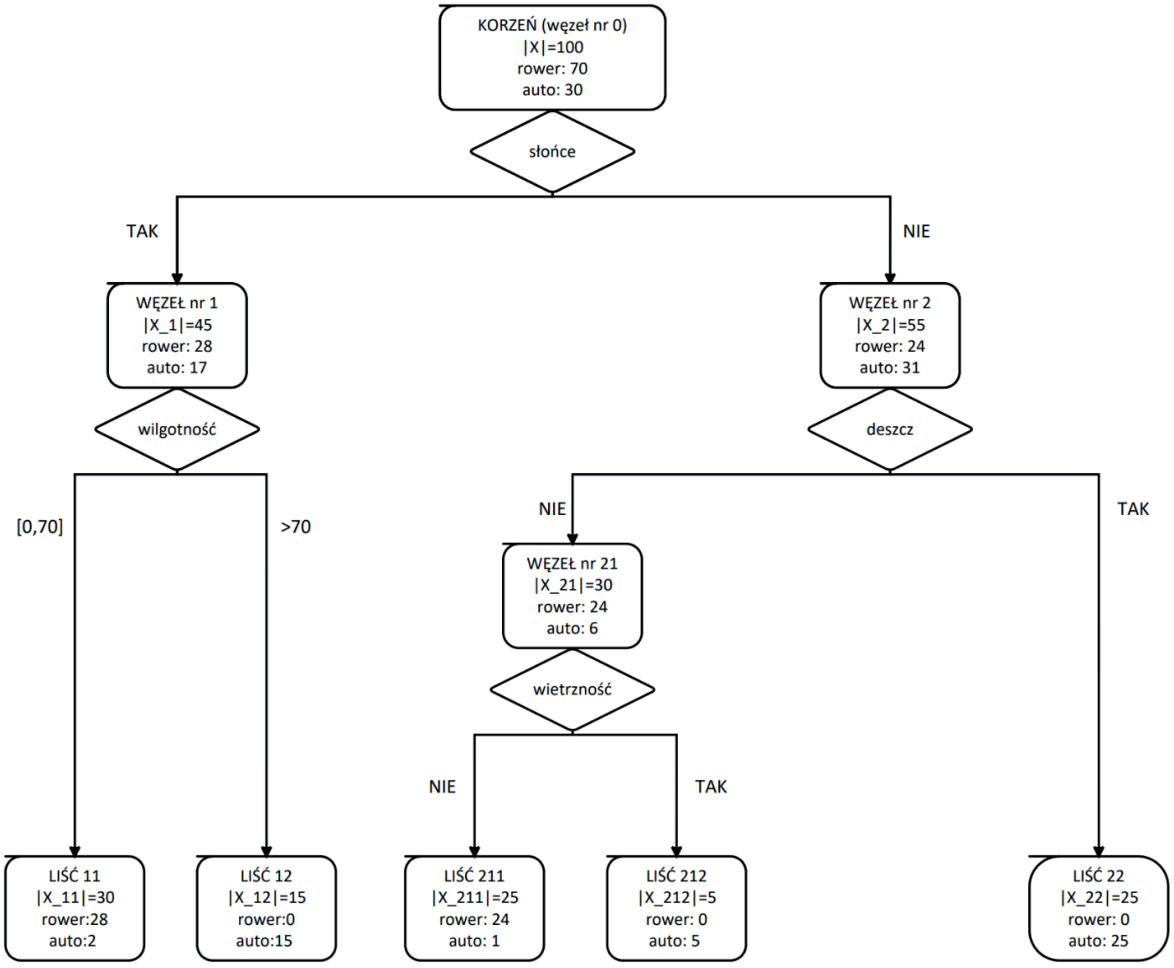
kategorii – rower albo auto, co odpowiada na pytanie, jaki środek transportu wybierze dany obiekt. W zbiorze treningowym mamy 100 obiektów. Na drzewie przedstawiono liczbę obiektów w kolejnych węzłach decyzyjnych w zależności od wartości atrybutu decyzyjnego.

Zauważmy, że zgodnie z kolejnymi podziałami $X = X\_1 + X\_2$, $X\_1 = X\_11 + X\_12$ oraz $X\_2 = X\_21 + X\_22$. Ponadto X_11 jest dzieckiem X_1 oraz potomkiem X.



Rysunek 16.1. Przykładowe niebinarne drzewo klasyfikacyjne

Zaproponowana budowa drzewa pozwala na podziały zarówno binarne, jak i wieloklasowe. Jest to zależne wyłącznie od konstrukcji kryterium podziału i tego, ile klas ma zmienna niezależna użyta w danym kryterium. Z drugiej strony każde drzewo niebinarne, czyli takie, w którym istnieje rodzic mający więcej niż dwoje dzieci, można transformować do drzewa o postaci binarnej. Dokonuje się tego przez użycie w węzłach odpowiednich sum warunków. W powyższym przykładzie pierwsze kryterium „zachmurzenie" można byłoby rozbić na dwa binarne kryteria: pierwsze „czy świeci słońce?", drugie „czy pada deszcz?". Na rysunku 16.2 przedstawiono drzewo binarne będące przykładową transformacją wieloklasowego drzewa zobrazowanego na rysunku 16.1.



Rysunek 16.2. Przykładowe drzewo klasyfikacyjne transformowane z postaci niebinarnej na binarną

Kroki algorytmu (trenowanie drzewa klasyfikacyjnego)

1. Ustalenie warunków stopu, czyli etapu, na którym otrzymane węzły zostaną uznane za liście, np.:
   – wszystkie przykłady w danym węźle należą do tej samej klasy (mają tę samą wartość atrybutu decyzyjnego),
   – nie ma atrybutu, który może dalej dzielić zbiór przykładów (wtedy gdy obiekty pochodzą z różnych klas atrybutu decyzyjnego, liść etykietowany jest wektorem wartości zwanym rozkładem decyzji),
   – wszystkie liście osiągnęły z góry założoną przewagę jednej klasy atrybutu decyzyjnego (np. maksymalnie 1% obiektów z innych klas niż dominująca).

2. Początek drzewa – ustanowienie korzenia, czyli pierwszego węzła decyzyjnego reprezentującego cały zbiór treningowy.
3. Wybór kryterium podziału węzła decyzyjnego na podstawie wybranej miary (np. współczynnik Giniego lub miara entropii i zysk informacyjny – opisane w kolejnych podrozdziałach).
4. Podział węzła decyzyjnego – w wyniku podziału powstaje rozgałęzienie i następuje jednoznaczny rozdział obiektów do nowych węzłów decyzyjnych.
5. Dla każdego z powstałych węzłów decyzyjnych sprawdzenie warunku stopu:
   - jeżeli dla danego węzła decyzyjnego nie jest spełniony warunek stopu, to powrót do kroku nr 3, czyli wybór nowego kryterium jego podziału,
   - jeżeli dla danego węzła decyzyjnego jest spełniony warunek stopu, to węzeł ten staje się liściem i nie następuje dalszy jego podział.
6. Rekurencyjne wykonywanie kroków 3–5 dla zbiorów obiektów powstałych we wcześniejszym kroku, aż osiągnięte zostanie kryterium stopu dla wszystkich węzłów decyzyjnych.

UWAGA

Opcjonalnie można użyć algorytmów optymalizujących tworzenie drzewa lub dokonać optymalizacji już utworzonego drzewa (np. przez przycięcie drzewa, co opisano w podrozdziale 16.1.3).

16.1.1. Współczynnik Giniego

Współczynnik Giniego jest jedną z podstawowych miar nierówności podziału, wykorzystywaną w celu znalezienia najlepszego kryterium podziału dla danego węzła decyzyjnego. Współczynnik Giniego przyjmuje wartości z przedziału [0, 1], przy czym wartość 0 oznacza, że wszystkie obiekty należą do tej samej klasy. Zatem wybrane będzie kryterium podziału z najwyższą wartością współczynnika Giniego.

W literaturze można znaleźć wiele równoważnych definicji współczynnika Giniego (Porath i Gilboa, 1994; Thon, 1982; Weymark, 1981). W tej pracy przyjęto następującą postać współczynnika podziału Giniego:

$$g = \sum_{j} P(S_j)\left(1 - \sum_{i=1}^{p} (P(d_i \mid S_j))^2\right), \tag{16.2}$$

gdzie:

d_i – kolejne wartości atrybutu decyzyjnego, $i = 1, ..., p$,
S_j – zbiory wyznaczone przez wybrane kryterium podziału.

Na potrzeby zastosowań prezentowanych w tym rozdziale wystarczające będą uproszczone wzory stosowane w binarnych drzewach klasyfikacyjnych (każdy rodzic ma maksymalnie dwoje dzieci, $j = 1, 2$) użytych w celu klasyfikacji binarnej (liczba klas atrybutu decyzyjnego $p = 2$). Wtedy:

$$gini(S_j) = 1 - \sum_{i=1}^{p} \left(\frac{m_i^{(j)}}{|S_j|} \right)^2, \tag{16.3}$$

gdzie:

$gini(S_j)$ – **współczynnik Giniego dla *j*-tego zbioru**, który jest wyznaczony przez wybrane kryterium podziału,
p – liczba klas atrybutu decyzyjnego, $p = 2$,
$|S_j|$ – moc zbioru S_j, $j = 1, 2$,
$m_i^{(j)}$ – liczba obiektów i-tej klasy w zbiorze S_j.

Na podstawie współczynników Giniego obliczonych dla każdego zbioru wyznaczonego przez badany podział, można obliczyć współczynnik podziału Giniego. Przyjmując jak wcześniej $p = 2$ oraz $j = 1, 2$, otrzymujemy:

$$g = \frac{|S_1|}{|S_1|+|S_2|} gini(S_1) + \frac{|S_2|}{|S_1|+|S_2|} gini(S_2) = \frac{m_1^{(1)} + m_2^{(1)}}{m_1 + m_2} gini(S_1) + \frac{m_1^{(2)} + m_2^{(2)}}{m_1 + m_2} gini(S_2), \tag{16.4}$$

gdzie:

g – **współczynnik podziału Giniego**,
$|S_j|$ – moc zbioru S_j, $j = 1, 2$,
$m_1^{(j)}, m_2^{(j)}$ – liczba obiektów odpowiednio klasy nr 1 i 2 w zbiorze S_j,
m_1, m_2 – liczba wszystkich obiektów odpowiednio klasy nr 1 i 2.

Zgodnie z powyższymi wzorami, aby ocenić podział wyznaczony przez wybrane kryterium podziału, najpierw należy obliczyć współczynnik Giniego zbioru $gini(S_j)$ dla każdego ze zbiorów powstających w wyniku tego podziału. Następnie na podstawie indywidualnych współczynników Giniego zbiorów oblicza się współczynnik podziału Giniego g. Procedurę tę trzeba powtórzyć dla wielu kryteriów podziału względem różnych zmiennych niezależnych, a następnie wybrać to kryterium, które daje najniższą wartość współczynnika g.

16.1.2. Entropia i zysk informacyjny

Entropia i zysk informacyjny to pojęcia, które można zastosować zamiennie ze współczynnikiem podziału Giniego. W tym podejściu należy najpierw wyznaczyć współczynnik entropii H dla atrybutu decyzyjnego, następnie entropię podziału zbioru ze względu na wybrany atrybut warunkowy, a na końcu zysk informacyjny, który jest różnicą tych dwóch wartości. W wyborze kryterium podziału należy się kierować się maksymalizacją zysku informacji. Niech $d_1, ..., d_p$ to kolejne wartości atrybutu decyzyjnego, natomiast S_j, $j = 1, ..., r$ to zbiory, które są wynikiem podziału względem badanego kryterium. Liczymy kolejno:

1) współczynnik entropii dla atrybutu decyzyjnego:

$$H = -\sum_{i=1}^{p} P(d_i) \cdot \log_2(P(d_i)), \tag{16.5}$$

2) współczynnik entropii dla atrybutu decyzyjnego w zbiorze S_j:

$$H_j = -\sum_{i=1}^{p} P(d_i \mid S_j) \cdot \log_2(P(d_i \mid S_j)), \tag{16.6}$$

3) entropię podziału zbioru ze względu na wybrane kryterium:

$$E = \sum_{j=1}^{r} \left(\frac{|S_j|}{\sum_{j=1}^{r} |S_j|} \cdot H_j \right), \tag{16.7}$$

4) zysk informacyjny

$$Gain = H - E. \tag{16.8}$$

16.1.3. Przycinanie drzew klasyfikacyjnych

W przycinaniu drzew stosowane są dwa podejścia:

1) przycinanie wsteczne – wygenerowanie drzewa bardzo dobrze dopasowanego do danych i usuwanie od dołu najmniej efektywnych węzłów,
2) przycinanie w przód – wstrzymanie dalszej rozbudowy gałęzi, jeśli na węźle znajduje się liczba próbek jednej klasy przekraczająca przyjęty próg.

Kroki algorytmu (przycinanie drzew klasyfikacyjnych)

1. Ustalenie kryterium stopu, np. brak dalszej poprawy, osiągnięcie maksymalnej liczby iteracji.
2. Wybranie i usunięcie elementu drzewa podejrzanego o wprowadzenie najmniej istotnego podziału.
3. Przetestowanie przyciętego drzewa na zbiorze testowym.
4. Sprawdzenie, czy przyjęta miara sukcesu wskazuje na poprawę klasyfikacji:
   - jeżeli tak, wybrany w kroku nr 2 element drzewa usuwany jest na stałe.
   - jeżeli nie, wybrany w kroku nr 2 element drzewa jest przywrócony.
5. Jeżeli warunek stopu nie jest spełniony, to powrót do kroku nr 2.

Problemy związane z algorytmem drzew klasyfikacyjnych:

– algorytm jest zachłanny, czyli drzewa są budowane przez wybieranie optymalnego podziału w danym kroku, co może prowadzić do otrzymania suboptymalnych rozwiązań globalnych,
– miary stosowane w wyborze kryterium prowadzą do tego, że nie uwzględnia się zmiennych o mniejszej sile predykcyjnej,
– nie uwzględnia się zmiennych o dużej sile predykcyjnej, które są skorelowane (podziały względem zmiennych, które są skorelowane będą zbliżone),
– istnieje duże ryzyko przetrenowania modelu, którego objawem jest zbyt wiele poddrzew (mały poziom generalizacji modelu, model zapamiętał jedynie przypadki treningowe).

W celu uniknięcia problemu przeuczenia klasyfikatora stosuje się m.in. opisany powyżej algorytm przycinania drzew klasyfikacyjnych, który równocześnie polepsza wydajność. Przeciwdziałaniu wyszczególnionym powyżej problemom służy również zastosowanie procedur agregujących rodziny klasyfikatorów, do których należą m.in. lasy losowe.

16.2. Lasy losowe

Lasy losowe są jedną z metod agregacji klasyfikatorów. Las losowy to grupa drzew klasyfikacyjnych pracujących równolegle, na podstawie których podejmowana jest ostateczna, wspólna decyzja klasyfikacyjna. Zatem do sklasyfikowania nowego obiektu konieczne jest poddanie go niezależnej klasyfikacji każdego drzewa w lesie losowym, a następnie jako ostateczny wynik predykcji przyjmuje się tę wartość atrybutu decyzyjnego, która została przydzielona najczęściej. Proces ten nazywa się głosowaniem większościowym drzew wchodzących w skład lasu. W celu zminimalizowania korelacji między drzewami podczas budowy każdego drzewa dokonuje się losowego wyboru atrybutów warunkowych, które zostaną wzięte pod uwagę w danym drzewie (Breiman, 2001).

Kroki algorytmu (budowanie lasu losowego)

1. Ustalenie L – liczby drzew.
2. Losowanie **ze zwracaniem** N obserwacji z N-elementowego zbioru obiektów przy założeniu, że prawdopodobieństwo wylosowania dla każdej próbki jest takie samo i wynosi $1/N$. Wylosowane obiekty trafiają do zbioru treningowego. Obiekty, które nie znalazły się w zbiorze treningowym, trafiają do zbioru testowego.
3. Losowanie(*) bez zwracania ustalonej liczby zmiennych niezależnych (atrybutów warunkowych).
4. Zbudowanie drzewa, w którym korzeniem jest zbiór wylosowany w kroku 2 i poszukiwane są najlepsze podziały dla zmiennych wylosowanych w kroku nr 3.
5. Obliczenie błędu klasyfikacji.
6. Sprawdzenie, czy zadana liczba L drzew została osiągnięta:
   - jeżeli nie, powrót do kroku nr 2 (rozpoczęcie budowy nowego drzewa),
   - jeżeli tak, obliczenie błędu klasyfikacji lasu losowego jako średniej arytmetycznej błędów klasyfikacji wszystkich L drzew.

(*) Przez losowy wybór atrybutów warunkowych zmniejszamy korelację drzew.

Przykład 16.1. Klasyfikacja zdarzeń akustycznych

Dane, na których zostanie pokazane działanie algorytmu lasu losowego, dotyczą tzw. klasyfikacji zdarzeń akustycznych (ang. *acoustic event classification*). Są to różnego rodzaju dźwięki, które można usłyszeć w mieście – głosy ludzi, zwierząt, odgłosy urządzeń technicznych, samochodów itp. Jest to dość trudny problem klasyfikacyjny, ponieważ model uczony jest rozpoznawania wielu klas równocześnie. Dane pochodzą ze strony: https://www.kaggle.com/chrisfilo/urbansound8k.

Klasyfikacja zdarzeń akustycznych wykonana zostanie na podstawie współczynników melowo-cepstralnych (MFCC), które opisano w rozdziale 7. Ekstrakcja cech została przeprowadzona przy użyciu poniższego kodu.

```
from librosa.feature import mfcc as extract_mfcc
from librosa.feature import delta as extract_mfcc_deltas
import numpy as np
import pandas as pd
import os
from pathlib import Path

current_dir_path = Path.cwd()
data_info_col_list = [0,5,6]
#kolumny z nazwą pliku, numerem podzbioru i klasą

data_info_df = pd.read_csv("UrbanSound8K.csv", usecols = data_info_col_list,
                                                low_memory = False)
labels = []
fold = []
mfccs = []

for i in data_info_df.index:

  fold_id = data_info_df.at[i, 'fold']
  fold_folder = 'fold' + str(fold_id)
  signal, fs = librosa.load(current_dir_path/fold_folder/data_info_df.at[
                                                        i, 'slice_file_name'],
                            sr = 16000,
                            mono = True)
  mfcc = extract_mfcc(signal, sr = fs, n_mfcc = 13)
  mfccs.append(mfcc)
  labels.append(data_info_df.at[i, 'classID'])
  fold.append(fold_id)

frames_num_mfcc = [i.shape[0] for i in mfccs]

cut_mfcc_feats = []
#dalszym analizom zostaną poddane wyłącznie pliki o długości nie większej
#niż 345 ramek - wartość ta została dobrana na podstawie analizy histogramu
#długości nagrań zawartych w bazie, sygnały zbyt długie zostaną przycięte
```

```
#liczba ramek została dobrana na podstawie analizy histogramu długości
#sygnałów zawartych w bazie danych
for feats, frames in zip(mfccs, frames_num_mfcc):
  if frames >= 345:
    cut_mfcc_feats.append(feats[:345,:])
cut_mfcc_feats = np.asarray(cut_mfcc_feats)

#labels, folds
labels = np.asarray(labels)
labels = labels[(frames_num) >= 345]
fold = np.asarray(fold)
fold = fold[(frames_num) >= 345]

#saving files
np.save('labels', labels)
np.save('folds', fold)
np.save('mfcc_feats', cut_mfcc_feats)
```

ZASADA

Większość algorytmów uczenia maszynowego wymaga wyznaczenia takiej samej liczby cech dla każdego obiektu. W przypadku sygnałów o różnej długości konieczne jest więc przycięcie zbyt długich sygnałów lub wydłużenie zbyt krótkich sygnałów przez uzupełnienie ich zerami (tzw. zero-padding) lub innymi wartościami, np. wartością średnią. W Pythonie operację uzupełnienia wektora danych można zrealizować, używając funkcji pad z biblioteki numpy znajdującej się pod adresem: https://numpy.org/doc/stable/reference/generated/numpy.pad.html.

Uzupełnienie sygnału można przeprowadzić zarówno na przebiegu czasowym sygnału, jeszcze przed ekstrakcją cech, jak i na wektorze lub macierzy wyekstrahowanych cech.

Przed przystąpieniem do klasyfikacji sygnałów dane należy odpowiednio przygotować. Na początek wczytane zostaną trzy pliki wygenerowane przedstawionym powyżej kodem:

1) mfcc_feats.npy – macierz zawierająca 13 współczynników MFCC wyznaczonych dla każdego sygnału,
2) labels.npy – wektor etykiet,
3) folds.npy – podział na podzbiory, który ma być wykorzystany podczas sprawdzianu krzyżowego.

W poprzednich rozdziałach sprawdzian krzyżowy wykonywany był z użyciem danych podzielonych na losowe podzbiory, jednak w niektórych sytuacjach przyporządkowanie danych do podzbiorów ma duże znaczenie i ich podział powinien być przeprowadzony w określony sposób. Jest tak np. wtedy, gdy w bazie znajduje się więcej niż jeden sygnał zawierający dźwięki emitowane przez to samo urządzenie lub tę samą osobę. Wtedy należy zagwarantować, że wszystkie sygnały związane z tym samym obiektem znajdują się w tym samym podzbiorze – gdyby tak nie było, to uzyskane wartości metryk sukcesu byłyby zawyżone i nie byłoby możliwe prawidłowe ocenienie zdolności generalizacyjnych modelu.

```
import numpy as np

X_mfcc = np.load('mfcc_feats.npy')
y_org = np.load('labels.npy')
folds = np.load('folds.npy')
```

```
#klasy:
#0 = airconditioner, 1 = carhorn, 2 = childrenplaying,
#3 = dogbark, 4 = drilling, 5 = engineidling, 6 = gunshot,
#7 = jackhammer, 8 = siren, 9 = street_music

print(X_mfcc.shape)
#wyświetlenie wymiarów macierzy cech - zbiór danych tworzony jest przez 7334
#sygnały o długości 345 ramek, z każdego sygnału wyekstrahowano 13 MFCC
```

Macierz cech jest trójwymiarowa, więc przed przystąpieniem do uczenia modelu należałoby każdą dwuwymiarową macierz cech opisujących pojedynczy obiekt zamienić na wektor. Sprawiłoby to jednak, że każdy sygnał byłby opisany 4486 cechami – algorytmy drzewiaste uczyłyby się na takich danych bardzo długo, dlatego wymiarowość danych powinna zostać zredukowana. Jednak zamiast prowadzić redukcję wymiarowości przy użyciu algorytmów takich jak analiza składowych głównych (podrozdz. 9.1), analiza składowych niezależnych (podrozdz. 9.2) czy metody selekcji cech, np. SelectKBest pokazana w Zadaniu 12.2, wyliczone zostaną parametry statystyczne charakteryzujące każde pasmo częstotliwości (każdy z 13 współczynników cepstralnych) na całej długości sygnału. Będą to bardzo podstawowe parametry:

- wartość średnia,
- odchylenie standardowe,
- mediana,
- I i III kwartyl,
- rozrzut pomiędzy 10 i 90 percentylem,
- kurtoza,
- skośność,
- wartość minimalna,
- wartość maksymalna.

W wyniku przeprowadzenia opisanej parametryzacji każdy sygnał będzie opisany tylko 130 cechami, więc można prowadzić procesy uczenia, walidacji i predykcji bez dodatkowej redukcji wymiarowości.

```
import scipy.stats

X = np.hstack((np.mean(X_mfcc, axis = 2),
       np.std(X_mfcc, axis = 2),
       np.median(X_mfcc, axis = 2),
       np.percentile(X_mfcc, 25, axis = 2),
       np.percentile(X_mfcc, 75, axis = 2),
       scipy.stats.iqr(X_mfcc, rng = (10, 90), axis = 2),
       scipy.stats.kurtosis(X_mfcc, axis = 2),
       scipy.stats.skew(X_mfcc, axis = 2),
       np.min(X_mfcc, axis = 2),
       np.max(X_mfcc, axis = 2)
       ))
```

Następnie sprawdzona zostanie liczba obiektów przynależących do poszczególnych klas. Jeżeli różnica pomiędzy liczebnością klas będzie bardzo duża, to prawdopodobnie klasyfikator nie będzie w stanie nauczyć się ich poprawnie rozpoznawać bez dodatkowego przetworzenia danych, np. nadpróbkowania lub podpróbkowania zbiorów danych, które można przeprowadzić z wykorzystaniem biblioteki imblearn znajdującej się pod adresem: https://imbalanced-learn.org.

```
from collections import Counter
print(Counter(y_org))
```

Najmniej liczna jest klasa 6 – należy do niej tylko 16 obiektów, czyli 0,2% wszystkich danych. Obiekty należące do tej klasy zostaną usunięte ze zbioru danych, ponieważ będą tylko utrudniać proces uczenia, a omawiane w tym rozdziale klasyfikatory nie będą w stanie nauczyć się klasyfikować ich poprawnie.

Podzbiór nr 10 zostanie wykorzystany jako zbiór testowy, a pozostałe dziewięć podzbiorów będzie tworzyło zbiór uczący – sprawdzian krzyżowy stosowany podczas optymalizacji hiperparametrów modelu będzie więc dziewięciokrotny. Przygotowane w ten sposób dane zostaną poddane standaryzacji.

```
from sklearn.preprocessing import StandardScaler

#usuwanie klasy 6 ze zbioru danych
y_not_6 = y_org != 6

X = X[y_not_6]
folds = folds[y_not_6]
y = y_org[y_not_6]

#podział na zbiory uczący i testowy
train_folds_mask = folds != 10
test_fold_mask = folds == 10

train_folds = folds[train_folds_mask]
test_fold = folds[test_fold_mask]

X_train = X[train_folds_mask]
X_test = X[test_fold_mask]

y_train = y[train_folds_mask]
y_test = y[test_fold_mask]

#standaryzacja danych
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

UWAGA

Jeżeli podział danych na podzbiory wykorzystywane podczas sprawdzianu krzyżowego jest ściśle określony, do funkcji cross_validate powinny zostać podane następujące argumenty:

- model,
- macierz cech uczących oraz wektor etykiet zbioru uczącego,
- metryka, która będzie wykorzystywana do ewaluacji modelu (scoring),
- parametr groups – jest to podział obiektów na podzbiory określony w zmiennej train_folds (dla każdego obiektu określony jest numer podzbioru, w którym ma się on znaleźć),
- parametr cv=GroupKFold(n_splits=len(np.unique(train_folds))) – na jego podstawie algorytm sprawdzianu krzyżowego określa, ile podzbiorów ma zostać utworzonych, i dostaje informację, że ma ten podział przeprowadzić na podstawie przyporządkowania do podzbiorów określonego w zmiennej train_folds.

```
from sklearn.metrics import f1_score, make_scorer
from sklearn.model_selection import cross_validate, GroupKFold
from sklearn.ensemble import RandomForestClassifier

scoring = {'f1_macro': make_scorer(f1_score, average = 'macro')}
model = RandomForestClassifier(random_state = 42)
scores = cross_validate(model, X_train,
                               y_train,
                               scoring = scoring,
                               groups = train_folds,
                               cv = GroupKFold(
                                 n_splits = len(np.unique(train_folds))))
```

W zmiennej scores wyznaczonej przez sprawdzian krzyżowy znajdują się wartości metryki F1 otrzymane dla każdego podzbioru walidacyjnego. Analizując te wartości, można m.in. sprawdzić, czy wyniki uzyskiwane przez model są stabilne.

ZASADA

W idealnym przypadku odchylenie standardowe wyników uzyskanych na wszystkich podzbiorach jest bliskie zeru, a wartość średnia jest bliska 1. Jeżeli dla jednego podzbioru walidacyjnego otrzymane wyniki są znacząco gorsze niż dla pozostałych, to oznacza, że prawdopodobnie znajdujące się w nim dane istotnie różnią się od pozostałych danych. Jeżeli odchylenie standardowe wyników uzyskanych dla poszczególnych podzbiorów jest duże, to świadczy to o złym uogólnieniu modelu.

Jak widać, wyniki nie są szczególnie dobre – dla ośmiu z dziewięciu podzbiorów otrzymano F1 < 0,6. Jeżeli przeprowadzona zostanie walidacja prosta (czyli uczenie na całym zbiorze uczącym i predykcje na zbiorze testowym), to również można spodziewać się wyników, które nie będą zadowalające.

```
from sklearn.metrics import accuracy_score, confusion_matrix

#trening modelu o domyślnych wartościahc hiperparametrów
model.fit(X_train, y_train)
preds = model.predict(X_test)
print('test accuracy = ', accuracy_score(y_test, preds))
print('test F1 = ', f1_score(y_test, preds, average = 'weighted'))
print(confusion_matrix(y_test, preds))
```

Przypuszczenia oparte na wynikach dla podzbiorów walidacyjnych potwierdziły się i wyniki uzyskane na zbiorze testowym nie są zbyt dobre, chociaż nie można ich też uznać za bardzo złe – dokładność klasyfikacji wynosząca 0,62 dla modelu z domyślnymi wartościami hiperparametrów i przy dziewięciu klasach to całkiem dobry wynik. Widać natomiast, że np. żaden obiekt z klasy 2 nie został zaklasyfikowany prawidłowo, więc warto spróbować otrzymać lepszy model przez zmianę użytych cech sygnału oraz optymalizację hiperparametrów modelu.

Przykład 16.2. Klasyfikacja zdarzeń akustycznych – analiza wpływu metody parametryzacji sygnału na wyniki klasyfikacji

W tym przykładzie będzie kontynuowana analiza danych przedstawionych w Przykładzie 16.1. Na początek wyznaczone zostaną dodatkowe cechy sygnałów – będą to współczynniki GFCC opisane w rozdziale 7.

Do ich wyznaczenia posłuży poniższy kod.

```
import numpy as np
from spafe.features.gfcc import gfcc
import pandas as pd
import os
from pathlib import Path
current_dir_path = Path.cwd()
data_info_col_list = [0,5,6]
#kolumny z nazwą pliku, numerem podzbioru i klasą

data_info_df = pd.read_csv("UrbanSound8K.csv",
                           usecols = data_info_col_list,
                           low_memory = False)
gfccs = []
for i in data_info_df.index:

  fold_id = data_info_df.at[i, 'fold']
  fold_folder = 'fold'+str(fold_id)
  signal, fs = librosa.load(
      current_dir_path/fold_folder/data_info_df.at[i, 'slice_file_name'],
      sr = 16000,
      mono = True)
  gfc = gfcc(signal, fs=fs, num_ceps = 13)
  gfccs.append(gfc)

frames_num_gfcc = [i.shape[0] for i in gfccs]

cut_gfcc_feats = []
for feats, frames in zip(gfccs,frames_num_gfcc):
  if frames >= 345:
    cut_gfcc_feats.append(feats[:345,:])
cut_gfcc_feats = np.asarray(cut_gfcc_feats)
cut_gfcc_feats = cut_feats.reshape(len(cut_gfcc_feats),-1)
np.save('gfcc_feats', cut_gfcc_feats)
```

Przygotowanie danych do dalszej analizy zostanie przeprowadzone w sposób analogiczny do przedstawionego w Przykładzie 16.1 – na podstawie danych zostaną wyliczone parametry statystyczne, następnie dane zostaną podzielone na zbiór uczący i testowy oraz poddane standaryzacji, a obiekty należące do klasy 6. zostaną usunięte. Tak przygotowane dane posłużą do treningu i ewaluacji modelu.

```
X_gfcc = np.load('gfcc_feats.npy')

X = np.hstack((np.mean(X_gfcc, axis = 2),
       np.std(X_gfcc, axis = 2),
       np.median(X_gfcc, axis = 2),
```

```
        np.percentile(X_gfcc, 25, axis = 2),
        np.percentile(X_gfcc, 75, axis = 2),
        scipy.stats.iqr(X_gfcc, rng = (10, 90), axis = 2),
        scipy.stats.kurtosis(X_gfcc, axis = 2),
        scipy.stats.skew(X_gfcc, axis = 2),
        np.min(X_gfcc, axis = 2),
        np.max(X_gfcc, axis = 2)
        ))

X = X[y_not_6]

X_train = X[train_folds_mask]
X_test = X[test_fold_mask]

model = RandomForestClassifier(random_state = 42)
model.fit(X_train, y_train)
preds = model.predict(X_test)
print('test accuracy = ', accuracy_score(y_test, preds))
print('test F1 = ', f1_score(y_test, preds, average = 'weighted'))
print(confusion_matrix(y_test, preds))
```

Wyniki otrzymane na GFCC są znacząco gorsze niż te uzyskane na MFCC – dokładność klasyfikacji wynosi zaledwie 0,53, a F1 wynosi 0,52.

W kolejnym kroku model zostanie poddany treningowi z równoczesnym wykorzystaniem obu rodzajów współczynników cepstralnych. Takie postępowanie, czyli wykorzystywanie zbioru różnych cech, które pojedynczo nie dają dobrych rezultatów, często pozwala znacząco poprawić jakość modelu.

```
X = np.hstack((X_mfcc, X_gfcc))
X = np.hstack((np.mean(X, axis = 2),
        np.std(X, axis = 2),
        np.median(X, axis = 2),
        np.percentile(X, 25, axis = 2),
        np.percentile(X, 75, axis = 2),
        scipy.stats.iqr(X, rng = (10, 90), axis = 2),
        scipy.stats.kurtosis(X, axis = 2),
        scipy.stats.skew(X, axis = 2),
        np.min(X, axis = 2),
        np.max(X, axis = 2)
        ))
```

Co prawda pojedynczy obiekt jest teraz opisywany przez aż 260 cech, jednak nadal nie jest to na tyle duża liczba, żeby proces uczenia trwał bardzo długo lub było duże ryzyko przeuczenia modelu, więc można nie przeprowadzać dodatkowej redukcji wymiarowości.

```
model.fit(X_train, y_train)
preds = model.predict(X_test)
```

```
print('test accuracy = ', accuracy_score(y_test, preds))
print('test F1 = ', f1_score(y_test, preds, average = 'weighted'))
print(confusion_matrix(y_test, preds))
```

Wyniki, które otrzymano na MFCC i GFCC równocześnie, są już bardziej obiecujące – nawet na nieoptymalizowanym modelu udało się uzyskać dokładność i F1 przekraczające 0,65. Biorąc pod uwagę liczbę klas występujących w zbiorze danych, duże różnice w ich liczebności oraz rodzaj klasyfikowanych sygnałów, taki wynik trzeba uznać za całkiem dobry. W celu jego poprawy należy przeprowadzić optymalizację hiperparametrów, co będzie celem Zadania 16.1.

Przykład 16.3. Przewidywanie miejsca drużyny w meczu na podstawie statystyk graczy – inżynieria cech i jej wpływ na jakość predykcji modelu

Pod adresem: https://www.kaggle.com/datasets/skihikingkevin/pubg-match-deaths znajdują się dane zawierające statystyki graczy w meczach PUBG w trybie Battle Royale – mecz wygrywa ostatni zespół, którego członkowie żyją (a przynajmniej jeden z nich). Baza danych jest bardzo duża, więc analizom zostanie poddany tylko jeden z plików wchodzących w jej skład.

Statystyki graczy posłużą do określenia, które miejsce zajęły poszczególne osoby oraz ich drużyny. Zwycięstwo odnosi gracz (drużyna), który zajmuje miejsce 1. Dane, które zostaną wczytane do struktury DataFrame, są podzielone na 15 kolumn:

1) date – data meczu,
2) game_size – liczba graczy, którzy brali udział w rozgrywce,
3) match_id – unikalny identyfikator meczu,
4) match_mode – tryb meczu,
5) party_size – liczba osób tworzących drużynę, w której znajdował się dany gracz,
6) player_assists – liczba asyst danego gracza,
7) player_dbno (down but not out) – zmienna określająca, czy gracz został powalony, ale przywrócony do gry,
8) player_dist_ride – dystans przejechany przez gracza,
9) player_dist_walk – dystans, który gracz przeszedł,
10) player_dmg – obrażenia zadane przeciwnikowi,
11) player_kills – liczba zabitych przez gracza przeciwników,
12) player_name – nazwa gracza,
13) player_survive_time – czas przeżyty przez gracza,
14) team_id – unikalny identyfikator drużyny,
15) team_placement – miejsce zajęte przez drużynę.

Sprawdzając, jakie tryby meczów występują w analizowanych danych, można stwierdzić, że są to wyłącznie mecze w trybie TTP, czyli takie, w których drużyna składa się z nie więcej niż dwóch graczy.

```
import pandas as pd

df = pd.read_csv('./agg_match_stats_0.csv')
print(df['match_mode'].unique())
print(df['match_id'].nunique())
```

Danych nadal jest bardzo dużo – statystyki spełniające zadane wcześniej kryteria dotyczą niemal 150 000 meczów. W celu zmniejszenia czasu potrzebnego na trening modelu w dalszej części przykładu wykorzystane zostanie tylko 10 000 meczów.

```
df = df[df['match_id'].isin(df['match_id'].unique()[:10_000])]

#usunięcie kolumn, które nie mają wpływu na przewidywaną zmienną
drop_cols = ['player_name', 'match_mode', 'date']
df = df.drop(drop_cols, axis = 1)
```

W poszczególnych meczach brała udział różna liczba graczy i drużyn, więc w celu lepszego zobrazowania rankingu miejsc zajmowanych przez poszczególnych uczestników zmienną zawierającą te miejsca poddano normalizacji.

```
df['normalized_team_placement'] = df['team_placement'] / df['game_size']
target = ['normalized_team_placement']
```

Po wstępnym przygotowaniu danych można przejść do ich podziału na zbiory uczący i testowy. Zbiór testowy będzie stanowił 10% wszystkich analizowanych meczów. Nie wszystkie dane powinny być poddane analizie – identyfikatory meczu i drużyny nie mają żadnego wpływu na miejsce zajmowane przez drużynę. Liczba graczy wchodzących w skład drużyny również nie ma wpływu na to, które miejsce zajął konkretny uczestnik. Zmienna game_size posłużyła do normalizacji zmiennej celu, więc ją również można wykluczyć z dalszych etapów analizy.

```
from sklearn.model_selection import train_test_split

train_match_ids, test_match_ids = train_test_split(
                                                df['match_id'].unique(),
                                                test_size = 0.1,
                                                random_state = 42)

not_features = ['match_id', 'party_size', 'team_id','game_size',
                                              'team_placement'] + target
feature_columns = list(filter(lambda x: x not in not_features, df.columns))

df_train = df[df['match_id'].isin(train_match_ids)]
df_test = df[df['match_id'].isin(test_match_ids)]

X_train = df_train[feature_columns]
y_train = df_train[target]
X_test = df_test[feature_columns]
y_test = df_test[target]
```

Analiza danych zostanie rozpoczęta od treningu regresyjnego modelu lasu losowego, który posłuży do określenia miejsca zajętego przez danego gracza na podstawie statystyk opisanych powyżej. Zastosowany zostanie model z domyślnymi wartościami hiperparametrów.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error
```

```
baseline_model = RandomForestRegressor(n_jobs = 10)
baseline_model.fit(X_train, y_train)
test_predictions = baseline_model.predict(X_test)
mean_absolute_error(y_test, test_predictions)
```

Średni błąd absolutny predykcji wynosi około 0,1. Jest to całkiem dobry wynik – model myli się średnio tylko o 0,1 miejsca znormalizowanego (czyli o około 10% miejsc możliwych do zajęcia w danym meczu).

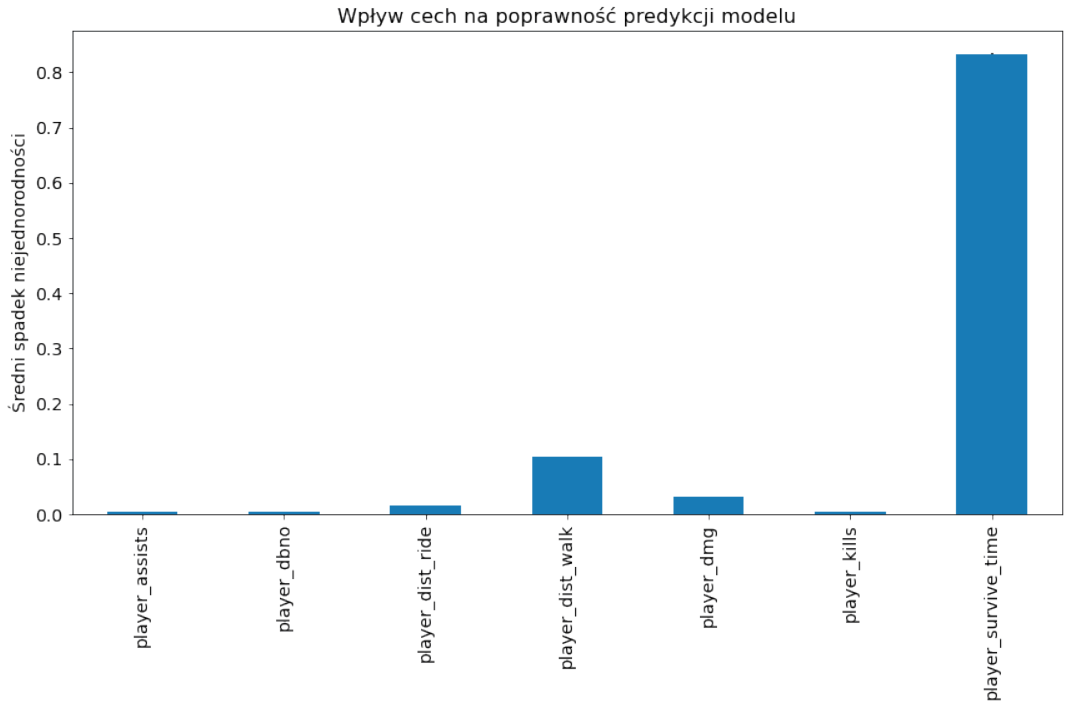
Następnie wyznaczona zostanie dokładność predykcji miejsca zajętego przez całą drużynę – zwycięska drużyna to ta, której członek zajął miejsce pierwsze, nawet jeżeli jej pozostali członkowie wcześniej zginęli. Podobnie wyznacza się pozostałe miejsca w rankingu drużyn – miejsce zajmowane przez całą drużynę wyznaczane jest na podstawie miejsca zajętego przez członka zespołu, który przeżył najdłużej.

```
import numpy as np
from sklearn.metrics import accuracy_score

test_df_with_preds = pd.concat([df_test.reset_index(drop = True),
                                                pd.Series(test_predictions,
                                                name = 'prediction')],
                                axis = 1)

grouped_team_results = test_df_with_preds.groupby(
  ['match_id', 'team_id']).agg(np.mean).groupby(
  'match_id').agg(np.argmin)

accuracy_score(grouped_team_results['normalized_team_placement'],
               grouped_team_results['prediction'])

#Uzyskana dokładność predykcji jest dość wysoka i wynosi 0,78.
```

Przydatną cechą lasów losowych jest możliwość sprawdzenia, które cechy miały największy wpływ na predykcję. Należy w tym celu użyć metody feature_importances_.

```
from matplotlib import pyplot as plt

importances = baseline_model.feature_importances_
std = np.std([tree.feature_importances_ for tree in
                            baseline_model.estimators_],axis = 0)

forest_importances = pd.Series(importances, index = feature_columns)

fig, ax = plt.subplots(figsize = (12,8))
forest_importances.plot.bar(yerr = std, ax = ax)
ax.set_title("Wpływ cech na poprawność predykcji modelu", fontsize = 16)
ax.set_ylabel("Średni spadek niejednorodności", fontsize = 14)
```

```
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
fig.tight_layout()
plt.show()
```



Rysunek 16.3. Wpływ analizowanych cech na predykcje modelu określony za pomocą metody feture_importances_. Im większa wartość, tym większy udział cechy w kryterium decyzyjnym modelu

Jak można zauważyć na rysunku 16.3, największe znaczenie miała zmienna player_survive_time, czyli czas przeżycia gracza. Jest to dosyć oczywiste – gracz, który zginął już na początku rozgrywki, nie może być zwycięzcą. Im dłużej gracz utrzymuje się w grze, tym większe są jego szanse na wygranie meczu.

Różnica pomiędzy wpływem zmiennej player_survive_time na predykcje a wpływem innych zmiennych jest bardzo duża. Można założyć, że analiza wyłącznie tej jednej zmiennej pozwoli uzyskać zbliżoną dokładność predykcji, więc analiza innych zmiennych byłaby pozbawiona sensu. Sytuacja, w której predykcje są wyznaczane na podstawie jednej zmiennej, nie jest jednak pożądana, ponieważ nie ma wówczas możliwości kompensacji ewentualnych błędów jej pomiaru lub występujących zakłóceń. Warto również zauważyć, że w przypadku analizowanych danych zmienna player_survive_time jest de facto równoważna zajmowanemu przez drużynę miejscu – im dłuższy czas przeżycia w grze, tym wyższe miejsce drużyny w rankingu.

Aby sprawdzić, czy model zwracałby satysfakcjonujące predykcje na podstawie wartości pozostałych zmiennych, usunięto z danych zmienną player_survive_time i powtórzono proces uczenia i ewaluacji modelu.

```
df = df.drop('player_survive_time', axis = 1)
feature_columns.remove('player_survive_time')

df_train = df[df['match_id'].isin(train_match_ids)]
df_test = df[df['match_id'].isin(test_match_ids)]
```

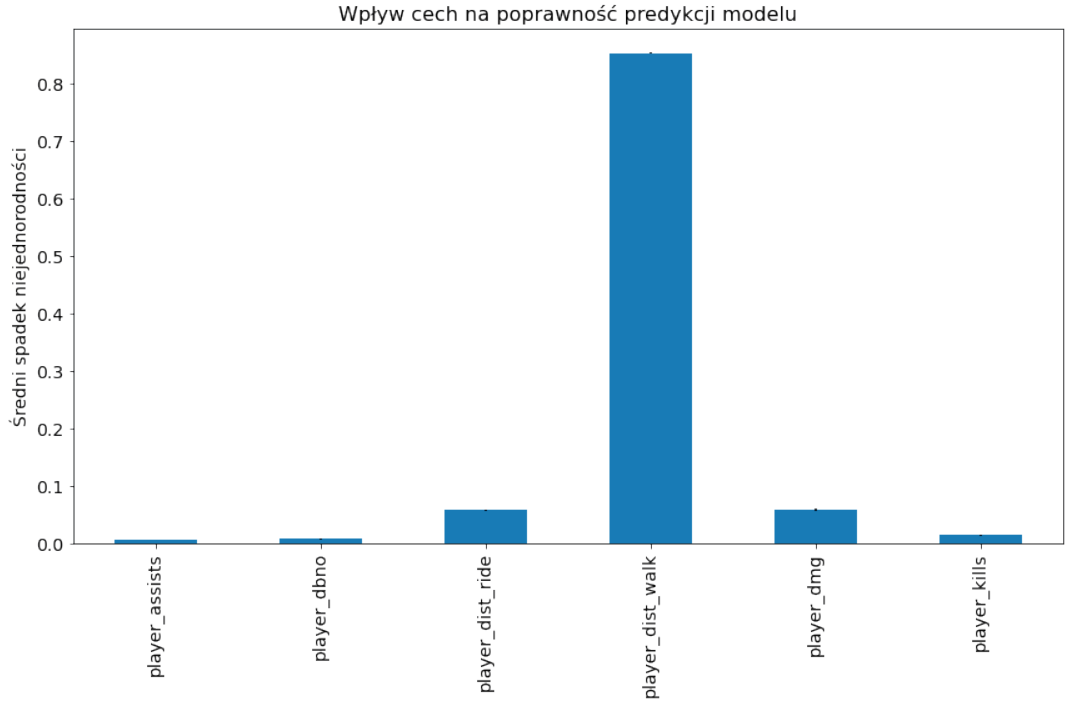
```
X_train = df_train[feature_columns]
y_train = df_train[target]
X_test = df_test[feature_columns]
y_test = df_test[target]

baseline_model.fit(X_train, y_train)

test_predictions = baseline_model.predict(X_test)
mean_absolute_error(y_test, test_predictions)
```

Otrzymany średni błąd bezwzględny predykcji miejsca zajmowanego przez gracza jest nieco większy niż wcześniej – wynosi około 0,13, podczas gdy poprzedni model uzyskał MAE wynoszące około 0,1.

```
test_df_with_preds = pd.concat([df_test.reset_index(drop = True),
                                pd.Series(test_predictions,
                                name = 'prediction')],
                               axis = 1)

grouped_team_results = test_df_with_preds.groupby(
                           ['match_id', 'team_id']).agg(np.mean).groupby(
                                'match_id').agg(np.argmin)

accuracy_score(grouped_team_results['normalized_team_placement'],
               grouped_team_results['prediction'])
```

Zdecydowanie gorzej wygląda natomiast zdolność modelu do przewidywania miejsca zajętego przez drużynę – po usunięciu zmiennej player_survive_time spadła z około 0,78 do zaledwie 0,39.

```
importances = baseline_model.feature_importances_
std = np.std([tree.feature_importances_ for tree in
                                        baseline_model.estimators_],
             axis = 0)

forest_importances = pd.Series(importances, index = feature_columns)

fig, ax = plt.subplots(figsize = (12,8))
forest_importances.plot.bar(yerr = std, ax = ax)
ax.set_title("Wpływ cech na poprawność predykcji modelu", fontsize = 16)
ax.set_ylabel("Średni spadek niejednorodności", fontsize = 14)
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
fig.tight_layout()
plt.show()
```



Rysunek 16.4. Wpływ analizowanych cech na predykcje modelu uczonego na danych niezawierających zmiennej player_survive_time

Tym razem największy wpływ na uzyskiwane predykcje miała zmienna player_dist_walk (rys. 16.4). Podobnie jak miało to miejsce w przypadku modelu uczonego na wszystkich danych, widać bardzo dużą różnicę pomiędzy ważnością cechy najważniejszej oraz ważnością pozostałych cech. Również tym razem predykcje są wyznaczane głównie na podstawie jednej zmiennej, jednak nie pozwala ona uzyskać zadowalających rezultatów.

Wyników można próbować poprawić przez zastosowanie tzw. inżynierii cech (ang. *feature engineering*), czyli ich odpowiednie przetworzenie. Podstawową metodą inżynierii cech jest wyznaczenie opisujących je parametrów statystycznych, takich jak wartość minimalna, maksymalna czy sumowanie odpowiednich rekordów tabeli.

Na podstawie dostępnych w bazie danych cech zostaną wyznaczone nowe cechy:

- total_dist – całkowity dystans przebyty przez gracza w dowolny sposób,
- kills_and_assists – liczba wrogów pokonanych przez gracza oraz asyst w pokonaniu wroga przez innego gracza,
- wartości średnia, maksymalna oraz sumaryczna statystyk wszystkich graczy wchodzących w skład drużyny,
- wartości średnia i maksymalna statystyk osiągniętych w danym meczu.

```
df['total_dist'] = df['player_dist_ride'] + df['player_dist_walk']
df['kills_and_assists'] = df['player_kills'] + df['player_assists']

feature_columns = feature_columns + ['total_dist','kills_and_assists']

team_aggregated = df.groupby(['match_id', 'team_id'], as_index = False)[
                  feature_columns].agg(['sum', 'max', 'mean'])
new_names = ['_'.join(['team'] + list(x)
                                    ) for x in team_aggregated.columns]
```

```
team_aggregated.columns = new_names

df = df.merge(team_aggregated, how = 'left', on = ['match_id', 'team_id'])

match_aggregated = df.groupby(['match_id'], as_index = False)[
                                        feature_columns].agg(['max', 'mean'])
new_names = ['_'.join(['match'] + list(x)
                                         ) for x in match_aggregated.columns]
match_aggregated.columns = new_names

df = df.merge(match_aggregated, how = 'left', on = ['match_id'])

not_features = ['match_id', 'party_size', 'team_id',
                'game_size', 'team_placement'] + target
feature_columns = list(filter(lambda x: x not in not_features, df.columns))
```

Po wyznaczeniu nowych cech i połączeniu ich z wcześniej posiadaną macierzą cech można ponownie podzielić dane na zbiory uczący i testowy oraz przeprowadzić trening i ewaluację modelu.

UWAGA

Zmienna player_survive_time nadal jest usunięta ze zbioru danych i nie jest brana pod uwagę przez model.

```
df_train = df[df['match_id'].isin(train_match_ids)]
df_test = df[df['match_id'].isin(test_match_ids)]

X_train = df_train[feature_columns]
y_train = df_train[target]
X_test = df_test[feature_columns]
y_test = df_test[target]
feature_model = RandomForestRegressor(n_jobs = 10)

feature_model.fit(X_train, y_train)

test_predictions = feature_model.predict(X_test)
mean_absolute_error(y_test, test_predictions)
```

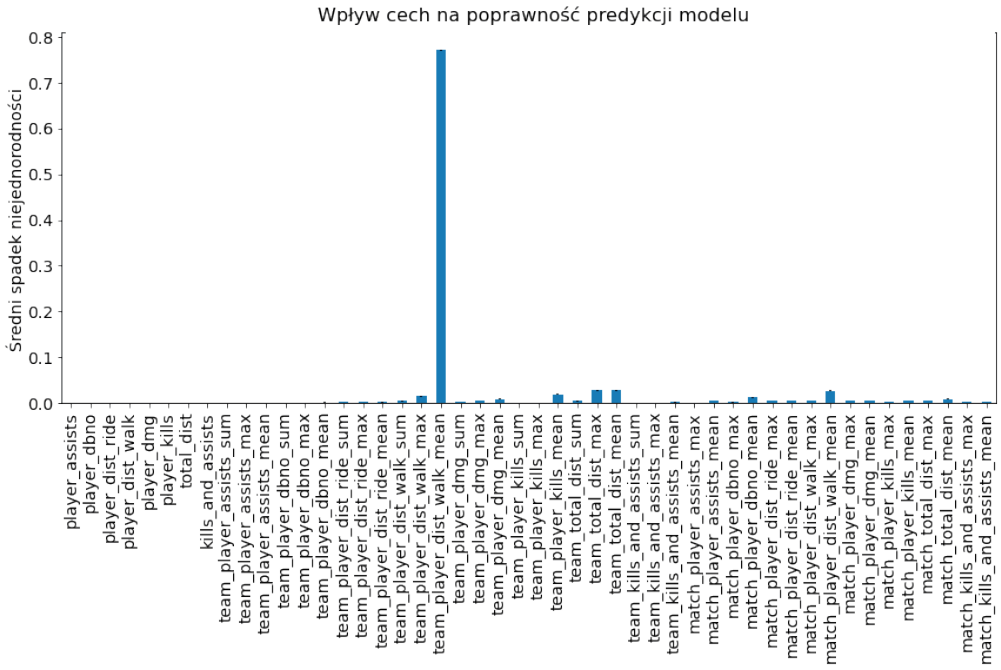
Zastosowanie inżynierii cech pozwoliło znacząco zmniejszyć popełniany błąd – wartość MAE spadła z około 0,12 do około 0,08. Warto zauważyć, że jest to wynik lepszy nawet od wyniku uzyskanego z uwzględnieniem zmiennej player_survive_time.

```
test_df_with_preds = pd.concat([df_test.reset_index(drop = True),
                                                pd.Series(test_predictions,
                                                name = 'prediction')],
                                    axis = 1)
```

```
grouped_team_results = test_df_with_preds.groupby(
  ['match_id', 'team_id']).agg(np.mean).groupby(
   'match_id').agg(np.argmin)

accuracy_score(grouped_team_results['normalized_team_placement'],
               grouped_team_results['prediction'])
```

W wyniku zastosowania nowych cech dokładność predykcji wzrosła do 0,484. Nadal jest dużo mniejsza niż dokładność uzyskana z użyciem zmiennej player_survive_time, jednak można zauważyć istotny wzrost względem dokładności otrzymanej wyłącznie na podstawie cech zawartych w oryginalnym zbiorze danych (nieuzyskanych w wyniku inżynierii cech).

```
importances = feature_model.feature_importances_
std = np.std([tree.feature_importances_ for tree in
                            feature_model.estimators_],axis = 0)

forest_importances = pd.Series(importances, index = feature_columns)

fig, ax = plt.subplots(figsize = (12,8))
forest_importances.plot.bar(yerr = std, ax = ax)
ax.set_title("Wpływ cech na poprawność predykcji modelu", fontsize = 16)
ax.set_ylabel("Średni spadek niejednorodności", fontsize = 14)
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
fig.tight_layout()
plt.show()
```



Rysunek 16.5. Wpływ analizowanych cech na predykcje modelu uczonego na danych uzyskanych w wyniku inżynierii cech

Na podstawie rankingu ważności cech zwróconego przez model (rys. 16.5) można stwierdzić, że cechą mającą największy wpływ na predykcje była zmienna team_player_dist_walk_mean, czyli średni dystans, który przeszli członkowie danej drużyny. Jest to jedna z cech, które zostały wyznaczone samodzielnie na podstawie wartości innych cech. Widać więc, że odpowiednie przetworzenie danych może znacząco wpłynąć na zdolności predykcyjne modelu i doprowadzić do otrzymania modelu lepszej jakości.

Zadanie 16.1. Optymalizacja klasyfikatora lasu losowego

1. Spróbuj poprawić wyniki klasyfikacji zdarzeń akustycznych uzyskane przez model przedstawiony w Przykładzie 16.2. W tym celu przeprowadź optymalizację trzech hiperparametrów lasu losowego:
   – n_estimators – liczba drzew klasyfikacyjnych, które będą tworzyły las,
   – max_depth – maksymalna głębokość (liczba poziomów) drzewa,
   – min_samples_split – minimalna liczba obiektów, które muszą znajdować się w węźle, by można go było poddać kolejnemu podziałowi.
   Lista pozostałych hiperparametrów modelu znajduje się w dokumentacji klasy RandomForestClassifier pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.
2. Wykonaj trening i ewaluację modelu. Zmierz czas potrzebny do wykonania optymalizacji hiperparametrów.
3. Przeanalizuj uzyskane wyniki klasyfikacji. Czy wyniki otrzymane przy użyciu optymalizowanego modelu są lepsze od tych, które uzyskano w Przykładzie 16.2 z wykorzystaniem domyślnych wartości hiperparametrów?

   Podpowiedź: liczba prób (*trials*) powinna wynosić co najmniej 60, aby wyniki optymalizacji były zadowalające.

Zadanie 16.2. Klasyfiacja zdarzeń akustycznych – algorytm ExtraTrees

Algorytmem bardzo podobnym do lasu losowego jest algorytm ExtraTrees. Od lasu losowego różni go przede wszystkich kryterium podziału drzewa w danym węźle: w ExtraTrees kryterium jest losowe, natomiast w lasach losowych stosowane jest kryterium optymalne (dające najlepszy podział), dzięki czemu algorytm ExtraTrees zazwyczaj jest szybszy. Kolejną istotną różnicą jest to, że podczas podziału danych w lesie losowym stosowane jest tzw. losowanie ze zwracaniem, natomiast w ExtraTrees nie, co oznacza, że w ExtraTrees obserwacje podane do kolejnych drzew nie będą się powtarzać, natomiast w lesie losowym mogą.

1. Zastosuj algorytm ExtraTrees do klasyfikacji zdarzeń akustycznych opisanych w Przykładach 16.1. i 16.2. Przeprowadź optymalizację hiperparametrów modelu w celu maksymalizacji wartości metryki F1. Listę hiperparametrów modelu ExtraTrees znajdziesz w dokumentacji klasy ExtraTreesClassifier pod adresem: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html.
2. Zmierz czas potrzebny na przeprowadzenie optymalizacji modelu. Czy jest on krótszy od czasu potrzebnego na optymalizację modelu lasu losowego, który wyznaczono w Zadaniu 16.1?
3. Porównaj wyniki otrzymane przez klasyfikator ExtraTrees i klasyfikator lasu losowego. Który model uzyskał większą dokładność klasyfikacji?

Zadanie 16.3. Klasyfiacja zdarzeń akustycznych – pakiet LightGBM

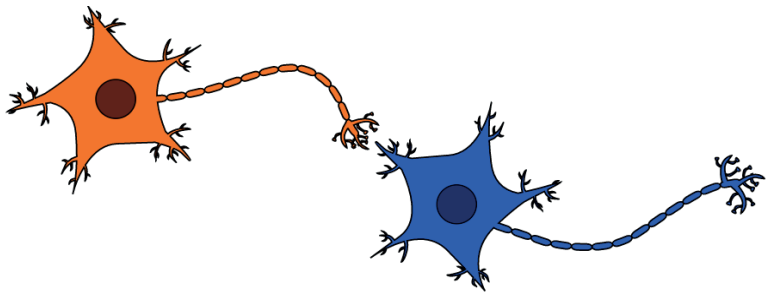
Algorytmy drzewiaste są zaimplementowane nie tylko w bibliotece scikit-learn, ale również w pakietach zewnętrznych. Jednym z najczęściej stosowanych pakietów jest pakiet LightGBM, który w porównaniu z lasami losowymi i algorytmem ExtraTrees cechuje się przede wszystkim większą efektywnością, mniejszym zużyciem pamięci i często pozwala uzyskać większą dokładność klasyfikacji. Ma też o wiele więcej hiperparametrów, które możemy określić lub optymalizować, znajdujących się pod adresem: https://lightgbm.readthedocs.io/en/latest/Parameters.html.

1. Zoptymalizuj algorytm drzewiasty zaimplementowany w pakiecie LightGBM. Optymalizacji poddaj wybrane hiperparametry spośród wymienionych poniżej:
   – boosting_type – algorytm wzmocnienia gradientowego,
   – num_leaves – maksymalna liczba liści, które mogą zostać utworzone,
   – n_estimators – liczba drzew,
   – max_depth – maksymalna głębokość drzewa (liczba poziomów),
   – learning_rate – współczynnik określający, z jakim krokiem mają być zmieniane wagi modelu podczas dążenia do osiągnięcia minimalnego błędu popełnianego przez model (zbyt mała wartość współczynnika powoduje nadmiernie długi czas uczenia, zbyt duża zwiększa ryzyko „przeskoczenia" minimum błędu i nieznalezienia optymalnych wag),
   – subsample – określa, jaka część obiektów (domyślnie losowanych bez zwracania) ma być podana do drzewa podczas treningu,
   – colsample_bytree – określa, jaka część wszystkich cech ma być podana do drzewa podczas treningu,
   – min_child_samples – minimalna liczba obiektów, które muszą znaleźć się w liściu,
   – min_split_gain – minimalny zysk informacyjny, który musi być osiągnięty po podziale węzła (jeżeli nie jest osiągnięty, to węzeł nie będzie dalej dzielony).
2. Przeprowadź trening i ewaluację modelu o dobranych wartościach hiperparametrów. Porównaj macierze pomyłek uzyskane przez ten model oraz przez modele uzyskane w poprzednich zadaniach. Która klasa jest najczęściej źle klasyfikowana bez względu na użyty model? Jak myślisz, dlaczego?

17 Wprowadzenie do sieci neuronowych

Sieci neuronowe to oddzielna i obszerna gałąź metod uczenia maszynowego. Choć pierwsze modele mogą wydawać się trywialne, to całkowicie zmieniły sposób myślenia i podejście do rozwiązywania problemów za pomocą sztucznej inteligencji. Ponadto pierwsze modele sieci neuronowych w połączeniu z aktualną cyfryzacją danych i ogromnym wzrostem możliwości komputerów stały się podstawą bardzo poważnych, wręcz przełomowych narzędzi, które zmieniły nasz świat. Jeszcze kilka dekad temu wydawało się, że automatyczne rozpoznawanie obrazów lub dźwięku pozostanie w zakresie fantastyki naukowej. Obecnie jest to nasza codzienność.

Sztuczna sieć neuronowa, jak wskazuje nazwa, jest modelem zainspirowanym pracą ludzkiego mózgu, który zbudowany jest z komórek nerwowych zwanych neuronami. Tak więc podstawową analogią pomiędzy sztucznymi i biologicznymi sieciami neuronowymi jest ustrukturyzowana budowa bazująca na wielokrotności tego samego elementu. Podobieństwo dotyczy również przekazywania informacji pomiędzy neuronami, ponieważ przewodzenie następuje zwykle tylko w jednym kierunku i obowiązuje zasada „wszystko albo nic", co oznacza, że dla słabych impulsów nie ma żadnej reakcji, zaś po przekroczeniu pewnego progu następuje jej przekazanie. Na rysunku 17.1 przedstawiono poglądowo dwa połączone ze sobą neurony.



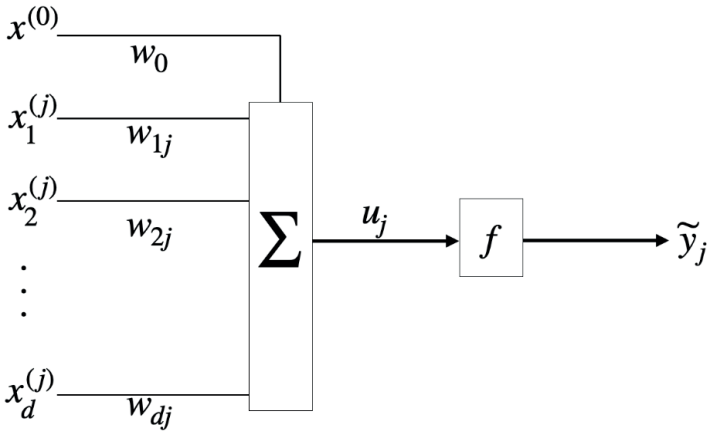
Rysunek 17.1. Dwa połączone ze sobą neurony

Pierwszy model pojedynczej komórki nerwowej opracowali Warren McCulloch i Walter Pitts w 1943 roku. Wykorzystano go do zamodelowania podstawowych funkcji logicznych takich jak koniunkcja czy alternatywa (McCulloch i Pitts, 1943). Model ten wykorzystał Frank Rosenblatt, proponując w 1958 roku pierwszą prostą sieć neuronową zwaną perceptronem.

17.1. Perceptron

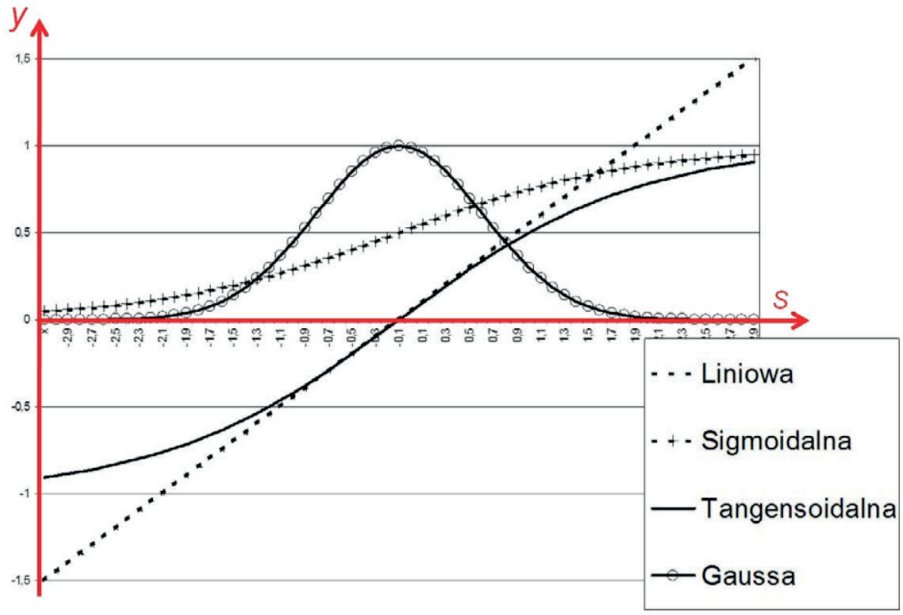
Budowa perceptronu:

- zbiór danych treningowych, tzw. sygnałów zewnętrznych $x^{(j)} = (x_1^{(j)}, x_2^{(j)}, \ldots, x_d^{(j)})$, $j = 1, ..., N$, które są kolejno kierowane na wejście sieci,
- wejście numer zero, gdzie podawany jest stały sygnał $x^{(0)} = 1$ z wagą w_0,
- wejścia $i = 1, ..., d$, przez które wprowadzane są sygnały zewnętrzne x_i,
- wagi w_{ij} określające wpływ poszczególnych zmiennych ($i = 1, ..., d$) dla j-tego obiektu,
- sumator, w którym obliczane jest łączne pobudzenie u_j neuronu,
- funkcja aktywacji $f(u_j)$ określająca wartość na wyjściu neuronu.



Rysunek 17.2. Budowa perceptronu Rosenblatta

Konstruując taką sieć, należy wybrać funkcję aktywacji. Jak wynika ze schematu przedstawionego na rysunku 17.2, funkcja ta określa ostateczną wartość predykcji. Wymaga się, aby była to funkcja monotoniczna, różniczkowalna w całej swojej dziedzinie i nie powinna być skokowa, choć często pożądane jest, aby swoim kształtem była bliska funkcji skokowej. Tak więc warunki te pozostawiają nadal dużą swobodę wyboru. Jednak w praktycznych zastosowaniach najczęściej można znaleźć cztery funkcje: liniową, sigmoidalną, tangensa hiperbolicznego i funkcję Gaussa. Bardzo często neurony określa się przymiotnikami pochodzącymi od użytej funkcji aktywacji. Na rysunku 17.3 przedstawiono wartości wyszczególnionych funkcji (Tadeusiewicz i Szaleniec, 2015).



Rysunek 17.3. Najczęściej stosowane funkcje aktywacji (Tadeusiewicz i Szaleniec, 2015)

W tak zbudowanej sieci na wyjściu sumatora otrzymujemy kombinację liniową sygnału zewnętrznego i wektora wag, czyli:

$$u_j = \sum_{i=1}^{d} w_{ij} x^{(j)} - w_0 x_0 = -\sum_{i=1}^{p} P(d_i) \cdot \log_2(P(d_i)), \tag{17.1}$$

gdzie $w_0 x_0$ nazywa się wartością progową funkcji aktywacji.

Na wyjściu neuronu otrzymujemy natomiast wartość funkcji dla pobudzenia neuronu:

$$\tilde{y}_j = f\left(w_0 + \sum_{i=1}^{d} w_i x_i \right). \tag{17.2}$$

Wartość $\tilde{y}_j$ jest wtedy predykcją dla obiektu $x^{(j)}$, którą na etapie treningu czy testowania można porównać z rzeczywistą wartością atrybutu decyzyjnego $y^{(j)}$. Wtedy trenowanie sieci neuronowej jest iteracyjne i polega na wprowadzaniu na wejścia neuronu kolejnych obiektów ze zbioru treningowego oraz odpowiedniej modyfikacji wag w przypadkach, w których predykcja była błędna. Zwykle pierwszy zestaw wag jest dobierany losowo. Przedstawienie całego zbioru treningowego kończy tzw. epokę i jeżeli nie zostało spełnione kryterium stopu, trening jest kontynuowany z nową, losowo przyjętą kolejnością obiektów ze zbioru treningowego. Ostatecznie celem treningu jest znalezienie takich wag, które pozwolą na zadowalającą predykcję.

Kroki algorytmu (trenowanie perceptronu)

1. Ustalenie wag początkowych (zwykle w sposób losowy).
2. Niech j: = 1.
3. Przedstawienie j-tego obiektu $x^{(j)}$ i otrzymanie wyniku sieci $\tilde{y}_j$.
4. Porównanie wyniku $\tilde{y}_j$ z wartością atrybutu decyzyjnego $y^{(j)}$ dla $x^{(j)}$.
5. Sprawdzenie, czy odpowiedź sieci jest prawidłowa:
   - jeżeli nie, to modyfikacja wag $w_i := w_i + \Delta w_i^{(*)}$, gdzie Δw_i zależy od algorytmu uczenia (np. metoda gradientowa),
   - jeżeli tak, przejście do kolejnego kroku algorytmu.
6. Jeżeli $j = N$, obliczenie błędu całej epoki oraz:
   - gdy błąd epoki większy od założonej wartości, losowa zmiana kolejności w zbiorze treningowym i powrót do kroku nr 2,
   - w przeciwnym wypadku $j := j + 1$ i powrót do punktu nr 3.

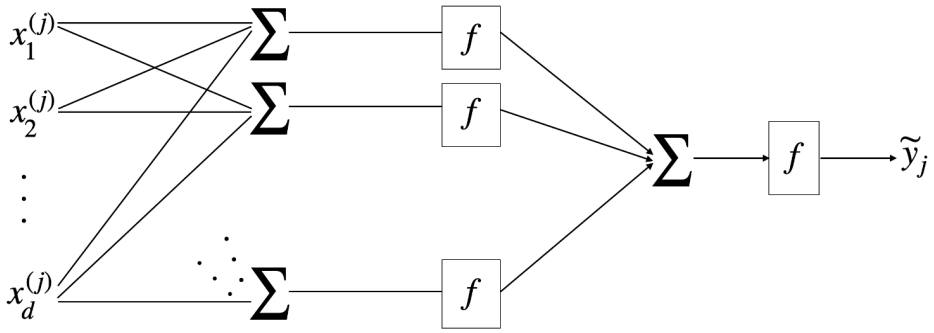
(*) Przykładowo:
$w_i := w_i + \alpha \cdot (y^{(j)} - \tilde{y}_j) \cdot x_i^{(j)}$,
$w_0 := w_0 + \alpha \cdot (y^{(j)} - \tilde{y}_j)$,
gdzie $\alpha > 0$ jest współczynnikiem uczenia.

Aby lepiej zrozumieć, czym są otrzymane w wyniku treningu wagi, rozważmy przykład klasyfikacji binarnej. W wyniku treningu perceptronu otrzymywana jest $(N - 1)$-wymiarowa hiperpłaszczyzna, która będzie granicą decyzyjną. Płaszczyzna ta dzieli N-wymiarową przestrzeń wektorów wejściowych na dwie półprzestrzenie odpowiadające poszczególnym klasom. W szczególności, jeżeli przestrzeń obiektów jest dwuwymiarowa, to granica decyzyjna będzie prostą. Zauważmy, że perceptron będzie skuteczny jedynie w przypadku danych, które są liniowo separowalne. Okazuje się jednak, że dodanie drugiej warstwy neuronów pozwala uzyskać nieliniową granicę decyzyjną. Ponadto im więcej warstw zostanie użytych, tym granica decyzyjna możliwa do uzyskania będzie bardziej złożona.

17.2. Wielowarstwowe sieci neuronowe

Wielowarstwowa sieć neuronowa to sieć, w której kilka warstw neuronów połączonych jest tak, że wejściami kolejnej warstwy są wyjścia neuronów warstwy wcześniejszej (Osowski, 1996). Już trzy warstwy pozwalają na uzyskanie dowolnej charakterystyki. Jednak nie opracowano optymalnej metody projektowania sieci neuronowej dla dowolnego problemu. Na rysunku 17.4 przedstawiono przykład prostej sieci dwuwarstwowej. W tym przypadku każdy element warstwy ukrytej dzieli przestrzeń obiektów w zbiorze treningowym na dwie półprzestrzenie rozdzielone hiperpłaszczyzną, a następnie element

wyjściowy dzieli każdą z półprzestrzeni na dwie podprzestrzenie. Zatem granica decyzyjna takiej dwuwarstwowej sieci jest zbiorem wielościennym wypukłym.



Rysunek 17.4. Schemat prostej sieci dwuwarstwowej

Kroki algorytmu (trenowanie wielowarstwowej sieci)

1. Ustalenie hiperparametrów (m.in. liczby warstw oraz liczby neuronów w warstwie).
2. Losowa inicjacja wag (często pożądane małe wartości).
3. Obliczenie odpowiedzi sieci, **warstwa po warstwie**.
4. Obliczenie błędu na wyjściu sieci.
5. **Propagacja błędu do wszystkich wcześniejszych warstw, czyli modyfikacja wagi w każdym neuronie (wagi inicjowane w kroku nr 2 zostają nadpisane)**.
6. Powtórzenie kroków nr 3–5 dla kolejnego obiektu ze zbioru treningowego.
7. Po wyczerpaniu obiektów w zbiorze treningowym (koniec epoki) sprawdzenie kryterium stopu (np. średni błąd przestaje maleć). Jeżeli kryterium stopu nie jest spełnione, to losowa zmiana kolejności obiektów i powrót do kroku nr 3.

Przykład 17.1. Rozpoznawanie mowy przy użyciu jednokierunkowej sieci neuronowej

Do implementacji sieci neuronowych w Pythonie stosowane są głównie dwa pakiety: TensorFlow (Abadi i in., 2016) i PyTorch (Paszke i in., 2019). Zarówno TensorFlow, jak i PyTorch zapewniają bardzo dużą swobodę w definiowaniu architektury sieci. Jest to ich duża zaleta, jednak dla początkujących osób może być to nieco problematyczne – każda klasa musi być odpowiednio zdefiniowana, konieczne jest samodzielne napisanie funkcji, które będą wykorzystywane m.in. do wczytywania danych i wykonania każdej epoki treningu sieci. Aby ułatwić pracę użytkownikom mniej zaawansowanym i ustrukturyzować kod, stworzono nakładki na oba pakiety, które upraszczają pracę z sieciami neuronowymi. W przypadku PyTorcha najczęściej używane są dwie pakiety: Ignite (Fomin i in., 2020) i Lightning (Falcon, 2019). W rozważanym przykładzie zostanie użyty pierwszy z nich, czyli pakiet Ignite.

Tematem, który dotychczas nie został poruszony w niniejszej publikacji, a dotyczy bardzo istotnego zastosowania uczenia maszynowego, jest rozpoznawanie mowy. Na tym przykładzie można dobrze pokazać potencjał sieci neuronowych – już prosta sieć, wykorzystująca jedynie podstawowe rodzaje warstw, pozwala osiągnąć stosunkowo dobre rezultaty.

W przykładzie wykorzystana będzie sieć jednokierunkowa złożona z warstw gęstych, które cechują się tym, że każdy neuron wejściowy warstwy połączony jest z każdym neuronem wyjściowym warstwy poprzedniej. Każde takie połączenie ma określoną wagę – te wagi to parametry sieci, które modyfikowane są podczas treningu sieci.

Do rozpoznawania mowy często stosowane są cechy ekstrahowane z sygnału w wyniku obliczenia spektrogramu i przefiltrowania go przy użyciu banku filtrów melowych, które opisano w rozdziale 7. Zazwyczaj ekstrakcja tych cech poprzedzona jest filtracją sygnału mowy filtrem preemfazy, który ma na celu uwydatnienie składowych o wyższych częstotliwościach. Po wyznaczeniu cech sygnału uzyskane wartości przelicza się na skalę logarytmiczną. Wszystkie te operacje można przeprowadzić przy użyciu biblioteki python_speech_features dostępnej pod adresem: http://python-speech-features.readthedocs.io/.

Nagrania, które będą poddawane analizie, pochodzą z bazy TensorFlow Speech Commands v0.02 znajdującej się pod adresem: http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz i zawierają 35 słów w języku angielskim. Sygnały mają długość 1 s – wszystkie krótsze zostały symetrycznie uzupełnione zerami, aby uzyskać stałą długość sygnału. Aby zredukować czas potrzebny na trening sieci oraz ograniczyć rozmiar danych, w przykładzie wykorzystane zostanie tylko 300 losowo wybranych nagrań każdego ze słów. Należy jednak pamiętać, że w praktyce powinno się wykorzystywać możliwe duży zbiór danych.

Ekstrakcję cech opisaną powyżej można zrealizować za pomocą następującego kodu.

```
from python_speech_features import logfbank
import scipy.io.wavfile as wav
import os
from pathlib import Path

paths = []
labels = []
labels_categorical = []
root = 'SpeechDataset/'
for ind, subdir in enumerate(os.listdir(root)):
  for file in os.listdir(os.path.join(root,subdir))[:300]:
    filepath = os.path.join(root,subdir,file)
    paths.append(filepath)
    labels.append(ind)
    labels_categorical.append(subdir)

logfbank_feats = []
for signal_path in paths:
  fs,sig = wav.read(signal_path)
  fbank_feat = logfbank(sig,samplerate = fs)
  logfbank_feats.append(fbank_feat)

lengths = [i.shape[0] for i in logfbank_feats]
max_len = np.max(lengths)

padded_feats = np.zeros((len(lengths), max_len,
                         logfbank_feats[0].shape[1]))

for i, feats in enumerate(logfbank_feats):
  padded_feats[i,:,:] = np.pad(feats, ((np.int(np.floor((
                         max_len-feats.shape[0])/2)),
                         np.int(np.ceil((max_len-feats.shape[
                                                 0])/2))), (0,0)))
```

```
np.save('logfbank_feats.npy', padded_feats)
np.save('labels.npy', labels)
np.save('labels_categorical.npy', labels_categorical)
```

Macierz cech oraz wektor etykiet zostały zapisane do plików, dzięki czemu będzie możliwe ich ponowne wykorzystanie bez konieczności ich powtórnego czasochłonnego obliczania.

Po zakończeniu ekstrakcji cech dane należy podzielić na zbiory. Podczas omawiania innych metod uczenia maszynowego wykorzystywane były dwa zbiory: uczący i testowy. Tym razem dane należy podzielić na trzy zbiory: uczący, walidacyjny i testowy. Zbiór uczący i testowy mają takie samo zastosowanie jak zawsze – uczący wykorzystywany jest do treningu modelu, a testowy do jego ewaluacji. Zbiór walidacyjny posłuży do ewaluacji sieci neuronowej podczas treningu. Zbiór uczący będzie zawierał 80% analizowanych danych, natomiast zbiór walidacyjny i testowy po 10% danych.

```
import numpy as np
from sklearn.model_selection import train_test_split

feats = np.load('logfbank_feats.npy')
labels = np.load('labels.npy')

feats = feats.reshape(feats.shape[0], -1)
feats = feats.astype(np.float32)

X_train, X_val_test, y_train, y_val_test = train_test_split(feats,
                                                            labels,
                                                            random_state = 42,
                                                            stratify = labels,
                                                            train_size = 0.8)

X_val, X_test, y_val, y_test = train_test_split(X_val_test,
                                                y_val_test,
                                                random_state = 42,
                                                stratify = y_val_test,
                                                train_size = 0.5)
```

Pakiety do implementacji i uczenia sieci neuronowych wymagają, by dane były w postaci tensorów, a nie macierzy lub tablic, tak jak było w przypadku dotychczas poznanych algorytmów. Należy więc zamienić wektory etykiet oraz macierze cech na tensory i utworzyć z nich TensorDataset, czyli zbiór danych w formacie dostosowanym do użycia w sieci neuronowej. W tym celu używa się funkcji torch.tensor oraz klasy TensorDataset.

```
import torch
from torch.utils.data import TensorDataset

trainset = TensorDataset(torch.tensor(X_train), torch.tensor(y_train))
valset = TensorDataset(torch.tensor(X_val), torch.tensor(y_val))
testset = TensorDataset(torch.tensor(X_test), torch.tensor(y_test))
```

Następnie przy użyciu klasy DataLoader tworzy się obiekty, które posłużą do wczytania danych przez sieć. Definiuje się w niej parametr batch_size określający liczbę próbek ze zbioru jednocześnie podanych do sieci. Podział zbioru na mniejsze części zwane wsadami (ang. *batch*) pozwala nie tylko zmniejszyć zużycie pamięci, co jest szczególnie istotne przy bardzo dużych zbiorach danych, ale też zazwyczaj przyspiesza proces uczenia.

W sytuacji gdy rozmiar wsadu jest mniejszy niż rozmiar całego zbioru danych, wagi modelu aktualizowane są podczas uczenia na każdym wsadzie, jednak epoka kończy się dopiero po przejściu przez sieć danych zawartych we wszystkich wsadach. Epoka jest wówczas dzielona na iteracje, których liczba jest równa liczbie wsadów.

```
from torch.utils.data import DataLoader

train_loader = DataLoader(trainset, batch_size = 256)
val_loader = DataLoader(valset, batch_size = 256)
test_loader = DataLoader(testset, batch_size = 256)
```

Po wczytaniu i odpowiednim przygotowaniu danych należy zdefiniować klasę, w której będzie zapisana architektura sieci. W klasie definiuje się dwie funkcje:

1) **init** – w niej należy określić, jakie warstwy będą występowały w sieci (ich rodzaj i rozmiar),
2) **forward** – w niej należy określić kolejność warstw, a tym samym kierunek przepływu danych wewnątrz sieci.

Warstwy gęste, z których będzie składała się zaimplementowana sieć jednokierunkowa, to warstwy liniowe, które tworzy się z użyciem klasy nn.Linear. Do klasy jako argumenty należy podać wymiary warstwy: liczbę wejść oraz liczbę wyjść.

ZASADA

Liczba wejść warstwy musi być taka sama jak liczba wyjść poprzedniej warstwy. W przypadku pierwszej warstwy w sieci liczba wejść musi być tak dobrana, by pasowała do rozmiaru danych uczących. W analizowanych w tym przykładzie danych każdy sygnał opisany jest za pomocą 2574 cech, więc taka musi być liczba wejść pierwszej warstwy.

Liczba wyjść ostatniej warstwy powinna być równa liczbie klas, które znajdują się w danych. W analizowanych danych występuje 35 klas, więc tyle wyjść powinna mieć warstwa wyjściowa sieci.

W funkcji forward określa się również funkcję aktywacji, która będzie użyta w ostatniej warstwie. Dobór funkcji aktywacji ma wpływ na uzyskiwane wyniki klasyfikacji, ponieważ to ona służy do obliczenia wartości pojawiającej się na wyjściu całej sieci. Jest wiele różnych funkcji aktywacji o różnych zastosowaniach. W przykładzie użyta zostanie funkcja log_softmax, czyli logarytmowana znormalizowana funkcja wykładnicza, zwana też funkcją logistyczną. Funkcja softmax pozwala uzyskać na wyjściu sieci wektor prawdopodobieństw przynależności obiektu do każdej z klas, natomiast zastosowanie logarytmu jest szybsze pod względem numerycznym.

```
import torch.nn.functional as F
import torch.nn as nn

class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(2574, 256)
    self.fc2 = nn.Linear(256, 120)
    self.fc3 = nn.Linear(120, 35)
```

```
  def forward(self, x):
    x = self.fc1(x)
    x = self.fc2(x)
    x = self.fc3(x)

    return F.log_softmax(x, dim = 1)
```

Przed rozpoczęciem treningu sieci należy określić, na czym będą wykonywane obliczenia, czyli czy jest dostępna karta graficzna, czy też będzie wykorzystywany zwykły procesor. Takiego określenia dokonuje się przez użycie zmiennej device – funkcja torch.cuda.is_available() służy do sprawdzenia, czy jest dostępna karta graficzna. Jeżeli tak, to zostanie ona użyta (device = „cuda"), jeżeli nie, to trzeba wykorzystać procesor (device = „cpu").

Następnie należy ustalić kryterium uczenia się sieci i do czego będzie ona dążyć, czyli określić zmienną criterion. W przykładzie jako kryterium zostanie użyta klasa nn. NLLLoss (ang. *negative log likelihood loss*) – im mniejszą wartość będzie przyjmować, tym mniejsza będzie różnica pomiędzy klasą rzeczywistą a predykcją sieci. NLLLoss jest odpowiednikiem entropii krzyżowej w sytuacji, gdy prawdopodobieństwa przynależności do klasy są zlogarytmowane, czyli używana jest funkcja aktywacji log_softmax. Algorytm obliczania entropii krzyżowej, w przypadku binarnym nazywanej startą logarytmiczną, przedstawiony został w rozdziale 14.

UWAGA

Sieć neuronowa wykorzystywana do klasyfikacji ma tyle wyjść, ile klas występuje w danych, nie trzeba więc liczyć opisanych w rozdziale 14 prawdopodobieństw skorygowanych – każde wyjście zwraca prawdopodobieństwo przynależności obiektu do jednej, konkretnej klasy. W takim przypadku entropia krzyżowa jest liczona jako ujemna średnia logarytmiczna prawdopodobieństw zwróconych przez odpowiednie wyjścia (te, które odpowiadają rzeczywistym klasom rozważanych obiektów).

Kolejnym istotnym krokiem jest dobór algorytmu optymalizacji wag modelu (ang. *optimizer*), który zmienia ich wartości w taki sposób, aby osiągnięto zadane kryterium, np. minimalną wartość NLLLoss.

Jednym z najczęściej używanych algorytmów optymalizacji wag jest algorytm stochastycznego zejścia gradientowego opisany w Przykładzie 12.2. Wielkość kroków podczas zmiany wag określana jest przez współczynnik uczenia (ang. *learning rate*).

```
import torch.optim as optim

device = "cuda" if torch.cuda.is_available() else "cpu"
criterion = nn.NLLLoss()
model = Net()
model.to(device)
optimizer = optim.SGD(model.parameters(), lr = 3e-4, momentum = 0.9)
```

W kolejnym kroku należy zainicjalizować zmienne, w których będą zapisywane aktualne stany modelu i optimizera.

```
init_model_state = model.state_dict()
init_opt_state = optimizer.state_dict()
```

Następnie należy zdefiniować dwa obiekty:

1) **trainer** służący do uczenia sieci; podaje się do niego model (sieć neuronową), kryterium oceny (criterion) oraz wskazuje urządzenie, na którym będą wykonywane obliczenia (device); podczas użycia obiektu trainer wagi modelu aktualizowane są tak, by dążyły do ekstremum zdefiniowanego kryterium (w tym przypadku do minimum, bo użytym kryterium jest NLLLoss);
2) **evaluator** służący do ewaluacji sieci, czyli do walidacji i wyliczenia metryk na zbiorze testowym; podaje się do niego model i metryki, które będą użyte; podczas jego użycia nie są aktualizowane wagi.

Dodatkowo, żeby mieć kontrolę nad przebiegiem treningu i tym, jaka jego część została już wykonana, zostanie użyty pasek postępu wyświetlany za pomocą funkcji ProgressBar. Będzie on wyświetlał wyniki uzyskane na zbiorze uczącym podczas kolejnych epok.

```
from ignite.metrics import Loss, Accuracy
from ignite.contrib.handlers import ProgressBar
from ignite.engine import (create_supervised_trainer,
                           create_supervised_evaluator)

trainer = create_supervised_trainer(model, optimizer,
                                    criterion,
                                    device = device)
evaluator = create_supervised_evaluator(model,
                    metrics = {"acc": Accuracy(),
                               "loss": Loss(nn.NLLLoss())},
                    device = device)
ProgressBar(persist = True).attach(trainer,
                                   output_transform = lambda x: {
                                                        "batch loss": x})
```

Przed rozpoczęciem treningu sieci należy również określić liczbę epok. Po każdej epoce można wyliczyć metryki uzyskiwane na zbiorze walidacyjnym – jest to zbiór danych, które nie są podawane do sieci w celu uczenia, więc uzyskane na nich metryki pozwalają ocenić, czy sieć się uczy oraz czy się nie przeucza.

Żeby przeprowadzić walidację w pakiecie Ignite, należy wywołać obiekt evaluator, do którego podaje się dane walidacyjne. Evaluator musi być umieszczony w funkcji podłączonej do obiektu trainer. Poniżej przedstawiono, w jaki sposób można spowodować wyświetlanie wyników walidacji po każdej epoce.

Po zakończonym treningu obiekt evaluator posłuży do wyliczenia metryk na zbiorze testowym.

```
from ignite.engine import Events

@trainer.on(Events.EPOCH_COMPLETED)
#walidacja ma być przeprowadzona po zakończeniu epoki

def log_validation_results(trainer):
  evaluator.run(val_loader)
  metrics = evaluator.state.metrics
  print("Validation Results - Epoch: {} Avg accuracy: {:.2f}
         Avg loss: {:.2f}".format(
         trainer.state.epoch, metrics[
         'acc'], metrics['loss']))
```

```
  #dokładność klasyfikacji i wartość funkcji kosztu (loss) będą
  #wyświetlone z dokładością do dwóch miejsc po przecinku

trainer.run(train_loader, max_epochs = 150)

evaluator.run(test_loader)
print(evaluator.state.metrics)
```

Przykład 17.2. Automatyczny dobór współczynnika uczenia

W Przykładzie 17.1 wartość współczynnika uczenia została wpisana ręcznie do funkcji *optim.SGD* i wynosi 3e-4. Współczynnik uczenia może być też dobierany w sposób automatyczny – służy do tego funkcja FastaiLRFinder. Dobrana wartość zależy od tego, ile wynosi parametr diverge_th: poszukiwania najlepszej wartości współczynnika uczenia przerywane są w momencie, gdy zostanie spełnione kryterium current loss > diverge_th * best_loss.

UWAGA

Parametr diverge_th może być dowolną liczbą nie mniejszą niż 1, domyślnie wynosi 5. Wartość mniejsza niż 5 powoduje szybsze zakończenie poszukiwań najlepszego współczynnika uczenia, ale jej wybór jest dość ryzykowny, ponieważ jeżeli na początku poszukiwań trafimy na minimum lokalne, to szukanie zostanie zakończone. W praktyce może się okazać, że niezatrzymanie poszukiwań i zezwolenie na tymczasowe osiąganie większych wartości current_loss pozwoli trafić na minimum globalne lub przynajmniej minimum lokalne o mniejszej wartości, a tym samym lepiej dobrać wartość parametru i w konsekwencji przeprowadzić bardziej efektywny trening sieci.

```
from ignite.handlers import FastaiLRFinder
import matplotlib.pyplot as plt

lr_finder = FastaiLRFinder()
to_save = {'model': model, 'optimizer': optimizer}
with lr_finder.attach(trainer, to_save,
         diverge_th = 1.1) as trainer_with_lr_finder:
  #domyślnie start_lr jest taki, jak określony w obiekcie optimizer,
  #a end_lr = 10
  trainer_with_lr_finder.run(train_loader)
  results = lr_finder.get_results()

lr_finder.plot()
print("Suggested LR", lr_finder.lr_suggestion())
```

Po znalezieniu najlepszej wartości współczynnika uczenia należy podać tę wartość do obiektu optimizer – służy do tego funkcja apply_suggested_lr.

```
trainer = create_supervised_trainer(model, optimizer, criterion,
                                                        device = device)
```

```
evaluator = create_supervised_evaluator(model,
                                        metrics = {"acc": Accuracy(),
                                                   "loss": Loss(nn.NLLLoss())},
                                        device = device)
ProgressBar(persist = True).attach(trainer,
                                   output_transform = lambda x: {"batch loss": x})

@trainer.on(Events.EPOCH_COMPLETED)
def log_validation_results(trainer):
  evaluator.run(val_loader)
  metrics = evaluator.state.metrics
  print("Validation Results - Epoch: {} Avg accuracy: {
                              :.2f} Avg loss: {:.2f}".format(
                              trainer.state.epoch, metrics[
                              'acc'], metrics['loss']))

lr_finder.apply_suggested_lr(optimizer)
print('Training with suggested lr: ',
      optimizer.param_groups[0]['lr'])

trainer.run(train_loader, max_epochs = 150)

evaluator.run(test_loader)
print(evaluator.state.metrics)
```

Przykład 17.3. Użycie warstwy dropout

Sieci neuronowe, zwłaszcza głębokie, czyli złożone z dużej liczby warstw ukrytych, mają tendencję do przeuczania się przy małej liczbie danych uczących. Nie zawsze jest możliwe powiększenie zbioru uczącego o nowe dane, więc konieczne jest zastosowanie innych technik pozwalających zmniejszyć przeuczenie. Jedną z takich technik jest dropout. Polega on na losowym „wyłączaniu" neuronów podczas kolejnych iteracji (czyli ustawianiu ich wag na 0) oraz skalowaniu niewyłączonych neuronów przez współczynnik $1/(1-p)$, gdzie p to prawdopodobieństwo wyłączenia neuronu. Dzięki temu sieć nie może dopasować się idealnie do danych uczących, ponieważ podczas treningu musi dopasować wagi w taki sposób, by uzyskać dobre rezultaty również wtedy, gdy część neuronów będzie nieaktywna.

Definiując warstwę dropout, należy określić, z jakim prawdopodobieństwem neurony będą „wyłączane". Domyślnie to prawdopodobieństwo wynosi 0,5. Wartość tę można zmienić, ale należy robić to w sposób przemyślany. Jeżeli ustalona zostanie zbyt duża wartość prawdopodobieństwa, to sieć może nie być w stanie nauczyć się rozpoznawać zależności występujących w danych, a tym samym – prowadzić efektywnej klasyfikacji. Jeżeli przyjęta wartość p będzie zbyt mała, to regularyzacja może być nieefektywna i nie doprowadzi do zmniejszenia przeuczenia sieci.

Warstwy dropout powinny być inicjalizowane w funkcji init.

```
class Net(nn.Module):
  def __init__(self):
```

```
        self.fc1 = nn.Linear(2574, 256)
        self.fc2 = nn.Linear(256, 120)
        self.fc3 = nn.Linear(120, 35)
        self.dropout1 = nn.Dropout(p = 0.5)
        self.dropout2 = nn.Dropout(p = 0.2)

    def forward(self, x):

        x = self.fc1(x)
        x = self.dropout1(x)
        x = self.fc2(x)
        x = self.dropout2(x)
        x = self.fc3(x)

        return F.log_softmax(x, dim = 1)
```

Zadanie 17.1. Modyfikacja architektury jednokierunkowej sieci neuronowej służącej do rozpoznawania mowy

W Przykładzie 17.1 została przedstawiona bardzo prosta architektura sieci. Jej użycie nie prowadzi do uzyskania satysfakcjonujących wyników – dokładność klasyfikacji na zbiorze walidacyjnym wynosi 0,33, a na zbiorze testowym 0,34. Spróbuj ją zmodyfikować tak, aby uzyskać większą dokładność klasyfikacji na obu zbiorach. W tym celu możesz wprowadzić zmiany, takie jak:

- dodanie kolejnych warstw (uwaga na liczbę wejść i wyjść),
- dodanie warstw dropout,
- zmiana funkcji aktywacji warstw (wszystkich lub niektórych), np.: $x = F.\text{relu}(\text{self.fc1}(x))$,
- zmiana algorytmu optymalizacji wag.

Po wprowadzeniu zmian zwróć uwagę, czy ustalona liczba epok jest wystarczająca do nauczenia sieci. Jeżeli do ostatniej epoki obserwujesz wzrost dokładności na zbiorze walidacyjnym, to prawdopodobnie sieć nadal nie jest w pełni nauczona i powinno się zwiększyć liczbę epok. Jeżeli od pewnego momentu treningu obserwujesz spadek dokładności, to sieć zaczęła się przeuczać i liczbę epok należy zmniejszyć.

UWAGA

O przeuczeniu sieci można mówić w sytuacji, gdy występuje wyraźny trend spadkowy dokładności klasyfikacji na zbiorze walidacyjnym. Pojedyncze spadki, po których następuje ustabilizowanie wartości metryki lub jej wzrost, są zjawiskiem normalnym.

Dodatek

Definicja D.1. Prawdopodobieństwo warunkowe

Niech (Ω, Σ, P) będzie przestrzenią probabilistyczną oraz $B \in \Sigma$ zdarzeniem takim, że $P(B) > 0$. Wtedy dla każdego zdarzenia $A \in \Sigma$ prawdopodobieństwo warunkowe zajścia zdarzenia A pod warunkiem B określa się wzorem:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}. \tag{D.1}$$

Definicja D.2. Niezależność zdarzeń losowych

Zdarzenia z rodziny $\{A_t, t \in \mathbb{T}\}$ z przestrzeni probabilistycznej (Ω, Σ, P) są niezależne, gdy $\forall n \in \mathbb{N}\ \forall t_1, t_2, ..., t_n \in \mathbb{T}, t_1 \neq ... \neq t_n$ prawdziwy jest wzór:

$$P(A_{t_1}, ..., A_{t_n}) = P(A_{t_1}) \cdot ... \cdot P(A_{t_n}). \tag{D.2}$$

UWAGA

Wprowadzona uogólniona definicja, pozwalająca stwierdzić niezależność dowolnej skończonej liczby n zdarzeń losowych $A_1, ..., A_n$, wymaga sprawdzenia powyższego warunku dla wszystkich dwu-, trzy-, aż do n-elementowych podzbiorów zbioru zdarzeń $A_1, ..., A_n$ (dla podzbiorów jednoelementowych wzór jest trywialny).

UWAGA

Zauważmy, że dla dwóch zdarzeń warunek niezależności upraszcza się do znanej postaci:

$$P(A, B) = P(A) \cdot (B). \tag{D.3}$$

Twierdzenie D.1. O niezależności zdarzeń losowych

Poniższe warunki są równoważne:

i) zdarzenia A i B są niezależne,
ii) $P(A, B) = P(A) \cdot (B)$,
iii) $\forall B : P(B) > 0 : P(A \mid B) = P(A)$,
iv) $\forall A : P(A) > 0 : P(B \mid A) = P(B)$.

Definicja D.3. Warunkowa niezależność zdarzeń

Zdarzenia A i B są warunkowo niezależne, pod warunkiem zdarzenia C, przy czym $P(C) > 0$, gdy prawdziwy jest wzór:

$$P(A \cap B \mid C) = P(A \mid C) \cdot P(B \mid C). \tag{D.4}$$

Własność D.1. Dla zdarzeń warunkowo niezależnych

Jeżeli zdarzenia A i B są warunkowo niezależne pod warunkiem C, oraz $P(B \cap C) > 0$ i $P(A \cap C) > 0$, to:

$$P(A \mid B \cap C) = \frac{P(A \cap B \cap C)}{P(B \cap C)} = \frac{P(A \cap B \mid C) \cdot P(C)}{P(B \mid C) \cdot P(C)} = \frac{P(A \cap B \mid C)}{P(B \mid C)} = \frac{P(A \mid C) \cdot P(B \mid C)}{P(B \mid C)} = P(A \mid C) \quad \text{(D.5)}$$

oraz analogicznie

$$P(B \mid A \cap C) = P(B \mid C). \quad \text{(D.6)}$$

Twierdzenie D.2. Twierdzenie Bayesa

Niech (Ω, Σ, P) będzie przestrzenią probabilistyczną oraz zdarzenia $B_1, B_2, ..., B_n \in \Sigma$ spełniają warunki:

i) $\forall i = 1, ..., nP(B_i) > 0$,
ii) $\forall i \neq jB_i \cap B_j \neq \emptyset$,
iii) $B_1 \cup B_2 \cup ... \cup B_n = \Omega$.

Wtedy dla każdego zdarzenia $A \in \Sigma$ prawdziwy jest wzór:

$$P(B_k \mid A) = \frac{P(A \mid B_k) \cdot P(B_k)}{\sum_{i=1}^{n} P(A \mid B_i) \cdot P(B_i)}. \quad \text{(D.7)}$$

UWAGA

Wzór Bayesa w podstawowej postaci, tj. dla dwóch zdarzeń A i B takich, że $P(A) > 0$, przyjmuje postać:

$$P(B \mid A) = \frac{P(A \mid B) \cdot P(B)}{P(A)}. \quad \text{(D.8)}$$

Własność D.2. Własności macierzy

$$\sum_i z_i^2 = z^T z \quad \text{(D.9)}$$

$$(A - B)^2 = A^2 - AB - BA + B^2 \quad \text{(D.10)}$$

$$\text{tr}(A) = \sum_{i=1}^{n} A_{i,i} \quad \text{(D.11)}$$

$$a \in \mathbb{R} \Rightarrow \text{tr}(a) = a \quad \text{(D.12)}$$

$$\text{tr}(A + B) = \text{tr}(A) + \text{tr}(B) \quad \text{(D.13)}$$

$$\text{tr}(A) = \text{tr}(A^T) \quad \text{(D.14)}$$

$$\text{tr}(aA) = a \cdot \text{tr}(A) \quad \text{(D.15)}$$

$$\nabla_A \text{tr}(AB) = B^T \quad \text{(D.16)}$$

$$\nabla_{A^T} f(A) = (\nabla_A f(A))^T \quad \text{(D.17)}$$

$$\nabla_A \text{tr}(ABA^T C) = CAB + C^T A \quad \text{(D.18)}$$

Definicja D.4. Macierz kowariancji

Macierzą kowariancji wektora losowego $(X_1, X_2, ..., X_d)$ nazywamy symetryczną macierz Σ postaci:

$$\sum = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \dots & \sigma_{1d} \\ \sigma_{21} & \sigma_2^2 & \dots & \sigma_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{d1} & \sigma_{d2} & \dots & \sigma_d^2 \end{bmatrix}, \tag{D.19}$$

gdzie:

$\sigma_{ij} = \operatorname{cov}(X_i, X_j)$,
$\sigma_{ii} = \sigma_i^2$ wariancja zmiennej X_i.

Definicja D.5. Wiarygodność parametru i funkcja wiarygodności

Niech (Ω, Σ, P) będzie przestrzenią probabilistyczną, przy czym P to miara probabilistyczna o parametrach θ, czyli $P_\theta : \theta \in \Theta$, oraz $p(x, \theta)$ oznacza prawdopodobieństwo zmiennej x przy ustalonym parametrze θ. Wtedy dla ustalonego $X \in \Omega$:
$L(\theta, X) = p(X, \theta)$ nazywamy **wiarygodnością parametru** θ,
$L(., X) : \Theta \ni \theta \to L(\theta, X)$ nazywamy **funkcją wiarygodności**.

Definicja D.6. Metryka

Metryką w $\mathbb{R}^n$ nazywamy funkcję $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_+$ przyporządkowującą parze punktów wartość rzeczywistą zwaną odległością tych punktów, która spełnia następujące warunki:

- $\forall x \in \mathbb{R}^n : d(x, y) = 0 \Leftrightarrow x = y$ (zwrotność),
- $\forall x, y \in \mathbb{R}^n : d(x, y) = d(y, x)$ (symetryczność),
- $\forall x, y, z \in \mathbb{R}^n : d(x, y) + d(y, z) \geq d(x, z)$ (warunek trójkąta).

Przykład D.1. Przykłady metryk

Niech $x = (x_1, x_2, ..., x_n), y = (y_1, y_2, ..., y_n) \in \mathbb{R}^n$.

Wtedy:

- metryka dyskretna

$$d(x, y) = \begin{cases} 1, & x \neq y \\ 0, & x = y \end{cases}, \tag{D.20}$$

- metryka euklidesowa

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}, \tag{D.21}$$

- metryka maksimowa

$$d(x, y) = \max_{i=1,2,...,n} |x_i - y_i|, \tag{D.22}$$

- metryka taksówkowa

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|. \tag{D.23}$$

Rozwiązania zadań

Zadanie 8.1

```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import (accuracy_score, recall_score,
                             precision_score, confusion_matrix,
                             classification_report, f1_score)
from sklearn.preprocessing import LabelEncoder
from sklearn.decomposition import PCA

#wczytanie danych i usunięcie pustej kolumny
df = pd.read_csv('data.csv')
df.drop(columns = 'Unnamed: 32',inplace = True)
df.dropna(inplace = True)

column_names = df.columns
data = column_names[2:] #wszystkie kolumny z wyjątkiem id i diagnosis
X = df[data].to_numpy()
y = df['diagnosis'].astype('category').to_numpy()
y = LabelEncoder().fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state = 42,
                                                    stratify = y,
                                                    test_size = 0.2)

#trening na wszystkich trzydziestu cechach
print('Klasyfikacja na podstawie parametrów oryginalnych')
knn = KNeighborsClassifier(n_neighbors = 3, weights = 'uniform')
knn.fit(X_train, y_train)
preds = knn.predict(X_test)

accuracy = accuracy_score(y_test, preds)
recall = recall_score(y_test, preds)
precision = precision_score(y_test, preds)
```

```
f1 = f1_score(y_test, preds)
print('dokładność klasyfikacji = ', accuracy,
      '\n czułość = ', recall,
      '\n precyzja = ', precision,
      '\n F1 = ', f1)

print('macierz pomyłek:\n', confusion_matrix(y_test, preds))

#trening na danych zredukowanych za pomocą PCA
print('Klasyfikacja na podstawie składowych głównych')
pca = PCA(n_components=30).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

knn_pca = KNeighborsClassifier(n_neighbors = 3)
knn_pca.fit(X_train_pca[:,:2], y_train)
preds_pca = knn_pca.predict(X_test_pca[:,:2])

accuracy = accuracy_score(y_test, preds_pca)
recall = recall_score(y_test, preds_pca)
precision = precision_score(y_test, preds_pca)
f1 = f1_score(y_test, preds_pca)
print('dokładność klasyfikacji = ', accuracy,
      '\n czułość = ', recall,
      '\n precyzja = ', precision,
      '\n F1 = ', f1)

print('macierz pomyłek:\n', confusion_matrix(y_test, preds_pca))

#część zmienności wyjaśniana przez poszczególne składowe główne
print(pca.explained_variance_ratio_)
```

Uzyskane wyniki można uznać za zadowalające – przy liczbie sąsiadów wynoszącej trzy uzyskano dokładność klasyfikacji przekraczającą 90% oraz precyzję równą 100%, co oznacza, że wszystkie nowotwory łagodne zostały poprawnie zaklasyfikowane. Mniej satysfakcjonująca jest czułość modelu – wynosi jedynie około 78%, co oznacza, że 12% nowotworów złośliwych zostało uznane za nowotwory łagodne.

Do osiągnięcia dokładności klasyfikacji przekraczającej 0,9 wystarczą dwie pierwsze składowe główne, wyjaśniające łącznie około 99,8% zmienności. Redukcję wymiarowości można uznać za udaną – zmniejszenie liczby cech opisujących obiekt z trzydziestu do zaledwie dwóch nie spowodowało pogorszenia zdolności predykcyjnych modelu.

Zadanie 9.1

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

#wczytanie danych oraz usunięcie kolumn z brakującymi danymi
df = pd.read_csv ('penguins_size.csv')
df.dropna(inplace = True)

#utworzenie wektora zawierającego gatunki występujące w bazie danych
species_categorical = np.unique(df['species'])

#wygenerowanie wektora zawierającego etykiety numeryczne gatunków
species_numerical = df['species'].map(
                           {species_categorical[0]: 0,
                            species_categorical[1]: 1,
                            species_categorical[2]: 2}).to_numpy()

#utworzenie macierzy z danymi, które posłużą do wyznaczenia składowych
#głównych
columns_to_analyze = ['culmen_length_mm', 'culmen_depth_mm',
                      'flipper_length_mm', 'body_mass_g']
data_to_analyze = df[columns_to_analyze]
#standaryzacja danych
scaled_data = StandardScaler().fit_transform(data_to_analyze)
#zastosowanie algorytmu PCA
pca = PCA(n_components = len(columns_to_analyze))
principal_components = pca.fit_transform(data_to_analyze)
principal_components_scaled = pca.fit_transform(scaled_data)

#zmienność wyjaśniana przez poszczególne składowe
print(pca.explained_variance_ratio_)

#wagi poszczególnych parametrów, które służą do wyznaczenia wzorów
#na składowe główne
print(pca.components_)

pca_df = pd.DataFrame(
                 data = (np.hstack((principal_components,
                                    species_numerical[:, np.newaxis]))),
                 columns = ['PC1', 'PC2', 'PC3', 'PC4', 'numer gatunku'])

pca_scaled_df = pd.DataFrame(
                data = (np.hstack((principal_components_scaled,
                                   species_numerical[:, np.newaxis]))),
                columns = ['PC1', 'PC2', 'PC3', 'PC4', 'numer gatunku'])
```

```
#wizualizacja na płaszczyźnie PC1-PC2
#dane nieustandaryzowane
fig = plt.figure(figsize = (10,10))
plt.subplots_adjust(hspace = 0.4) #odległość pomiędzy wykresami
ax = fig.add_subplot(2,1,1)
ax.set_xlabel('Pierwsza składowa główna (PC1)', fontsize = 10)
ax.set_ylabel('Druga składowa główna (PC2)', fontsize = 10)
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2:\ndane
                                                    nieustandaryzowane',
             fontsize = 15)
targets = [0, 1, 2]
colors = ['r', 'g', 'b']
for target, color in zip(targets, colors):
  indicesToKeep = pca_df['numer gatunku'] == target
  ax.scatter(pca_df.loc[indicesToKeep, 'PC1'],
             pca_df.loc[indicesToKeep, 'PC2'],
             c = color,
             s = 50)

ax.legend(species_categorical)
ax.grid()

#dane ustandaryzowane
ax = fig.add_subplot(2,1,2)
ax.set_xlabel('Pierwsza składowa główna (PC1)', fontsize = 10)
ax.set_ylabel('Druga składowa główna (PC2)', fontsize = 10)
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2:\ndane
                                                  ustandaryzowane',
             fontsize = 15)
targets = [0, 1, 2]
colors = ['r', 'g', 'b']
for target, color in zip(targets, colors):
  indicesToKeep = pca_df['numer gatunku'] == target
  ax.scatter(pca_scaled_df.loc[indicesToKeep, 'PC1'],
             pca_scaled_df.loc[indicesToKeep, 'PC2'],
             c = color,
             s = 50)

ax.legend(species_categorical, title = 'Gatunek pingwina')
ax.grid()
plt.show()

#wizualizacja w przestrzeni PC1-PC2-PC3
#dane nieustandaryzowane
fig = plt.figure(figsize = (16,16))
plt.subplots_adjust(hspace = 0.4) #odległość pomiędzy wykresami
```

```
ax = fig.add_subplot(2,1,1,projection = '3d')
ax.set_xlabel('Pierwsza składowa główna\n(PC1)', fontsize = 10)
ax.set_ylabel('Druga składowa główna\n(PC2)', fontsize = 10)
ax.set_zlabel('Trzecia składowa główna\n(PC3)', fontsize = 10)
ax.set_title('Rzut obiektów na przestrzeń PC1-PC2-PC3:\ndane
                                                    nieustandaryzowane',
             fontsize = 15)
targets = [0, 1, 2]
colors = ['r', 'g', 'b']
for target, color in zip(targets, colors):
  indicesToKeep = pca_df['numer gatunku'] == target
  ax.scatter(pca_df.loc[indicesToKeep, 'PC1'],
             pca_df.loc[indicesToKeep, 'PC2'],
             pca_df.loc[indicesToKeep, 'PC3'],
             c = color,
             s = 50)

ax.legend(species_categorical, title = 'Gatunek pingwina')
ax.grid()

#dane ustandaryzowane
ax = fig.add_subplot(2,1,2,projection = '3d')
ax.set_xlabel('Pierwsza składowa główna\n(PC1)', fontsize = 10)
ax.set_ylabel('Druga składowa główna\n(PC2)', fontsize = 10)
ax.set_zlabel('Trzecia składowa główna\n(PC3)', fontsize = 10)
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2-PC3:\ndane
                                                    ustandaryzowane',
             fontsize = 15)
targets = [0, 1, 2]
colors = ['r', 'g', 'b']
for target, color in zip(targets, colors):
  indicesToKeep = pca_df['numer gatunku'] == target
  ax.scatter(pca_scaled_df.loc[indicesToKeep, 'PC1'],
             pca_scaled_df.loc[indicesToKeep, 'PC2'],
             pca_scaled_df.loc[indicesToKeep, 'PC3'],
             c = color,
             s = 50)

ax.legend(species_categorical, title = 'Gatunek pingwina')
ax.grid()
plt.show()
```

Kolejne składowe główne wyjaśniają odpowiednio około 68,64%, 19,45%, 9,22% i 2,69% zmienności. Można je wyznaczyć ze wzorów:

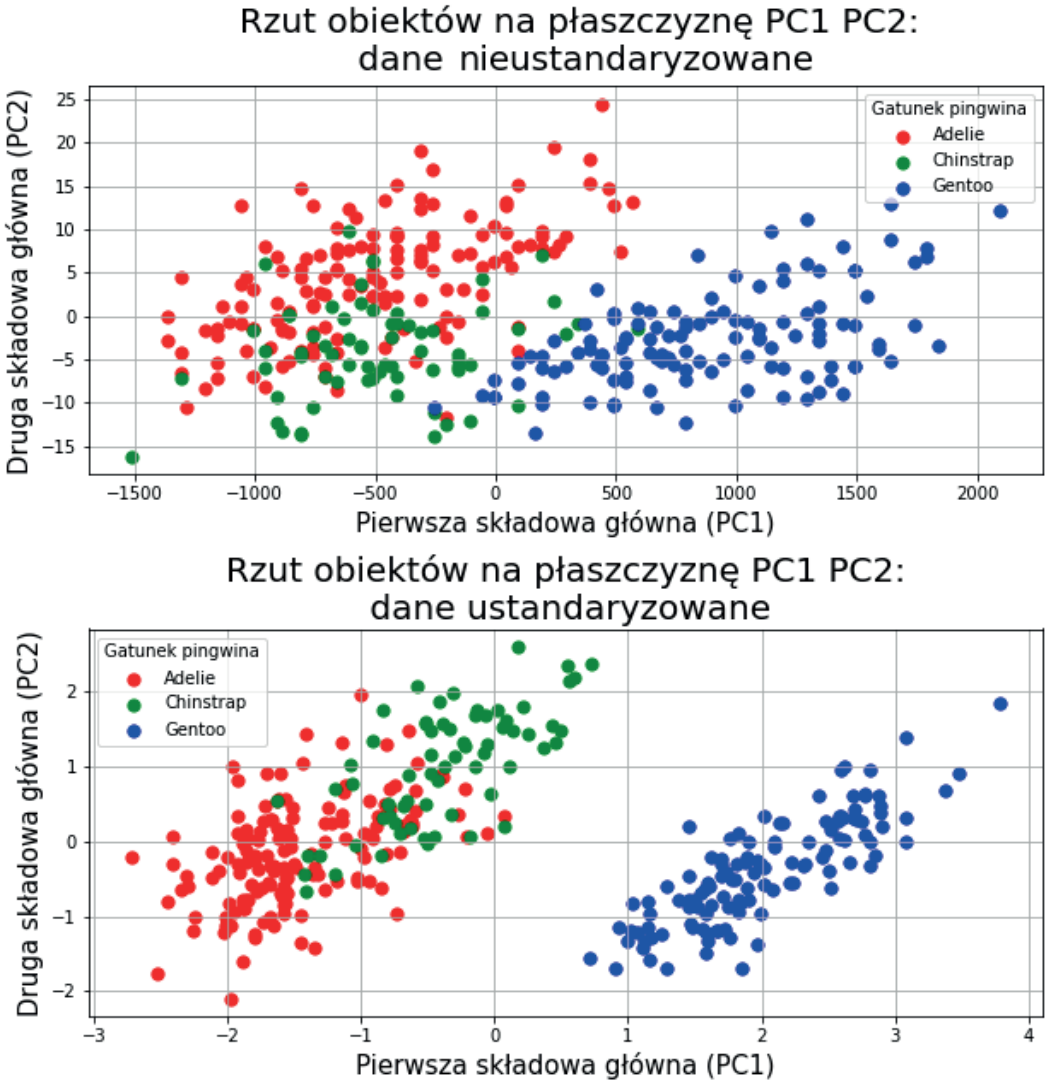
$$PC1 = 0{,}45220241 \cdot \text{culmen\_ lenght\_mm} + (-0{,}39953853) \cdot \text{culmen\_depth\_mm} + 0{,}57678201 \cdot \text{flipper\_length\_mm} + 0{,}54973485 \cdot \text{body\_mass\_g}$$

$$PC2 = 0{,}60201807 \cdot \text{culmen\_lenght\_mm} + 0{,}79485063 \cdot \text{culmen\_depth\_mm} + 0{,}00512057 \cdot \text{flipper\_length\_mm} + 0{,}07589797 \cdot \text{body\_mass\_g}$$

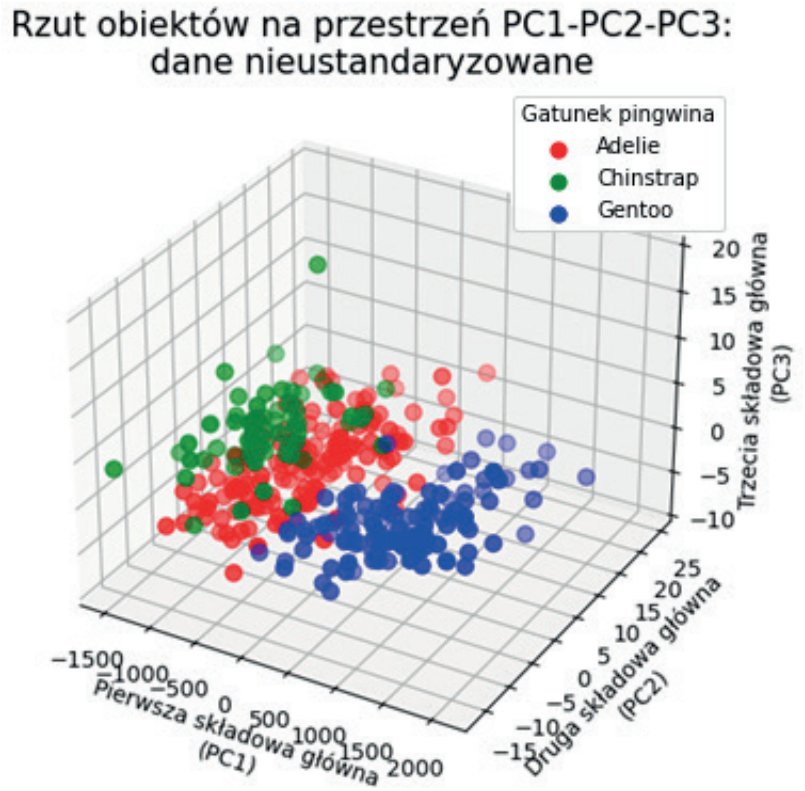
$$PC3 = 0{,}64136461 \cdot \text{culmen\_lenght\_mm} + (-0{,}42778965) \cdot \text{culmen\_depth\_mm} + (-0{,}23691056) \cdot \text{flipper\_length\_mm} + (-0{,}59120287) \cdot \text{body\_mass\_g}$$

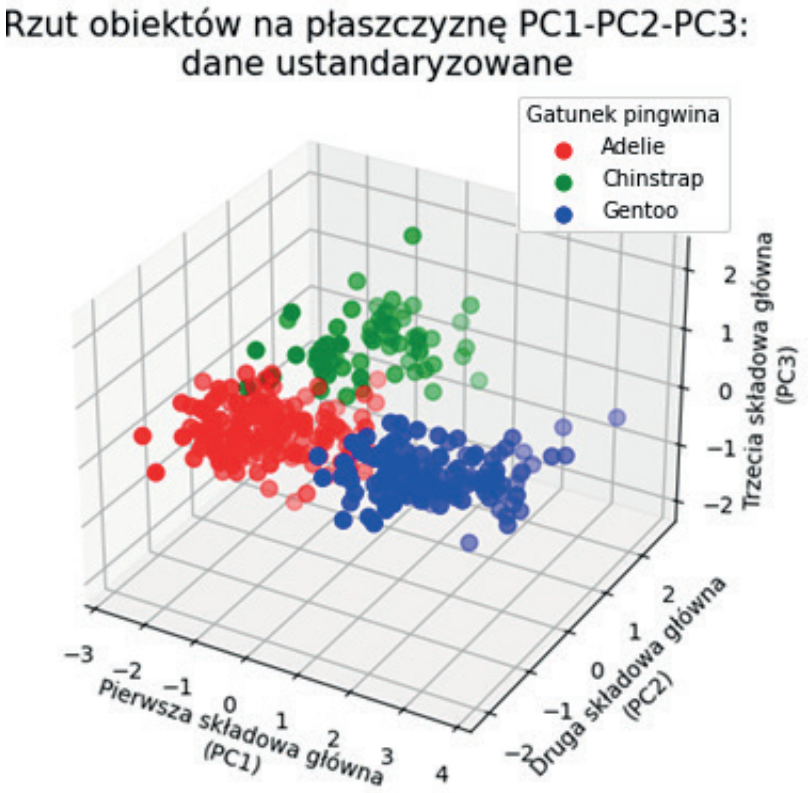
$$PC4 = 0{,}144402294 \cdot \text{culmen\_lenght\_mm} + (-0{,}15992954) \cdot \text{culmen\_depth\_mm} + (-0{,}78177342) \cdot \text{flipper\_length\_mm} + 0{,}58524376 \cdot \text{body\_mass\_g}$$

Wykresy przedstawiające obiekty zrzutowane na płaszczyznę utworzoną przez pierwszą i drugą składową główną przedstawiono na rysunku R.1, zaś wykresy przedstawiające obiekty zrzutowane na przestrzeń utworzoną przez pierwszą, drugą i trzecią składową główną przedstawiono na rysunku R.2.



Rysunek R.1. Wizualizacja danych na płaszczyźnie PC1-PC2





Rysunek R.2. Wizualizacja danych w przestrzeni PC1-PC2-PC3

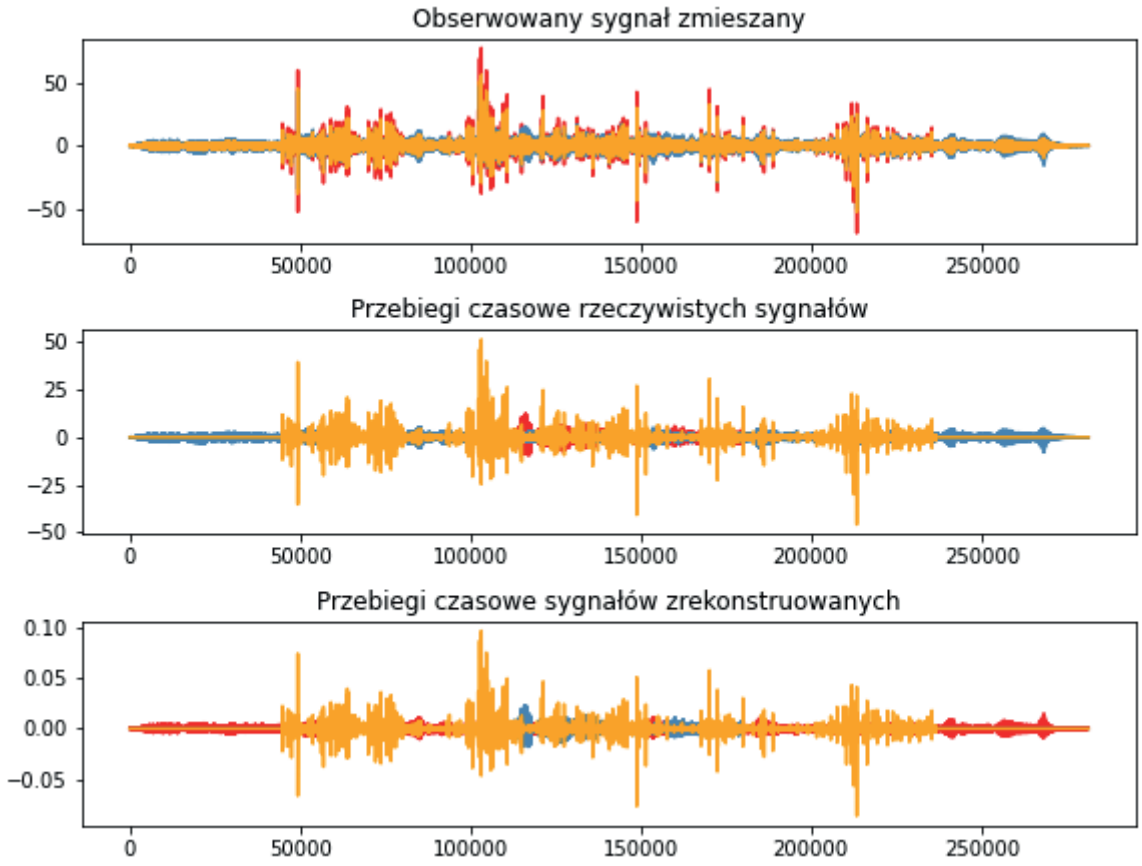
Standaryzacja danych pozwoliła na bardzo dobre wyodrębnienie jednej z klas – po zrzutowaniu danych na płaszczyznę PC1-PC2 klasa oznaczona kolorem niebieskim (gatunek Gentoo) jest wyraźnie oddzielona od pozostałych dwóch klas, natomiast klasy oznaczone kolorem czerwonym i niebieskim (gatunki Adelie i Chinstrap) tworzą skupienia częściowo nachodzące na siebie na płaszczyźnie. W przypadku rzutowania obiektów na przestrzeń PC1-PC2-PC3 wszystkie trzy klasy tworzą skupienia nienachodzące na siebie. W przypadku danych nieustandaryzowanych klasy tworzą mniej wyraźne skupienia i ich odróżnienie od siebie jest trudniejsze.

Zadanie 9.2

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import FastICA
from scipy.io.wavfile import read as read_wav
from sklearn.preprocessing import StandardScaler
from scipy.io.wavfile import write as write_wav
#wczytanie sygnałów
#fs1, fs2, fs3 - częstotliwości próbkowania sygnałów
#s1, s2, s3 - przebiegi czasowe sygnałów
fs1, s1 = read_wav('violin_162.wav')
fs2, s2 = read_wav('violin_0.wav')
fs3, s3 = read_wav('rustle_177.wav')
#sygnały mają różną długość, więc zostaną uzupełnione zerami tak, aby
#wszystkie miały taką samą długość jak najdłuższy z nich
max_len = np.max([len(s1), len(s2), len(s3)])
padded_s1 = np.pad(s1, (np.floor((max_len-len(s1))/2).astype(int),
                        np.ceil((max_len-len(s1))/2).astype(int)),
                       'constant', constant_values = (0))
padded_s2 = np.pad(s2, (np.floor((max_len-len(s2))/2).astype(int),
                        np.ceil((max_len-len(s2))/2).astype(int)),
                       'constant', constant_values = (0))
padded_s3 = np.pad(s3, (np.floor((max_len-len(s3))/2).astype(int),
                        np.ceil((max_len-len(s3))/2).astype(int)),
                       'constant', constant_values = (0))
S = np.stack((padded_s1.astype(float),
              padded_s2.astype(float),
              padded_s3.astype(float))).T
#standaryzacja sygnałów
S = StandardScaler().fit_transform(S) #standaryzacja
#macierz mieszająca
A = np.array([[1, 1, 0.5], [0.5, 2, 1.20], [1.5, 1.0, 1.1]])
X = np.dot(S, A)
```

```
#separacja źródeł dźwięku z sygnału zmieszanego
ica = FastICA(n_components = 3, whiten = True)
S_reconstructed = ica.fit_transform(X) #rekonstrukcja sygnałów
A_reconstructed = ica.mixing_ #estymata macierzy miksowania
assert np.allclose(X, np.dot(S_reconstructed,
                             A_reconstructed.T) + ica.mean_)
#wizualizacja sygnałów
plt.figure()
models = [X, S, S_reconstructed]
names = ['Observations (mixed signal)',
         'True Sources',
         'ICA recovered signals']
colors = ['red', 'steelblue', 'orange']
for ii, (model, name) in enumerate(zip(models, names), 1):
  plt.subplot(4, 1, ii)
  plt.title(name)
  for sig, color in zip(model.T, colors):
    plt.plot(sig, color = color)
plt.tight_layout()
plt.show()
#przeskalowanie amplitud sygnałów
S1_reconstructed = S_reconstructed[:,0] * (
                        2 ** 15 - 1)/np.max(np.abs(S_reconstructed[:,0]))
S2_reconstructed = S_reconstructed[:,1] * (
                        2 ** 15 - 1)/np.max(np.abs(S_reconstructed[:,1]))
S3_reconstructed = S_reconstructed[:,2] * (
                        2 ** 15 - 1)/np.max(np.abs(S_reconstructed[:,2]))
#zapis zrekonstruowanych sygnałów do plików
write_wav("source1.wav", fs1, S1_reconstructed.astype(np.int16))
write_wav("source2.wav", fs1, S2_reconstructed.astype(np.int16))
write_wav("source3.wav", fs1, S3_reconstructed.astype(np.int16))
#odtworzenie wygenerowanych plików .wav w Pythonie
from pydub import AudioSegment
from pydub.playback import play
sound = AudioSegment.from_wav("source1.wav")
play(sound)
sound = AudioSegment.from_wav("source2.wav")
play(sound)
sound = AudioSegment.from_wav("source3.wav")
play(sound)
```

Analiza wykresów przebiegów czasowych sygnałów oryginalnych oraz zrekonstruowanych przedstawionych na rysunku R.3 pozwala stwierdzić, że rekonstrukcja przebiegła prawidłowo. Widoczne są różnice w przebiegach czasowych, jednakże są one niewielkie. Do podobnych wniosków można dojść, odsłuchując oryginalne nagrania, z których utworzono miks, oraz sygnały uzyskane przez zastosowanie algorytmu ICA – różnice są słyszalne, ale niewielkie i jakość sygnałów zrekonstruowanych jest wysoka.



Rysunek R.3. Przebiegi czasowe sygnałów zmieszanych (na górze), rzeczywistych sygnałów ukrytych (w środku) oraz sygnałów ukrytych zrekonstruowanych przy użyciu algorytmu ICA (na dole)

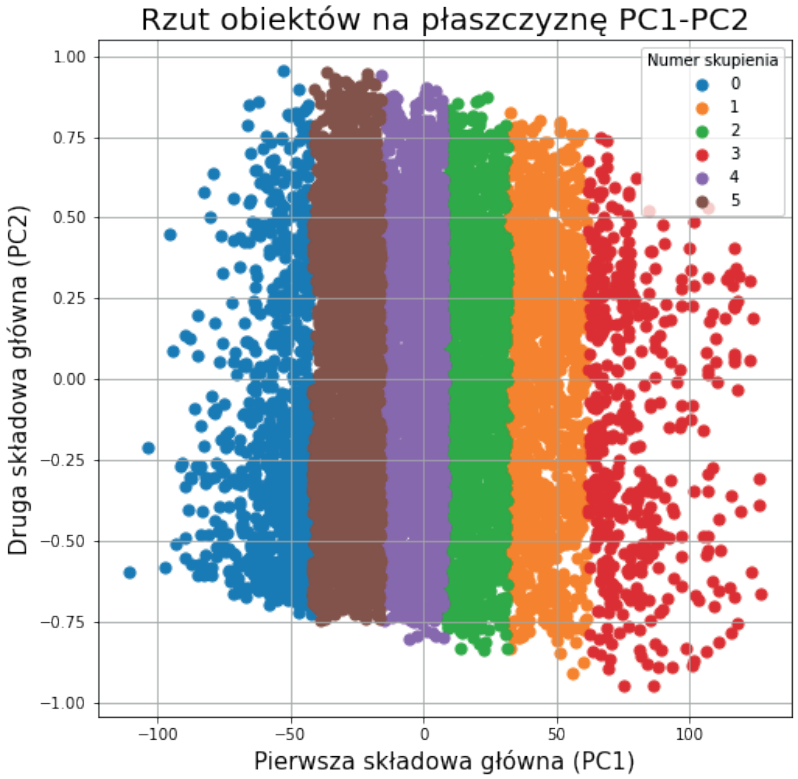
Zadanie 10.1

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

feats_col_list = [0,1,2,3,4,5,6,7,8]
feats_df = pd.read_csv("fma_metadata/echonest.csv",
                       usecols = feats_col_list,
                       low_memory = False,
                       header = 2)

feats_df.rename(columns = {'Unnamed: 0': 'track_id'}, inplace = True)

feats_df. dropna(inplace = True)
feats_df.reset_index(drop = True, inplace = True)
```

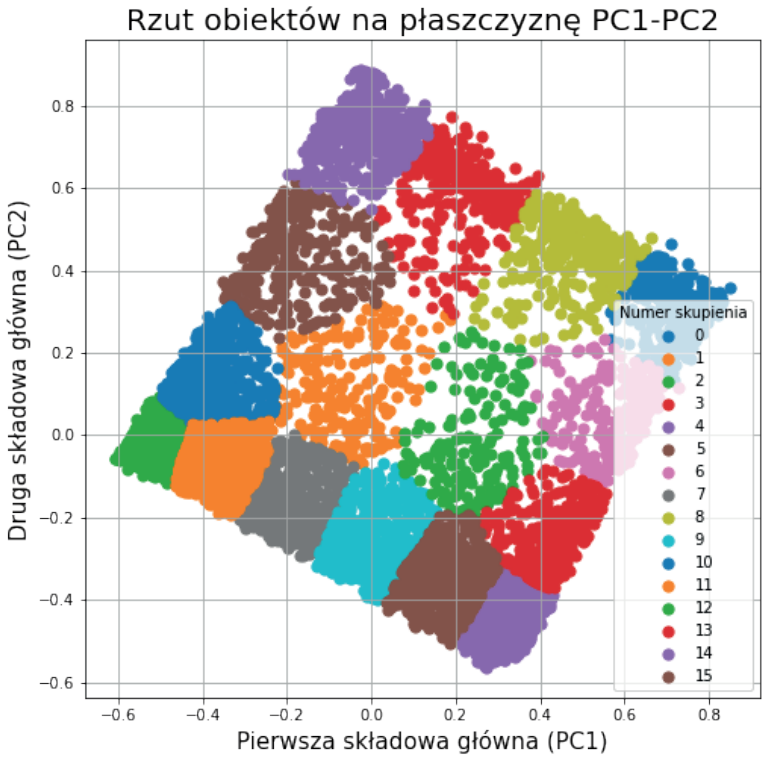
```
genre_col_list = [0,40]
genre_df = pd.read_csv("fma_metadata/tracks.csv",
                       usecols = genre_col_list,
                       low_memory = False,
                       header = 1)
genre_df.rename(columns = {'Unnamed: 0': 'track_id'}, inplace = True)
genre_df.dropna(inplace = True)
genre_df.reset_index(drop = True, inplace = True)

labels_dict = {value: index for index, value in enumerate(
  genre_df["genre_top"].unique())}

genre_df['genre_numeric_label'] = [labels_dict[genre_df[
          "genre_top"][i]] for i in genre_df.index]

all_data_df = pd.merge(feats_df, genre_df, on = 'track_id')

#zmniejszenie liczby klastrów do sześciu
pca = PCA(n_components = 8, random_state = 0)

feats_names = ["acousticness", "danceability", "energy",
               "instrumentalness", "liveness", "speechiness",
               "tempo", "valence"]

PCA_feats = pca.fit_transform(all_data_df[feats_names])
all_data_df['PC1'] = PCA_feats[:,0]
all_data_df['PC2'] = PCA_feats[:,1]
kmeans = KMeans(n_clusters = 6, random_state = 0, n_init = 10)
kmeans_labels1 = kmeans.fit_predict(PCA_feats[:,:2])

fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pierwsza składowa główna (PC1)', fontsize = 15)
ax.set_ylabel('Druga składowa główna (PC2)', fontsize = 15)
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2',
                                        fontsize = 20)targets = range(0,6)
for target in targets:
  indicesToKeep = kmeans_labels1 == target
  ax.scatter(all_data_df.loc[indicesToKeep, 'PC1'],
             all_data_df.loc[indicesToKeep, 'PC2'],
             s = 50)
ax.legend(targets, title = 'Numer skupienia')
ax.grid()
plt.show()
```

```
#zmniejszenie liczby analizowanych cech akustycznych, analizy będą
#prowadzone z wykorzystaniem tylko czterech z ośmiu dostępnych cech
pca = PCA(n_components = 4)
feats_names = [“acousticness”, “danceability”,
               “instrumentalness”, “speechiness”]

PCA_feats = pca.fit_transform(all_data_df[feats_names])
print(pca.explained_variance_ratio_)
all_data_df[‘PC1’] = PCA_feats[:,0]
all_data_df[‘PC2’] = PCA_feats[:,1]

kmeans = KMeans(n_clusters = 16, random_state = 0, n_init = 10)
kmeans_labels2 = kmeans.fit_predict(PCA_feats[:,:2])

fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel(‘Pierwsza składowa główna (PC1)’, fontsize = 15)
ax.set_ylabel(‘Druga składowa główna (PC2)’, fontsize = 15)
ax.set_title(‘Rzut obiektów na płaszczyznę PC1-PC2’,
                                        fontsize = 20)targets = range(0,6)
targets = range(0,16)
for target in targets:
  indicesToKeep = kmeans_labels2 == target
  ax.scatter(all_data_df.loc[indicesToKeep, ‘PC1’],
             all_data_df.loc[indicesToKeep, ‘PC2’],
             s = 50)
ax.legend(targets, title = ‘Numer skupienia’)
ax.grid()
plt.show()

#zmniejszenie liczby analizowanych cech akustycznych oraz liczby tworzonych
#klastrów, analizy będą prowadzone z wykorzystaniem tylko czterech
#z ośmiu dostępnych cech, równocześnie liczba klastrów zostaje zmniejszona
#do sześciu
pca = PCA(n_components = 4)
feats_names = [“acousticness”, “danceability”,
               “instrumentalness”, “speechiness”]

PCA_feats = pca.fit_transform(all_data_df[feats_names])
print(pca.explained_variance_ratio_)
all_data_df[‘PC1’] = PCA_feats[:,0]
all_data_df[‘PC2’] = PCA_feats[:,1]

kmeans = KMeans(n_clusters = 6, random_state = 0, n_init = 10)
kmeans_labels3 = kmeans.fit_predict(PCA_feats[:,:2])
```

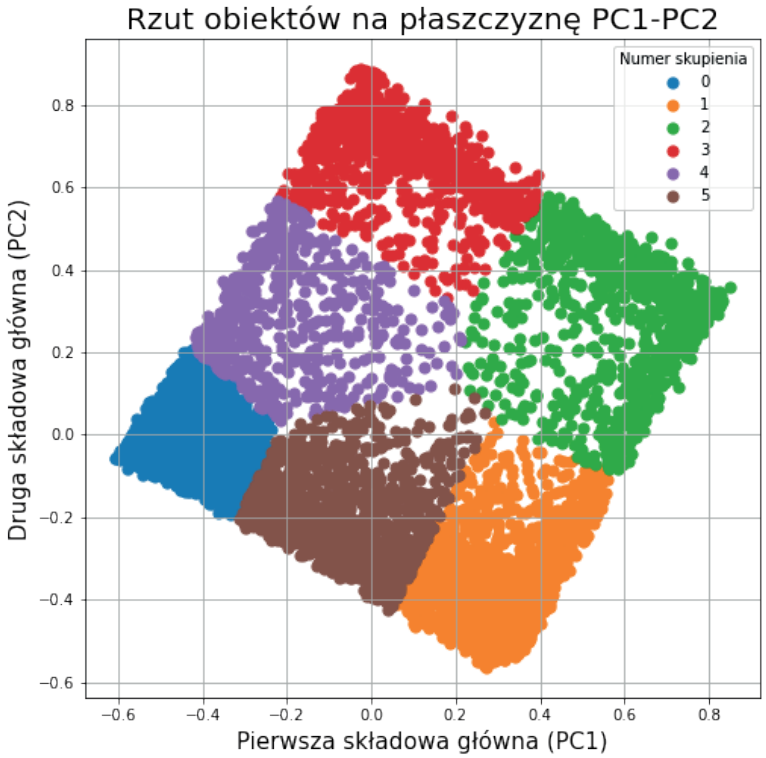
```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pierwsza składowa główna (PC1)', fontsize = 15)
ax.set_ylabel('Druga składowa główna (PC2)', fontsize = 15)
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2',
                                        fontsize = 20)targets = range(0,6)
targets = range(0,6)
for target in targets:
  indicesToKeep = kmeans_labels3 == target
  ax.scatter(all_data_df.loc[indicesToKeep, 'PC1'],
             all_data_df.loc[indicesToKeep, 'PC2'],
             s = 50)
ax.legend(targets, title = 'Numer skupienia')
ax.grid()
plt.show()

#klasteryzacja przeprowadzona na danych ustandaryzowanych przykład z użyciem
#wszystkich ośmiu cech akustycznych liczba wyznaczonych klastrów wynosi 16
feats_names = ["acousticness", "danceability", "energy",
               "instrumentalness","liveness", "speechiness", "tempo",
               "valence"]

scaled_feats = StandardScaler().fit_transform(all_data_df[feats_names])

pca = PCA(n_components = 8,random_state = 0)
PCA_feats = pca.fit_transform(scaled_feats)
print(pca.explained_variance_ratio_)
all_data_df['PC1'] = PCA_feats[:,0]
all_data_df['PC2'] = PCA_feats[:,1]

kmeans = KMeans(n_clusters = 16, random_state = 0, n_init = 10)
kmeans_labels4 = kmeans.fit_predict(PCA_feats[:,:2])

fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Pierwsza składowa główna (PC1)', fontsize = 15)
ax.set_ylabel('Druga składowa główna (PC2)', fontsize = 15)
ax.set_title('Rzut obiektów na płaszczyznę PC1-PC2',
                                        fontsize = 20)targets = range(0,6)
targets = range(0,16)
for target in targets:
  indicesToKeep = kmeans_labels4 == target
  ax.scatter(all_data_df.loc[indicesToKeep, 'PC1'],
             all_data_df.loc[indicesToKeep, 'PC2'],
             s = 50)
ax.legend(targets, title = 'Numer skupienia')
ax.grid()
plt.show()
```



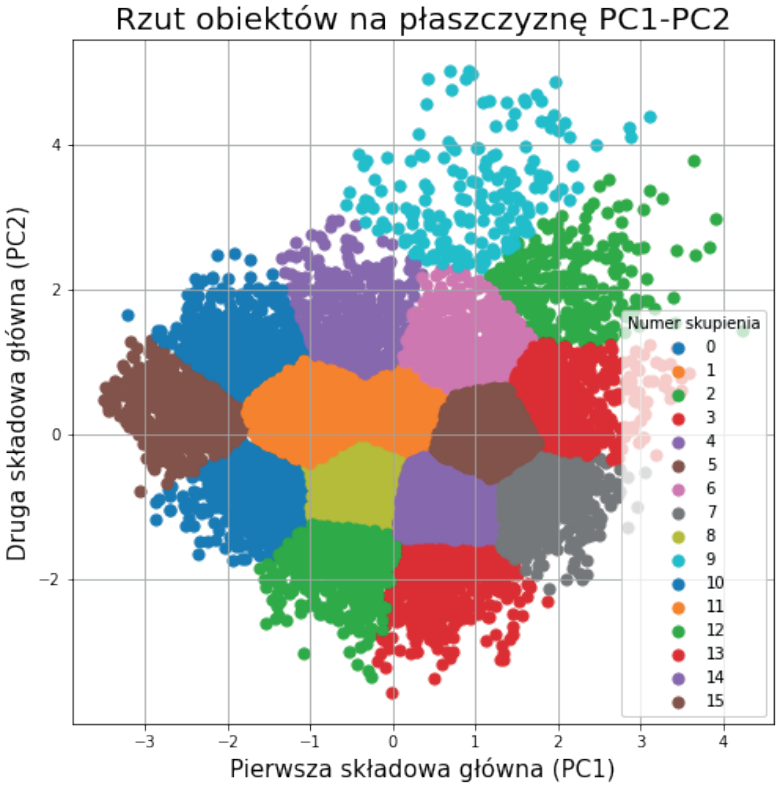
Rysunek R.4. Skupienia zawierające utwory muzyczne wyznaczone metodą *k*-średnich przedstawione na płaszczyźnie utworzonej przez dwie pierwsze składowe główne – podział danych na sześć skupień



Rysunek R.5. Skupienia zawierające utwory muzyczne wyznaczone metodą *k*-średnich przedstawione na płaszczyźnie utworzonej przez dwie pierwsze składowe główne – skupienia wyznaczone na podstawie czterech cech akustycznych



Rysunek R.6. Skupienia zawierające utwory muzyczne wyznaczone metodą *k*-średnich przedstawione na płaszczyźnie utworzonej przez dwie pierwsze składowe główne – podział danych na sześć skupień wyznaczonych na podstawie czterech cech akustycznych



Rysunek R.7. Skupienia zawierające utwory muzyczne wyznaczone metodą *k*-średnich przedstawione na płaszczyźnie utworzonej przez dwie pierwsze składowe główne – skupienia wyznaczone na podstawie danych ustandaryzowanych

Standaryzacja dość istotnie wpłynęła na uzyskane wyniki – wykresy przedstawiające obiekty rzutowane na płaszczyznę PC1-PC2 (rys. R.4–R.7) istotnie różnią się od siebie w zależności od tego, czy przeprowadzono standaryzację danych, czy nie. Przede wszystkim można zauważyć, że po ustandaryzowaniu danych pierwsza składowa wyjaśnia jedynie 22,85% zmienności, podczas gdy bez standaryzacji wyjaśniała ponad 99% zmienności. Po ustandaryzowaniu danych klastry nie są więc wydzielone na podstawie tylko jednej składowej i na wykresie nie układają się w pionowe pasy.

Zadanie 10.2

Jeżeli istotne jest zachowanie interpretowalności wyników, to lepszym rozwiązaniem jest klasteryzacja na podstawie oryginalnych cech akustycznych sygnału, a nie składowych głównych – łatwiej jest zrozumieć podział dokonany na podstawie np. tempa utworu i jego energii niż sum ważonych tych cech.

Zadanie 11.1

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram

feats_col_list = [0,1,2,3,4,5,6,7,8]
feats_df = pd.read_csv("fma_metadata/echonest.csv",
    usecols = feats_col_list, low_memory = False, header = 2)
feats_df.rename(columns = {'Unnamed: 0': 'track_id'}, inplace = True)
feats_df. dropna(inplace = True)
feats_df.reset_index(drop = True, inplace = True)

genre_col_list = [0,40]
genre_df = pd.read_csv("fma_metadata/tracks.csv",
    usecols = genre_col_list, low_memory = False, header = 1)
genre_df.rename(columns = {'Unnamed: 0': 'track_id'}, inplace = True)
genre_df.dropna(inplace = True)
genre_df.reset_index(drop = True, inplace = True)

labels_dict = {value: index for index,
               value in enumerate(genre_df["genre_top"].unique())}
genre_df['genre_numeric_label'] = [labels_dict[genre_df[
                                  "genre_top"][i]] for i in genre_df.index]

all_data_df = pd.merge(feats_df, genre_df, on = 'track_id')

feats_names = ["acousticness", "danceability", "energy",
               "instrumentalness", "liveness", "speechiness",
               "tempo", "valence"]
```

```
scaled_feats = StandardScaler().fit_transform( all_data_df[feats_names])

#Część pierwsza zadania: sprawdzenie, do jakich skupień zostały
#przyporządkowane utwory z wybranego gatunku.
model = AgglomerativeClustering(n_clusters = 4, affinity = 'euclidean')
clusters = model.fit_predict(scaled_feats)
dendrogram_clusters_df = pd.DataFrame()
dendrogram_clusters_df['track_id'] = all_data_df["track_id"]
dendrogram_clusters_df['real genre'] = all_data_df["genre_top"]
dendrogram_clusters_df['dendrogram clusters'] = clusters
print(dendrogram_clusters_df.loc[dendrogram_clusters_df[
                                                'real genre'] == 'Blues'])

#Część druga zadania: sprawdzenie, jakie gatunki zostały przyporządkowane
#do każdego z czterech skupień i jaka jest ich liczba
from collections import Counter

klaster_0 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 0][
  'real genre']
print(Counter(klaster_1))

klaster_1 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 1][
  'real genre']
print(Counter(klaster_2))

klaster_2 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 2][
  'real genre']
print(Counter(klaster_3))

klaster_3 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 3][
  'real genre']
print(Counter(klaster_4))

#Podział nie jest równomierny. Nie jest co prawda tak, że wszystkie
#utwory z danego gatunku zostały przyporządkowane do tego samego klastra,
#jednak można zauważyć, że większość znajduje się w jednym klastrze,
#np. rock został podzielony w ten sposób: klaster_0: 2950, klaster_1: 637,
#klaster_2: 117, klaster_3: 188.
#W przypadku pozostałych gatunków wygląda to podobnie. Widać też, że
#nie wszystkie gatunki znajdują się w każdym klastrze, np. żaden utwór
#z gatunku experimental nie został przydzielony do klastra 3,
#a w klastrze 2 znalazł się tylko jeden utwór z tego gatunku.
```

Utwory należące do poszczególnych gatunków muzycznych nie zostały rozdzielone równomienie pomiędzy utworzone skupienia, przykładowo 2950 spośród 3892 utworów należących do gatunku rock zostało przydzielone do tego samego skupienia. Analizując uzyskane wyniki, można również zauważyć, że utwory należące do gatunku muzyki eksperymentalnej (experimental) zostały przydzielone tylko do trzech skupień. Co więcej, w jednym skupieniu znalazł się tylko jeden utwór tego gatunku.

Zadanie 11.2

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram

feats_col_list = [0,1,2,3,4,5,6,7,8]
feats_df = pd.read_csv("fma_metadata/echonest.csv",
    usecols = feats_col_list, low_memory = False, header = 2)
feats_df.rename(columns = {'Unnamed: 0': 'track_id'}, inplace = True)
feats_df. dropna(inplace = True)
feats_df.reset_index(drop = True, inplace = True)

genre_col_list = [0,40]
genre_df = pd.read_csv("fma_metadata/tracks.csv",
    usecols = genre_col_list, low_memory = False, header = 1)
genre_df.rename(columns = {'Unnamed: 0': 'track_id'}, inplace =   True)
genre_df.dropna(inplace = True)
genre_df.reset_index(drop = True, inplace = True)

labels_dict = {value: index for index, value in enumerate(
  genre_df["genre_top"].unique())}
genre_df['genre_numeric_label'] = [labels_dict[genre_df[
                                  "genre_top"][i]] for i in genre_df.index]

all_data_df = pd.merge(feats_df, genre_df, on = 'track_id')

feats_names = ["acousticness", "danceability", "energy",
               "instrumentalness", "liveness", "speechiness",
               "tempo", "valence"]

scaled_feats = StandardScaler().fit_transform(all_data_df[feats_names])

#Przykładowe porównanie wyników:
# 1) odległość euklidesowa, metoda wykorzystująca średnią odległość
```

```
model = AgglomerativeClustering(n_clusters = 4,
                                linkage = 'average',
                                affinity = 'euclidean')
clusters = model.fit_predict(scaled_feats)
dendrogram_clusters_df = pd.DataFrame()
dendrogram_clusters_df['track_id'] = all_data_df["track_id"]
dendrogram_clusters_df['real genre'] = all_data_df["genre_top"]
dendrogram_clusters_df['dendrogram clusters'] = clusters
print(dendrogram_clusters_df.loc[dendrogram_clusters_df[
                                              'real genre'] == 'Blues'])

klaster_0 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 0]['real genre']
print(Counter(klaster_0))

klaster_1 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 1]['real genre']
print(Counter(klaster_1))

klaster_2 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 2]['real genre']
print(Counter(klaster_2))

klaster_3 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 3]['real genre']
print(Counter(klaster_3))

# 2) odległość euklidesowa, metoda wykorzystująca największą odległość

model = AgglomerativeClustering(n_clusters = 4, linkage = 'complete',
                                              affinity = 'euclidean')
clusters = model.fit_predict(scaled_feats)
dendrogram_clusters_df = pd.DataFrame()
dendrogram_clusters_df['track_id'] = all_data_df["track_id"]
dendrogram_clusters_df['real genre'] = all_data_df["genre_top"]
dendrogram_clusters_df['dendrogram clusters'] = clusters
print(dendrogram_clusters_df.loc[dendrogram_clusters_df[
                                              'real genre'] == 'Blues'])

klaster_0 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 0]['real genre']
print(Counter(klaster_0))

klaster_1 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 1]['real genre']
print(Counter(klaster_1))
```

```
klaster_2 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 2]['real genre']
print(Counter(klaster_2))

klaster_3 = dendrogram_clusters_df.loc[
  dendrogram_clusters_df['dendrogram clusters'] == 3]['real genre']
print(Counter(klaster_3))
```

Wybór metryki oraz metody tworzenia skupień mają istotny wpływ na wyniki klasteryzacji. Przykładowo, stosując odległość euklidesową i metodę wykorzystującą średnią odległość od obiektów znajdujących się w danym skupieniu, utwory należące do gatunku rock zostały przydzielone do czterech skupień w następujący sposób: klaster_0: 3844, klaster_1: 10, klaster_2: 3, klaster_3: 35. Natomiast stosując odległość euklidesową oraz metodę wykorzystującą największą odległość od obiektów należących do poszczególnych skupień, uzyskano następujący podział: klaster_0: 3583, klaster_1: 212, klaster_2: 69, klaster_3: 28. Widać więc, że liczba obiektów przyporządkowanych do utworzonych skupień znacząco się różni.

UWAGA

Kolejność tworzonych klastrów jest losowa, więc numery skupień, do których zostały przyporządkowane obiekty, mogą być inne przy każdym uruchomieniu kodu, jednak liczba obiektów znajdujących się w jednym skupieniu powinna pozostać taka sama.

Zadanie 12.1

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import (mean_absolute_error,
                             explained_variance_score,
                             max_error)

df = pd.read_csv('Real estate.csv')
print(df.columns)

#do zmiennej X przypisane będą wszystkie kolumny z wyjątkiem pierwszej,
#zawierającej numer porządkowy oraz ostatniej, zawierającej cenę
#nieruchomości
X_col_names = df.columns[1:-1]
X = df[X_col_names]

#do zmiennej y zostanie przypisana ostatnia kolumna,zawierająca cenę
#nieruchomości
y = df[-1]

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state = 42,
                                                    test_size = 0.2)
```

```
#jeżeli dane mają być użyte przez model regresyjny, nie należy
#przeprowadzać stratyfikacji

reg = LinearRegression()
reg.fit(X_train, y_train)
preds = reg.predict(X_test)
print(mean_absolute_error(y_test, preds))
print(max_error(y_test, preds))
```

Jeżeli random_state = 42, to wartość średniego błędu wynosi około 5,31, natomiast największy popełniony błąd wynosi 27,63.

Używając funkcji LinearRegression, nie można przeprowadzić optymalizacji hiperparametrów modelu – jest to model wykorzystujący podstawową wersję metody najmniejszych kwadratów i nie ma żadnych hiperparametrów. Optymalizować można inne modele liniowe zaimplementowane w bibliotece scikit-learn. Ich listę można znaleźć pod adresem: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model.

Zadanie 12.2

Ekstrakcja cech:

```
import librosa
import opensmile
from pathlib import Path
import numpy as np
from scipy.io.wavfile import read as read_wav

#pobranie ścieżki dostępu do aktualnego folderu roboczego
path = Path.cwd()
#używając biblioteki pathlib ścieżki dostępu łączy się za pomocą /
#jeżeli folder roboczy jest zdefiniowany jako zwykły string, to ścieżkę
#do folderu z danymi należy do niego dołączyć znakiem +
path = path / 'ND357A_24bit_cut_ALL/cut_norm/'

smile = opensmile.Smile(
                    feature_set = opensmile.FeatureSet.ComParE_2016,
                    feature_level = opensmile.FeatureLevel.Functionals,)

opensmile_feats = []
fund_freqs = []

fmin = librosa.note_to_hz('C2')
fmax = librosa.note_to_hz('C7')

for file in os.listdir(path):
  if file.endswith('.wav'):
    fs, signal = read_wav(path/file)
    #wyznaczenie częstotliwości tonu podstawowego
```

```
 f0, voiced_flag, voiced_probs = librosa.pyin(signal.astype(float),
                                              fmin,
                                              fmax,
                                              sr = fs)
     fund_freqs.append(np.round(np.mean(f0[voiced_flag == True])))
     #wyznaczenie cech akustycznych sygnału za pomocą biblioteki openSMILE
    y = smile.process_file(path/file)
    opensmile_feats.append(y)

#zamiana list na tablice
opensmile_feats = np.asarray(opensmile_feats)
fund_freqs = np.asarray(fund_freqs).astype(int)

#zapisanie danych do plików
np.save('opensmile_feats', opensmile_feats)
np.save('fundamental_frequencies', fund_freqs)
```

Trening i ewaluacja modeli regresyjnych:

```
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.metrics import (mean_absolute_error, make_scorer,
                             mean_squared_error)
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
import matplotlib.pyplot as plt
from sklearn.svm import SVC
import optuna
from sklearn.model_selection import cross_validate, KFold
import pickle
from sklearn.pipeline import make_pipeline

X = np.load('opensmile_feats.npy')
y = np.load('fundamental_frequencies.npy')

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state = 42)

#standaryzacja danych
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

#redukcja wymiarowości - metoda selekcji cech SelectKBest
liczba_cech = 100
```

```
selector = SelectKBest(k = liczba_cech)
X_train_dim_reducted = selector.fit_transform(X_train_scaled, y_train)
X_test_dim_reducted = selector.transform(X_test_scaled)

#trening modelu z domyślnymi hiperparametrami i ustaloną liczbą cech
ridge_reg = Ridge()
ridge_reg.fit(X_train_dim_reducted, y_train)
preds = ridge_reg.predict(X_test_dim_reducted)
print('test MAE: ', mean_absolute_error(y_test, preds))
print('test MSE: ', mean_squared_error(y_test, preds))

#optymalizacja modelu - użycie metryki MAE
scoring_mae = {'mae': make_scorer(mean_absolute_error)}

def objective_mae(trial, model, space, X, y):
  model_space = get_space(trial)

  mdl = model(**model_space)
  scores = cross_validate(mdl, X, y, scoring = scoring_mae,
                                     cv = KFold(n_splits = 5),
                                     return_train_score = True)

  return np.mean(scores['test_mae'])

model = Ridge

def get_space(trial):
  space = {"alpha": trial.suggest_uniform("alpha", 0, 5),
           'max_iter': trial.suggest_int('max_iter', 1000, 1000),
           "solver": trial.suggest_categorical("solver",
                                     ['auto', 'svd', 'cholesky', 'lsqr',
                                     'sparse_cg', 'sag', 'saga'])}
  return space

trials = 100

study = optuna.create_study(direction = 'minimize')
study.optimize(lambda x: objective_mae(x, model, get_space,
                                                 X_train_dim_reducted,
                                                 y_train),
               n_trials = trials)

print('params: ', study.best_params)

ridge_reg = model(**study.best_params)
ridge_reg.fit(X_train_dim_reducted, y_train)
preds = ridge_reg.predict(X_test_dim_reducted)
pickle.dump(ridge_reg, open('ridge_F0_model_mae', 'wb+'))
```

```
print('test MAE: ', mean_absolute_error(y_test, preds))
print('test MSE: ', mean_squared_error(y_test, preds))

#optymalizacja modelu - użycie metryki MSE
scoring_mse = {'mse': make_scorer(mean_squared_error)}

def objective_mse(trial, model, space, X, y):
  model_space = get_space(trial)

  mdl = model(**model_space)
  scores = cross_validate(mdl, X, y, scoring = scoring_mse,
                                     cv = KFold(n_splits = 5),
                                     return_train_score = True)

  return np.mean(scores['test_mse'])

model = Ridge

def get_space(trial):
  space = {"alpha": trial.suggest_uniform("alpha", 0, 5),
           'max_iter': trial.suggest_int('max_iter', 1000, 1000),
           "solver": trial.suggest_categorical("solver",
                                     ['auto', 'svd', 'cholesky','lsqr',
                                      'sparse_cg', 'sag', 'saga'])}
  return space

trials = 100

study = optuna.create_study(direction = 'minimize')
study.optimize(lambda x: objective_mse(x, model, get_space,
                                            X_train_dim_reducted,
                                            y_train),
               n_trials = trials)
print('params: ', study.best_params)

ridge_reg = model(**study.best_params)
ridge_reg.fit(X_train_dim_reducted, y_train)
preds = ridge_reg.predict(X_test_dim_reducted)
pickle.dump(ridge_reg, open('ridge_F0_model_mse', 'wb+'))
print('test MAE: ', mean_absolute_error(y_test, preds))
print('test MSE: ', mean_squared_error(y_test, preds))

#optymalizacja modelu oraz liczby cech użytych do treningu
scoring_mse = {'mse': make_scorer(mean_squared_error)}

def objective_mse(trial, model, space, X, y):
  model_space = get_space(trial)
```

```
  sel = SelectKBest(k = model_space["k"])
  del model_space["k"]
  mdl = model(**model_space)
  pipeline = make_pipeline(sel, mdl)
  scores = cross_validate(pipeline, X, y, scoring = scoring_mse,
                                          cv = KFold(n_splits = 5),
                                          return_train_score = True)

  return np.mean(scores['test_mse'])

model = Ridge

def get_space(trial):
  space = {"k": trial.suggest_int("k", 10, 200),
           "alpha": trial.suggest_uniform("alpha", 0, 5),
           'max_iter': trial.suggest_int('max_iter', 100, 100),
           "solver": trial.suggest_categorical("solver",
                                  ['auto', 'svd', 'cholesky', 'lsqr',
                                   'sparse_cg', 'sag', 'saga'])}
  return space

trials = 500 #liczba prób

study = optuna.create_study(direction = 'minimize')
study.optimize(lambda x: objective_mse(x, model, get_space,
                                       X_train_scaled,
                                       y_train),
               n_trials = trials)

print('params: ', study.best_params)
best_params = study.best_params
sel = SelectKBest(k = best_params["k"])
del best_params['k']

ridge_reg = model(**best_params)
pipeline = make_pipeline(sel, ridge_reg)
pipeline.fit(X_train_scaled, y_train)
preds = pipeline.predict(X_test_scaled)
pickle.dump(pipeline, open('ridge_F0_model_mse_kbest', 'wb+'))
print('test MAE: ', mean_absolute_error(y_test, preds))
print('test MSE: ', mean_squared_error(y_test, preds))
```

Metryka użyta do ewaluacji modeli w trakcie optymalizacji nie ma wpływu na uzyskane wyniki – w obu przypadkach model mylił się o około 37 Hz, wartość błędu średniokwadratowego również jest taka sama. Poprawę wyników spowodowało natomiast zoptymalizowanie liczby cech wybieranych w wyniku selekcji metodą SelectKBest. Optymalna liczba cech wynosi 197, czyli niemal dwukrotnie więcej niż zastosowane wcześniej 100. Model trenowany z użyciem 197 cech uzyskał wartość MAE wynoszącą około 28 Hz, a więc prawie o 10 Hz mniejszą niż przy zastosowaniu mniejszej liczby cech.

Zadanie 14.1

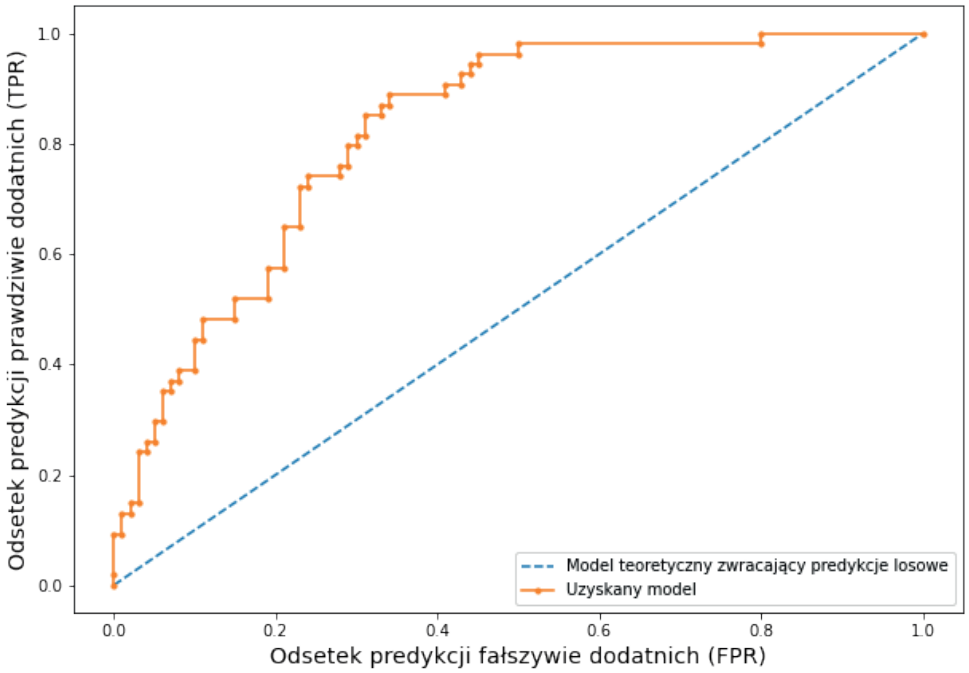
```
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (log_loss,
                             roc_auc_score,
                             confusion_matrix,
                             classification_report,
                             make_scorer)
import matplotlib.pyplot as plt
from sklearn.model_selection import (cross_validate, StratifiedKFold)
import optuna
df = pd.read_csv('diabetes.csv')
#etykiety - ostatnia kolumna
columns = df.columns
X = df[columns[:-1]]
y = df[columns[-1]]

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state = 42,
                                                    stratify = y,
                                                    test_size = 0.2)

#trening regresji logistycznej z domyślnymi wartościami hiperparametrów
lr = LogisticRegression()
lr.fit(X_train, y_train)
preds = lr.predict(X_test)

#macierz pomyłek
print(confusion_matrix(y_test, preds))
#metryki: czułość, precyzja, F1, dokładność klasyfikacji
print(classification_report(y_test, preds))

#metryka użyta do ewaluacji modelu w trakcie optymalizacji
scoring = {'roc_macro': make_scorer(roc_auc_score)}

#funkcja celu
def objective(trial, model, get_space, X, y):
  model_space = get_space(trial)

  mdl = model(**model_space)
   #5-krotny sprawdzian krzyżowy
  scores = cross_validate(mdl, X, y, scoring = scoring,
                                     cv = StratifiedKFold(n_splits = 5),
                                     return_train_score = True)

  return np.mean(scores['test_roc_macro'])
```

```
#przestrzeń wartości hiperparametrów poddawanych optymalizacji
def get_space(trial):
  space = {"C": trial.suggest_uniform("C", 0, 2),
     'max_iter': trial.suggest_int('max_iter', 100, 1000),
     "solver": trial.suggest_categorical("solver", ["lbfgs", "liblinear"])}
  return space

trials = 15 #liczba prób

model = LogisticRegression

#inicjalizacja optymalizacji
study = optuna.create_study(direction = 'maximize')
study.optimize(lambda x: objective(x, model, get_space,
                                   X_train,
                                   y_train),
               n_trials = trials)

#dobrane w wyniku optymalizacji wartości hiperparametrów
print('params: ', study.best_params)
lr = model(**study.best_params)
lr.fit(X_train, y_train)
preds = lr.predict(X_test)
print('test ROC_AUC: ', roc_auc_score(y_test, preds))
print(classification_report(y_test, preds))

def get_threshold(y_train, y_test, train_preds, test_preds):
  from sklearn.metrics import roc_curve
  import numpy as np

  y = np.append(y_train, y_test)
  preds = np.append(train_preds, test_preds)
  fpr, tpr, thresholds = roc_curve(y, preds)
  gmeans = np.sqrt(tpr * (1 - fpr))
  ix = np.argmax(gmeans)
  print('Best Threshold = %f' % thresholds[ix])
  return thresholds[ix]

def plot_roc_curve(y, preds, image_path = None):
  import matplotlib.pyplot as plt
  from sklearn.metrics import roc_curve

  fpr, tpr, thresholds = roc_curve(y, preds)
  plt.plot([0,1], [0,1], linestyle = '--', label = 'Model
  teoretyczny zwracający predykcje losowe')
  plt.plot(fpr, tpr, marker = '.', label = 'Uzyskany model')
  plt.xlabel('Odsetek predykcji fałszywie dodatnich (FPR)')
```

```
  plt.ylabel('Odstek predykcji prawdziwie dodatnich (TPR)')
  plt.legend()

#predykcja prawdopodobieństwa przynależności do klas
lr = model(**study.best_params)
lr.fit(X_train, y_train)
preds_test = lr.predict_proba(X_test)
preds_train = lr.predict_proba(X_train)
plot_roc_curve(y_test, preds_test[:,1])
#dobór punktu odcięcia na krzywej ROC
th = get_threshold(y_train, y_test, preds_train[:,1], preds_test[:,1])
bool_preds = preds_test[:,1] > th
print(confusion_matrix(y_test, bool_preds))
print(classification_report(y_test, bool_preds))
```

Przeprowadzenie optymalizacji hiperparametrów modelu najsilniej wpłynęło na jego czułość – model o domyślnych wartościach hiperparametrów uzyskał czułość wykrywania obiektów z klasy 1. (osób chorych na cukrzycę) wynoszącą 0,52, zaś model zoptymalizowany czułość 0,56. Należy jednak zauważyć, że jest to zmiana stosunkowo niewielka i odpowiada wzrostowi liczby poprawnie zaklasyfikowanych obiektów należących do klasy 1. z 28 do 30.

Wyniki uzyskane za pomocą metody predict_proba różnią się od tych uzyskanych metodą predict, które przytoczono powyżej. Zastosowanie metody predict_proba i dobranie punktu odcięcia na krzywej ROC (rys. R.8) za pomocą zaimplementowanego algorytmu bazującego na maksymalizacji średniej geometrycznej precyzji i czułości pozwoliło uzyskać czułość modelu wynoszącą 0,83, co odpowiada poprawnej klasyfikacji 45 spośród 54 obiektów należących do klasy 1. Zaobserwowano natomiast spadek specyficzności modelu, co świadczy o tym, że mniejsza liczba obiektów należących do klasy 0 została zaklasyfikowana poprawnie.



Rysunek R.8. Krzywa ROC uzyskana na podstawie predykcji dla zbioru testowego

Podjęcie decyzji, która metryka jest bardziej istotna – czułość czy specyficzność – wymaga rozważenia, jakiego typu dane są analizowane i jakie będą konsekwencje nieprawidłowego zaklasyfikowania obiektów należących do poszczególnych klas. W zadaniu analizowane są dane medyczne, a klasa oznaczona liczbą 1 odpowiada osobom chorym. W przypadku modeli stosowanych w diagnostyce medycznej zazwyczaj dąży się do uzyskania możliwie dużej liczby predykcji prawdziwie pozytywnych, nawet wtedy, gdy wiąże się to ze wzrostem liczby predykcji fałszywie pozytywnych – fałszywy alarm może doprowadzić do niepotrzebnego przeprowadzenia dalszej diagnostyki, jest to jednak mniej niekorzystne dla zdrowia pacjenta niż uznanie osoby chorej za zdrową, co może doprowadzić do niepodjęcia leczenia.

Zadanie 15.1

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import (StandardScaler, OrdinalEncoder)
from sklearn.model_selection import train_test_split
from sklearn.metrics import (confusion_matrix,
                             classification_report,
                             make_scorer,
                             accuracy_score)
from sklearn.svm import SVC
from sklearn.model_selection import (cross_validate, StratifiedKFold)
from collections import Counter
import optuna

#wczytanie danych
df = pd.read_csv('6 class csv.csv')
df.dropna(inplace = True)

#utworzenie wektora etykiet i macierzy parametrów
columns = df.columns
data = df[columns[columns != 'Star type']].to_numpy()
labels = df['Star type'].to_numpy()

#zakodowanie zmiennych kategorialnych za pomocą zmiennych liczbowych
enc = OrdinalEncoder()
data = enc.fit_transform(data)

#wyświetlenie liczebności klas
print(Counter(labels))

#podział danych na zbiór uczący i testowy
X_train, X_test, y_train, y_test = train_test_split(data,
                                                    labels,
                                                    stratify = labels,
                                                    random_state = 42)
```

```
#standaryzacja danych
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

#trening modelu o domyślnych wartościach hiperparametrów
svm = SVC()
svm.fit(X_train_scaled, y_train)
preds = svm.predict(X_test_scaled)
print(classification_report(y_test, preds))
print(confusion_matrix(y_test, preds))

#metryka wykorzystywana do ewaluacji modelu podczas optymalizacji
scoring = {'accuracy': make_scorer(accuracy_score)}

#funkcja celu
def objective(trial, model, get_space, X, y):
  model_space = get_space(trial)

  mdl = model(**model_space)
  scores = cross_validate(mdl, X, y, scoring = scoring,
                                     cv = StratifiedKFold(n_splits = 5),
                                     return_train_score = True)

  return np.mean(scores['test_accuracy'])

#przestrzeń hiperparametrów
def get_space(trial):
  space = {"C": trial.suggest_uniform("C", 0, 2),
     "max_iter": trial.suggest_int("max_iter", 100, 1000),
     "kernel": trial.suggest_categorical("kernel",
                                  ["linear", "poly", "rbf", "sigmoid"]),
      "degree": trial.suggest_int("degree", 2,5)}
  return space

trials = 50 #liczba prób

model = SVC

#optymalizacja modelu
study = optuna.create_study(direction = 'maximize')
study.optimize(lambda x: objective(x, model, get_space,
                                           X_train_scaled,
                                           y_train),
               n_trials = trials)
```

```
#dobrane wartości hiperparametrów
print('params: ', study.best_params)

#trening modelu o optymalnych wartościach hiperparametrów
svm = model(**study.best_params)
svm.fit(X_train_scaled, y_train)

#ewaluacja modelu
preds = svm.predict(X_test_scaled)
print(classification_report(y_test, preds))
print(confusion_matrix(y_test, preds))
```

Zbiór danych jest idealnie zrównoważony (liczebność wszystkich klas jest taka sama), więc ewaluację modelu podczas optymalizacji można było przeprowadzić na podstawie dokładności klasyfikacji. Inne metryki również mogłyby zostać użyte, jednak wymagałoby to uśredniania ich wartości dla wszystkich klas, ponieważ są zdefiniowane dla przypadków binarnych. Dokładność klasyfikacji jest wyznaczana tak samo bez względu na liczbę klas obecnych w analizowanym zbiorze danych.

Optymalizacja hiperparametrów klasyfikatora pozwoliła uzyskać znacznie lepsze rezultaty niż w przypadku modelu o domyślnych wartościach hiperparametrów – po optymalizacji uzyskano dokładność klasyfikacji wynoszącą 100%. Największą poprawę można zaobserwować dla klasy 4., w przypadku której precyzja klasyfikacji wzrosła z 0,69 do 1,00 oraz dla klasy 3., w przypadku której precyzja i czułość wzrosły odpowiednio z 0,89 i 0,80 do wartości 1,00.

Zadanie 16.1

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (accuracy_score, f1_score,
                             make_scorer, confusion_matrix)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import optuna
from sklearn.model_selection import cross_validate, GroupKFold
import pickle
import scipy.stats

X_mfcc = np.load('mfcc_feats.npy')
X_gfcc = np.load('gfcc_feats.npy')
y_org = np.load('labels.npy')
folds = np.load('folds.npy')

X = np.hstack((X_mfcc, X_gfcc))

X = np.hstack((np.mean(X, axis = 2),
       np.std(X, axis = 2),
       np.median(X, axis = 2),
```

```
        np.percentile(X, 25, axis = 2),
        np.percentile(X, 75, axis = 2),
        scipy.stats.iqr(X, rng = (10, 90), axis = 2),
        scipy.stats.kurtosis(X, axis = 2),
        scipy.stats.skew(X, axis = 2),
        np.min(X, axis = 2),
        np.max(X, axis = 2)
        ))

y_not_6 = y_org != 6
X = X[y_not_6]
folds = folds[y_not_6]
y = y_org[y_not_6]

train_folds_mask = folds != 10
test_fold_mask = folds == 10

train_folds = folds[train_folds_mask]
test_fold = folds[test_fold_mask]

y_train = y[train_folds_mask]
y_test = y[test_fold_mask]

X_train = X[train_folds_mask]
X_test = X[test_fold_mask]

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

scoring = {'f1_weighted': make_scorer(f1_score, average = 'weighted')}

model = RandomForestClassifier
def get_space(trial):
  space = {"n_estimators": trial.suggest_int("n_estimators", 10, 200),
           "max_depth": trial.suggest_int("max_depth", 1, 100),
           "min_samples_split": trial.suggest_int(
                                          "min_samples_split", 2, 20),
           "n_jobs": trial.suggest_int("n_jobs", -1, -1)}

  return space

trials = 60 #liczba prób

def objective(trial, model, X, y):
  model_space = get_space(trial)
```

```
  mdl = model(**model_space)
  scores = cross_validate(mdl, X, y, groups = train_folds,
                                     scoring = scoring,
                                     cv = GroupKFold(n_splits = len(
                                                np.unique(train_folds))),
                                    return_train_score = True)

  return np.mean(scores['test_f1_weighted'])

study = optuna.create_study(direction = 'maximize')
study.optimize(lambda x: objective(x, model, X_train,y_train),
               n_trials = trials)

print('params: ', study.best_params)
rf = model(**study.best_params)
rf.fit(X_train, y_train)
preds = rf.predict(X_test)
pickle.dump(rf, open('rf_model_f1', 'wb+'))
print('test accuracy = ', accuracy_score(y_test, preds))
print('test F1 = ', f1_score(y_test, preds, average = 'weighted'))
print(confusion_matrix(y_test, preds))
```

Zoptymalizowany model dał nieco lepsze rezultaty: z użytymi domyślnymi wartościami hiperparametrów dokładność wynosi około 0,66, natomiast z hiperparametrami dobranymi w wyniku optymalizacji wynosi około 0,68.

Zadanie 16.2

```
import numpy as np
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.metrics import (accuracy_score, f1_score,
                             make_scorer, confusion_matrix)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import optuna
from sklearn.model_selection import cross_validate, GroupKFold
import pickle
import scipy.stats

X_mfcc = np.load('mfcc_feats.npy')
X_gfcc = np.load('gfcc_feats.npy')
y_org = np.load('labels.npy')
folds = np.load('folds.npy')

X = np.hstack((X_mfcc, X_gfcc))
```

```
X = np.hstack((np.mean(X, axis = 2),
       np.std(X, axis = 2),
       np.median(X, axis = 2),
       np.percentile(X, 25, axis = 2),
       np.percentile(X, 75, axis = 2),
       scipy.stats.iqr(X, rng = (10, 90), axis = 2),
       scipy.stats.kurtosis(X, axis = 2),
       scipy.stats.skew(X, axis = 2),
       np.min(X, axis = 2),
       np.max(X, axis = 2)
       ))

y_not_6 = y_org != 6
X = X[y_not_6]
folds = folds[y_not_6]
y = y_org[y_not_6]

train_folds_mask = folds != 10
test_fold_mask = folds == 10

train_folds = folds[train_folds_mask]
test_fold = folds[test_fold_mask]

y_train = y[train_folds_mask]
y_test = y[test_fold_mask]

X_train = X[train_folds_mask]
X_test = X[test_fold_mask]

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

scoring = {'f1_weighted': make_scorer(f1_score, average = 'weighted')}

model = ExtraTreesClassifier
def get_space(trial):
  space = {"n_estimators": trial.suggest_int("n_estimators", 1, 200),
           "max_depth": trial.suggest_int("max_depth", 1, 100)}

  return space

trials = 60 #liczba prób
def objective(trial, model, X, y):
  model_space = get_space(trial)

  mdl = model(**model_space)
```

```
  scores = cross_validate(mdl, X, y, groups=train_folds,
                                     scoring = scoring,
                                     cv = GroupKFold(n_splits = len(
                                               np.unique(train_folds))),
                                    return_train_score = True)

  return np.mean(scores['test_f1_weighted'])

study = optuna.create_study(direction = 'maximize')
study.optimize(lambda x: objective(x, model, X_train, y_train),
               n_trials = trials)

print('params: ', study.best_params)
xt = model(**study.best_params)
xt.fit(X_train, y_train)
preds = xt.predict(X_test)
pickle.dump(xt, open('extratrees_model_f1', 'wb+'))
print('test accuracy = ', accuracy_score(y_test, preds))
print('test F1 = ', f1_score(y_test, preds, average = 'weighted'))
print(confusion_matrix(y_test, preds))
```

Klasyfikator ExtraTrees uzyskał dokładność klasyfikacji zbliżoną do dokładności klasyfikatora lasu losowego – wynosi ona około 0,66. Czas trwania optymalizacji był zgodnie z oczekiwaniami krótszy – model lasu losowego był optymalizowany przez 9 min i 24 s, natomiast model ExtraTrees przez 7 min i 21 s.

UWAGA

Czas trwania optymalizacji zależy od dostępnych zasobów obliczeniowych i może być inny w przypadku uruchomienia kodu na innym komputerze.

Zadanie 16.3

```
import numpy as np
import lightgbm as lgb
from sklearn.metrics import (accuracy_score, f1_score,
                             make_scorer, confusion_matrix)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import optuna
from sklearn.model_selection import cross_validate, GroupKFold
import pickle
import scipy.stats

X_mfcc = np.load('mfcc_feats.npy')
X_gfcc = np.load('gfcc_feats.npy')
y_org = np.load('labels.npy')
folds = np.load('folds.npy')
```

```
X = np.hstack((X_mfcc, X_gfcc))

X = np.hstack((np.mean(X, axis = 2),
       np.std(X, axis = 2),
       np.median(X, axis = 2),
       np.percentile(X, 25, axis = 2),
       np.percentile(X, 75, axis = 2),
       scipy.stats.iqr(X, rng = (10, 90), axis = 2),
       scipy.stats.kurtosis(X, axis = 2),
       scipy.stats.skew(X, axis = 2),
       np.min(X, axis = 2),
       np.max(X, axis = 2)
       ))

y_not_6 = y_org != 6
X = X[y_not_6]
folds = folds[y_not_6]
y = y_org[y_not_6]

train_folds_mask = folds != 10
test_fold_mask = folds == 10

train_folds = folds[train_folds_mask]
test_fold = folds[test_fold_mask]

y_train = y[train_folds_mask]
y_test = y[test_fold_mask]

X_train = X[train_folds_mask]
X_test = X[test_fold_mask]

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

scoring = {'f1_weighted': make_scorer(f1_score, average = 'weighted')}

model = lgb.LGBMClassifier
def get_space(trial):
  space = {"boosting_type": trial.suggest_categorical(
                            'boosting_type', ["gbdt"]),
           "num_leaves": trial.suggest_int("num_leaves", 2, 200),
           "n_estimators": trial.suggest_int("n_estimators", 2, 200),
           "max_depth": trial.suggest_int("max_depth", 2, 200),
           "learning_rate": trial.suggest_discrete_uniform(
                              "learning_rate", 0.00001, 1, 0.0001),
```

```
            "subsample": trial.suggest_discrete_uniform(
                                "subsample", 0.01, 1, 0.01),
            "colsample_bytree": trial.suggest_discrete_uniform(
                                "colsample_bytree", 0.01, 1, 0.01),
            "min_split_gain": trial.suggest_discrete_uniform(
                                "min_split_gain", 0.01, 1, 0.01),
            "min_child_samples": trial.suggest_int(
                                "min_child_samples", 1, 50),
            "n_jobs": trial.suggest_int("n_jobs", -1, -1)}

  return space

trials = 160 #liczba prób

def objective(trial, model, X, y):
  model_space = get_space(trial)

  mdl = model(**model_space)
  scores = cross_validate(mdl, X, y, groups = train_folds,
                                     scoring = scoring,
                                     cv = GroupKFold(n_splits = len(
                                              np.unique(train_folds))),
                                    return_train_score = True)

  return np.mean(scores['test_f1_weighted'])

study = optuna.create_study(direction = 'maximize')
study.optimize(lambda x: objective(x, model, X_train, y_train),
               n_trials = trials)

print('params: ', study.best_params)
lgbm = model(**study.best_params)
lgbm.fit(X_train, y_train)
preds = lgbm.predict(X_test)
pickle.dump(lgbm, open('lgb_model_f1', 'wb+'))
print('test accuracy = ', accuracy_score(y_test, preds))
print('test F1 = ', f1_score(y_test, preds, average = 'weighted'))
print(confusion_matrix(y_test, preds))
```

Najlepsze wyniki udało się uzyskać przy zastosowaniu klasyfikatora zaimplementowanego przy użyciu pakietu LightGBM: uzyskana dokładność klasyfikacji wynosi około 0,72.

Najczęściej źle klasyfikowaną klasą jest klasa 1. Wynika to prawdopodobnie z jej niewielkiej liczebności – w klasie 1. są tylko 203 obiekty, podczas gdy prawie wszystkie pozostałe klasy zawierają cztery razy więcej obiektów (jedynie klasa 3. liczy 675 obiektów).

Zadanie 17.1

Poniżej zamieszczono kod zawierający definicję przykładowej sieci neuronowej pozwalającej uzyskać dokładność klasyfikacji wynoszącą odpowiednio 0,63 i 0,61 dla zbiorów walidacyjnego i testowego. Względem architektury przedstawionej w Przykładzie 17.1 wprowadzono następujące zmiany:

- dodano dwie warstwy ukryte,
- zmieniono algorytm optymalizacji wag z SGD na Adam – opis algorytmu można znaleźć w dokumentacji pakietu PyTorch: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html,
- dodano warstwy dropout,
- zmieniono wartości współczynnika uczenia,
- zwiększono liczbę epok.

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
from ignite.engine import (create_supervised_trainer,
                           create_supervised_evaluator, Events)
from ignite.metrics import Loss, Accuracy
from ignite.contrib.handlers import ProgressBar
from ignite.handlers import FastaiLRFinder

class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(2574, 256)
    self.fc2 = nn.Linear(256, 120)
    self.fc3 = nn.Linear(120, 20)
    self.fc4 = nn.Linear(20, 60)
    self.fc5 = nn.Linear(60, 35)
    self.dropout1 = nn.Dropout(p = 0.2)

  def forward(self, x):
    x = F.relu(self.fc1(x))
    x = self.dropout1(x)
    x = F.relu(self.fc2(x))
    x = self.dropout1(x)
    x = F.relu(self.fc3(x))
    x = F.relu(self.fc4(x))
    x = self.fc5(x)
    return F.log_softmax(x, dim = 1)
```

```
feats = np.load('logfbank_feats.npy')
labels = np.load('labels.npy')
feats = feats.reshape(feats.shape[0],-1)
feats = feats.astype(np.float32)

X_train, X_val_test, y_train, y_val_test = train_test_split(
                                                    feats,
                                                    labels,
                                                    random_state = 42,
                                                    stratify = labels,
                                                    train_size = 0.8)
X_val, X_test, y_val, y_test = train_test_split(X_val_test,
                                                    y_val_test,
                                                    random_state = 42,
                                                    stratify = y_val_test,
                                                    train_size = 0.5)

trainset = TensorDataset(torch.tensor(X_train),torch.tensor(y_train))
valset = TensorDataset(torch.tensor(X_val), torch.tensor(y_val))
testset = TensorDataset(torch.tensor(X_test), torch.tensor(y_test))

trainloader = DataLoader(trainset, batch_size = 256)
valloader = DataLoader(valset, batch_size = 256)
testloader = DataLoader(testset, batch_size = 256)

device = "cuda" if torch.cuda.is_available() else "cpu"
criterion = nn.NLLLoss()

model = Net()
model.to(device)

optimizer = optim.Adam(model.parameters(), lr = 3e-5)
init_model_state = model.state_dict()
init_opt_state = optimizer.state_dict()

trainer = create_supervised_trainer(model, optimizer, criterion,
                                            device = device)
evaluator = create_supervised_evaluator(model,
                                        metrics = {
                                            "acc": Accuracy(),
                                            "loss": Loss(nn.NLLLoss())},
                                        device = device)

ProgressBar(persist = True).attach(trainer,
                output_transform = lambda x: {"batch loss": x})
```

```
@trainer.on(Events.EPOCH_COMPLETED)
def log_validation_results(trainer):
  evaluator.run(valloader)
  metrics = evaluator.state.metrics
  print("Validation Results - Epoch: {} Avg accuracy: {
                                          :.2f} Avg loss: {:.2f}".format(
                                            trainer.state.epoch, metrics[
                                                'acc'], metrics['loss']))

trainer.run(trainloader, max_epochs = 400)
evaluator.run(testloader)
print(evaluator.state.metrics)
```

Bibliografia

Abadi M., Barham P., Chen J. i in. (2016). *TensorFlow: Large-scale Machine Learning on Heterogeneous Systems.* Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16), November 2–4, Savannah, GA, USA. Berkeley: USENIX Association, s. 265–283.

Abhang A.P., Gawali B.W., Mehrotra S.C. (2016). *Technical Aspects of Brain Rhythms and Speech Parameters* [w:] Abhang P.A., Gawali B., Mehrotra S., *Introduction to EEG- and Speech-Based Emotion Recognition.* Cambridge: Academic Press, s. 51–79.

Akiba T., Sano S., Yanase T., Ohta T., Koyama M. (2019). *Optuna: A Next-generation Hyperparameter Optimization Framework.* Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 4–8, Anchorage, AK, USA, s. 2623–2631.

Bell A., Sejnowski T.J. (1995). *An information maximization approach to blind separation and blind deconvolution.* Neural Computation, vol. 7, no. 6, s. 1129–1159.

Boashash B. (2015). *Time-frequency Signal Analysis and Processing: A Comprehensive Reference.* Elsevier LTD.

Boser B., Guyon I., Vapnik V. (1992). *A Training Algorithm for Optimal Margin Classifiers*. COLT 92: Proceedings of the 5th Annual Workshop on Computational Learning Theory, July 27–29, Pittsburgh, PA, USA. New York: Association for Computing Machinery, s. 144–152.

Breiman L. (2001). *Random forests.* Machine Learning, vol. 45, s. 5–32.

Comon P. (1994). *Independent component analysis, A new concept?* Signal Processing, vol. 36, issue 3, s. 287–314.

Cortes C., Vapnik V. (1997). *Soft Margin Classifier.* United States Patent, Patent Number 5,640,492.

Cover T., Hart P. (1967). *Nearest neighbor pattern classification.* IEEE Transactions on Information Theory, vol. 13, issue 1, s. 21–27.

Duda R., Hart P., Stork D. (2000). *Pattern Classification.* New York: Wiley.

Eyben F., Wöllmer M., Schuller B. (2010). *openSMILE – The Munich Versatile and Fast Open-Source Audio Feature Extractor.* Proceedings of the 9thACM International Conference on Multimedia, October 25–29, Firenze, Italy, s. 1459–1462.

Falcon W. (2019). *PyTorch Lightning*, https://github.com/PyTorchLightning/pytorch-lightning, dostęp: 10.03.2022.

Fomin V., Anmol J., Desroziers S., Kriss J., Tejani A. (2020). *High-level library to help with training neural networks in PyTorch*, https://github.com/pytorch/ignite, dostęp: 10.03.2022.

Goodfellow I., Bengio Y., Courville A. (2018). *Deep learning. Systemy uczące się.* Warszawa: Wydawnictwo Naukowe PWN.

Hall P., Byeong U.P., Samworth R.J. (2007*). Choice of neighbor order in nearest-neighbor classification.* The Annals of Statistics, vol. 36, no. 5, s. 2135–2152.

Hand D., Mannila H., Smyth P. (2005). *Eksploracja danych.* Warszawa: Wydawnictwa Naukowo-Techniczne.

Hartigan J.A., Wong M.A. (1979). *Algorithm AS 136: A K-Means Clustering Algorithm.* Journal of the Royal Statistical Society. Series C (applied Statistics), vol. 28, no. 1, s. 100–108.

Hosom J.-P. (2003). *Speech Recognition* [w:] Bidgoli H. (Ed.), *Encyclopedia of Information Systems*, Academic Press, s. 55–169.

Hyvärinen A., Karhunen J., Oja E. (2001). *Independent Component Analysis.* New York – Chichester – Weinheim – Brisbane – Singapore – Toronto: John Wiley & Sons.

Jaskowiak P.A., Campello R.J.G.B. (2011). *Comparing Correlation Coefficients as Dissimilarity Measures for Cancer Classification in Gene Expression Data*. VI Brazilian Symposium on Bioinformatics, 12–12 August 2011, Brasilia, Brazil.

Johnson R.A., Wichern D.W. (2007). *Applied Multivariate Statistical Analysis.* New York: Pearson.
Jurlewicz T., Skoczylas Z. (2001). *Algebra liniowa 1. Definicje, twierdzenia, wzory.* Wydanie ósme. Wrocław: Oficyna Wydawnicza GiS.
Kaiser H.F. (1960). *The Application of Electronic Computers to Factor Analysis.* Educational and Psychological Measurement, vol. 20, no. 1, s. 141–151.
Ketkar N. (2017). *Stochastic Gradient Descent* [w:] Ketkar N., *Deep Learning with Python. A Hands-on Introduction*. Berkeley: Apress, s. 113–132.
Koronacki J., Ćwik J. (2015). *Statystyczne systemy uczące się.* Warszawa: Akademicka Oficyna Wydawnicza EXIT.
Krzyśko M. (2000). *Podstawy wielowymiarowego wnioskowania statystycznego.* Poznań: Wydawnictwo Naukowe UAM.
Liu G.K. (2018). *Evaluating Gammatone Frequency Cepstral Coefficients with Neural Networks for Emotion Recognition from Speech.* arXiv.
Liu X. (2012). *Classification accuracy and cut point selection.* Statistics in Medicine, vol. 31, issue 23, s. 2676–2686.
Machine Learner, C. B.-S.-s. (2022). https://commons.wikimedia.org/wiki/File:Nonlinear_SVM_example_illustration.svg, dostęp: 20.04.2022.
MacQueen J. (1967). *Some methods for classification and analysis of multivariate observations* [w:] Le Cam L.M., Neyman J. (Eds.), *Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley: University of California Press, s. 281–297.
Mardia K., Kent J., Bibby J. (1979). *Multivariate Analysis.* London: Academic Press.
MartinThoma, c. C.-S.-s., https://commons.wikimedia.org/wiki/File:Roc_curve.svg, dostęp: 20.04.2022.
McCullagh P., Nelder J.A. (1989). *Generalized Linear Models.* 2nd Edition. New York, Chapman & Hall.
McCulloch C., Searle S.R. (2000). *Generalized, Linear, and Mixed Models.* New York – Chichester – Weinheim – Brisbane – Singapore – Toronto: John Wiley & Sons.
McCulloch W.S., Pitts W. (1943). *A logical calculus of the ideas immanent in nervous activity.* The Bulletin of Mathematical Biophysics, vol. 3, s. 115–133.
Mercer J. (1909). *Functions of positive and negative type and their connection with the theory of integral equations.* Philosophical Transactions of the Royal Society of London. Series A, vol. 209, s. 415–446.
Morrison D.F. (1990). *Multivariate statistical methods.* New York: McGraw-Hill.
Nelder J.A., Wedderburn R.W. (1972). *Generalized Linear Models.* Journal of the Royal Statistical Society. Series A (General), vol. 135, no. 3, s. 370–384.
Olson R.S., Bartley N., Urbanowicz R.J., Moore J.H. (2016). *Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science.* Proceedings of the Genetic and Evolutionary Computation Conference, Denver, CO, USA, s. 485–492.
Osowski S. (1996). *Sieci neuronowe w ujęciu algorytmicznym.* Warszawa: Wydawnictwa Naukowo-Techniczne.
Paszke A., Gross S., Massa F., Lerer A., Bradbury J., Chanan G., Killeen T., Lin Z., Gimelshein N., Antiga L., Desmaison A., Köpf A., Yang E., DeVito Z., Raison M., Tejani A., Chilamkurthy S., Steiner B., Fang L., Bai J., Chintala S. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library* [w:] Wallach H., Larochelle H., Beyegelzimer A., d'Alché-Buc F., Fox E., Garnett R. (Eds.), *Advances in Neural Information Processing Systems 32*, Paper ID 4399, s. 8026–8037.
Porath E.B., Gilboa I. (1994). *Linear measures, the Gini index, and the income-equality trade-off.* Journal of Economic Theory, vol. 64, issue 2, s. 443–467.
Ravindran S., Demirogulu C., Anderson D.V. (2003). *Speech recognition using filter-bank features.* 37th Asilomar Conference on Signals, Systems & Computers, Pacific Grove, CA, USA, vol. 2, s. 1900–1903.
Shao Y., Jin Z., Wang D.L., Srinivasan S. (2009). *An auditory-based feature for robust speech recognition.* IEEE International Conference on Acoustics, Speech and Signal Processing, April 19–24, Taipei, Taiwan, s. 4625–4628.
Skulimowski A.M. (2002). *Metoda obwiedni ekstremalnych w prognozowaniu kursów walutowych* [w:] Trzaskalik T. (red. nauk.), *Modelowanie preferencji a ryzyko '02*. Katowice: Wydawnictwo Akademii Ekonomicznej im. Karola Adamieckiego, s. 315–329.

Skulimowski A.M.J. (2014). *Reveiling complexity-related time-series features with the monotonic aggregation transform*. Proceedings: 2014 IEEE 26th International Conference on Tools with Artificial Intelligence. ICTAI 2014, November 10–12, Limassol, Cyprus, s. 694–700.

Sneath P., Sokal R. (1973). *Numerical Taxonomy: The Principles and Practice of Numerical Classification.* San Francisco: W.H. Freeman and Company.

Stanisz A. (2007). *Przystępny kurs statystyki: z zastosowaniem STATISTICA PL na przykładach z medycyny, T. 3: Analizy wielowymiarowe.* Kraków: StatSoft.

Szeliga M. (2017). *Data science i uczenie maszynowe.* Warszawa: Wydawnictwo Naukowe PWN.

Tadeusiewicz R., Szaleniec M. (2015). *Leksykon sieci neuronowych.* Wrocław: Wydawnictwo Fundacji Projekt Nauka.

Thon D. (1982). *An axiomatization of the Gini coefficient.* Mathematical Social Sciences, vol. 2, issue 1, s. 131–143.

Tkach D., Huang H., Kuiken T.A. (2010). *Study of stability of time-domain features for electromyographic pattern recognition.* Journal of NeuroEngineering and Rehabilitation, vol. 7, no. 1, Article No. 21.

Unal I. (2017). *Defining an Optimal Cut-point Value in ROC Analysis: An Alternative Approach.* Computational and Mathematical Methods in Medicine, Article ID 3762651.

Vapnik V.N. (1998). *Statistical Learning Theory.* New York – Chichester – Weinheim – Brisbane – Singapore – Toronto: John Wiley & Sons.

Wang X., Liu A., Zhang Y., Xue F. (2019). *Underwater Acoustic Target Recognition: A Combination of Multi-Dimensional Fusion Features and Modified Deep Neural Network.* Remote Sensing, vol. 11, no. 16, Article ID 1888.

Ward J. (1963). *Hierarchical grouping to optimize an objective function.* Journal of the American Statistical Association, vol. 58, issue 301, s. 236–244.

Weymark J.A. (1981). *Generalized Gini inequality indices.* Mathematical Social Sciences, vol. 1, s. 409–430.

Yang Y. (2017). *Signal Theoretic Approach for Envelope Analysis of Real-Valued Signals.* IEEE Access, vol. 5, s. 5623–5630.

Yu X., Zhang J., Liu J., Wan W., Yang W. (2010). *An audio retrieval method based on chromagram and distance metrics.* International Conference on Audio, Language and Image Processing, November 23–25, Shanghai, China.

e-ISBN 978-83-67427-05-0