Uniwersytet Zielonogórski

Wydział Informatyki, Elektrotechniki i Automatyki

Praca dyplomowa

Kierunek: Informatyka

Analiza porównawcza bibliotek uczenia maszynowego języka C++ na potrzeby zastosowań w biostatystyce

inż. Kacper Wojciechowski

Promotor:
Prof. dr hab. inż. Dariusz Uciński

Pracę akceptuję:

.....
(data i podpis promotora)

Zielona Góra, czerwiec 2023

Streszczenie

Niniejsza praca ma na celu analizę i porównanie dostępnych w języku C++ bibliotek uczenia maszynowego, pod kątem ich zastosowania w pracy na danych biostatystycznych. W kolejnych rozdziałach czytelnik zapoznawany jest z:

- Ogólną postacią problemów napotykanych w procesie implementacji rozwiązań uczenia maszynowego;
- Charakterystyką wybranego zestawu danych biostatystycznych wykorzystanych do testów omawianych bibliotek;
- Typami oraz uzyskiwanymi wynikami wybranych pod kątem danych eksperymentalnych metod uczenia maszynowego w środowisku prototypowym;
- Bibliotekami Shogun, Shark-ML i Dlib wraz z metodami implementacji poszczególnych metod wzorcowych;
- Zbiorczym podsumowaniem funkcjonalności oferowanych przez wyżej wymienione biblioteki.

Słowa kluczowe: uczenie maszynowe, C++, biblioteka, sieci neuronowe, głębokie uczenie maszynowe, płytkie uczenie maszynowe.

Spis treści

Spis rysunków

Spis tabel

Rozdział 1

Wstęp

1.1. Wprowadzenie

We współczesnym stanie techniki coraz częściej można spotkać się z urządzeniami i programami o inteligentnych funkcjach, takich jak predykcja zjawisk na podstawie zestawu danych, rozpoznawanie obrazu, analiza mowy, czy przetwarzanie języka naturalnego. Znajdują one zastosowanie w różnych dziedzinach codziennego życia, m.in. w medycynie. W zależności od potrzeb, techniki uczenia maszynowego można wykorzystać do zastosowań medycznych, jak np. rozpoznawanie komórek rakowych na skanach rezonansem magnetycznym, podejmowanie decyzji na podstawie zbioru objawów obecnych u pacjenta, lub przewidywanie norm związków naturalnie występujących w organiźmie ludzkim w zależności od okoliczności i wyników pomiarów.

Jedną z istotnych dziedzin medycyny jest biostatystyka, polegająca na wykorzystaniu analizy statystycznej do wnioskowania na podstawie zbiorów danych, takich jak rezultaty przeprowadzonych badań (np. morfologicznych, poziomu poszczególnych hormonów we krwii, itp.), informacji o nawykach żywieniowych oraz stylu życia pacjenta. Szczególnie istotną formą systemów operujących w tej dziedzinie są systemy eksperckie, wykorzystujące techniki płytkiego i głębokiego uczenia maszynowego w celu wspierania diagnozy stawianej przez wykwalifikowanych lekarzy.

U podstaw wyżej wymienionych zagadnień leży implementacja rozwiązań opartych o teorię uczenia maszynowego, oraz wszelkie związane z tym problemy. W związku z tym na przestrzeni lat powstało wiele gotowych narzędzi, takich jak biblioteki i *frameworki*, mające na celu wsparcie programistów w szybkim i prawidłowym wprowadzaniu rozwiązań sztucznej inteligencji na różne platformy docelowe oraz w różnych językach, począwszy od języka C++, przez Python, po środowiska takie jak Matlab.

Istotnym krokiem w przygotowywaniu oprogramowania wykorzystującego sztuczną inteligencję jest prawidłowy wybór wspomnianych wcześniej narzędzi dokonywany na etapie projektowania, tak, aby oferowały one możliwości adekwatne do wymagań funkcjonalnych. Niniejsza praca dokonuje analizy porównawczej bibliotek uczenia maszynowego dla języka C++ w kontekście zastosowań w dziedzinie biostatystyki, celem umożliwienia czytelnikowi trafnego wyboru odpowiedniego narzędzia do realizacji projektu badawczego. Warto zaznaczyć, że niniejsza praca przedstawia jedynie wybrany zakres głównych funkcjonalności omawianych bibliotek ze względu na zastosowania w biostatystyce, w związku z czym mogą one posiadać większą ilość

bardziej szczegółowych funkcjonalności, lub nowe metody dodane po utworzeniu niniejszej pracy.

1.2. Cel i zakres pracy

Celem pracy jest przeprowadzenie analizy i przygotwanie zestawienia bibliotek do uczenia maszynowego dla języka C++, obrazując przykłady bazujące na zestawie danych biostatystycznych.

Zakres pracy obejmował:

- Przegląd dostępnych bibliotek języka C++;
- Inżynierię i kształtowanie danych;
- Płytkie i głębokie uczenie nadzorowane;
- Kwestie wydajnościowe w dopasowywaniu i wdrażaniu modeli;
- Badania praktyczne w oparciu o zestaw danych medycznych i biologicznych.

1.3. Struktura pracy

Pierwszy rozdział przedstawia ogólnym zagadnieniem dotykanym przez pracę, począwszy od dziedziny problemu i jej zastosowań, do istoty tematu pracy. Dodatkowo omawiany jest cel i zakres realizacji pracy, oraz jej strukturą.

Kolejny rozdział wprowadza czytelnika do tematu uczenia maszynowego, oraz napotykanych w nim problemów dotyczących złożoności obliczeniowej oraz zużycia zasobów. Stanowią one podstawę do zaproponowania języka C++ jako technologi wspierającej ich rozwiązanie przy pomocy bibliotek.

Tematem rozdziału trzeciego jest przygotowanie elementów testowych do wykorzystania w późniejszej analizie porównawczej. Składa się na nie wybranie i przygotowanie do zestawu danych biostatystycznych do procesu uczenia oraz wybrane wzorcowych rozwiązań. Czytelnik przeprowadzony jest przez normalizację danych i selekcję najlepiej dopasowanych regresorów, oraz zostaje zapoznany z przykładowymi wynikami rozwiązań wzorcowych.

Kolejne trzy rozdziały skupiają się na analizie głównych funkcjonalności wybranych bibliotek pod kątem zastosowań w biostatystyce. Czwarty rozdział przedstawia bibliotekę Shogun, piąty zapoznaje użytkownika z biblioteką Shark-ML, natomiast szósty omawia bibliotekę Dlib. Wprowadzają one czytelnika kolejno w poszczególne aspekty pracy z wybranym produktem, od akceptowanych formatów danych, przez manipulację obserwacjami, po dostępne modele i metody ich analizy.

Rozdział siódmy zestawia podobieństwa i różnice między bibliotekami na podstawie wyników przeprowadzonych procesów uczenia, zestawiając wyniki uzyskanych modeli oraz dostępne funkcjonalności w formie tabel. Dodatkowo, zawarta tu została także subiektywna opinia autora w postaci opisów słownych na podstawie jego doświadczeń z implementacją rozwiązań i pracą ze źródłami wiedzy.

Rozdział 2

Uczenie maszynowe w ujęciu praktycznym

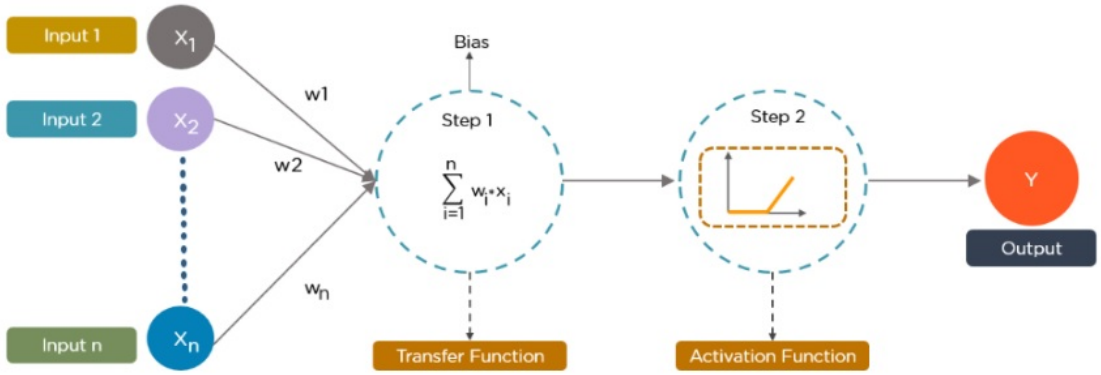
2.1. Problemy współczesnego uczenia maszynowego

Na uczenie maszynowe składają się zaawansowane techniki algorytmiczne i złożone struktury danych przeprowadzające obliczenia na zadanym przez użytkownika zestawie danych uczących, testujących, i danych otrzymywanych w trakcie użytkowania wytworzonego modelu.

Do podstawowych form modeli należą modele produkowane w wyniku technik takich jak regresja liniowa i nieliniowa, regresja logistyczna czy liniowa analiza dyskryminacyjna. W ich wyniku tworzone są modele w postaci wielomianów, które później wymagają stosunkowo bardzo małych nakładów mocy obliczeniowej w celu ewaluacji wyników na podstawie zadanego zestawu danych.

Bardziej zaawansowanymi metodami uczenia maszynowego są drzewa decyzyjne, stanowiące strukturę opartą o logikę drzewa. Każdy z poziomów drzewa odpowiada najlepszemu na danym etapie predyktorowi z dostępnych regresorów, powodując rozgałęzienie na poszczególne wartości lub zakresy. Proces obliczania wartości zmiennej wyjściowej odbywa się poprzez przejście przez drzewo od korzenia do jednego z końcowych liści.

Do najbardziej zaawansowanych, aczkolwiek także najbardziej wymagających obliczeniowo i pamięciowo technik uczenia maszynowego należą techniki uczenia głębokiego wykorzystujące sieci neuronowe, jak np. głębokie sieci neuronowe (ang. *Deep Neural Network, DNN*) i konwolucyjne sieci neuronowe (ang. *Convolutional Neural Network, CNN*). U podstaw tych metod leży struktura sieci neuronowej, składająca się z warstwy wejściowej, jednej lub więcej warstw ukrytych posiadających perceptrony, oraz jednej warstwy wyjściowej. Każdy węzeł z poprzedniej warstwy połączony jest z każdym węzłem w następnej warstwie, lecz perceptrony znajdujące się w tej samej warstwie są wzajemnie niezależne. Każde połączenie posiada przypisaną wagę użytą do przeliczenia wartości wchodzącej do danego perceptronu z danego sąsiada z poprzedniej warstwy. Wewnątrz peceptronu obliczana jest suma iloczynów wyjść z poprzednich perceptronów i wag odpowiadających połączeniom, a następnie dla uzyskanej sumy obliczana jest wartość funkcji aktywacyjnej, która stanowi wartość wyjściową perceptronu. Przykładowa sieć wykorzystująca pojedynczy perceptron w pojedynczej warstwie ukrytej przedstawiona została na rys. 2.1.



Rysunek 2.1. Schemat perceptronu - Simplelearn

Bardziej rozbudowane metody wykorzystujące sieci neuronowe, jak np. CNN, wymagają dodatkowych kroków obliczeniowych związanych z wstępnym przetworzeniem danych wejściowych, aby były one przyswajalne dla wykorzystywanej sieci.

Analizując struktury danych wymagane przez poszczególne omówione powyżej rodzaje modeli, wyróżnić można następujące problemy napotykane podczas implementacji metod uczenia maszynowego:

- Wymagania wydajnościowe – są one ściśle powiązane ze złożonością obliczeniową wykorzystanych metod, wydajnością zastosowanego języka i wydajnością zastosowanej platformy sprzętowej. Docelowym efektem jest minimalizacja czasu wymaganego na uczenie modelu (chodź tutaj tolerowane są także długie czasy, szczególnie w przypadku dużych zestawów danch uczących) i czasu propagacji modelu (w przypadku czego minimalizacja czasu propagacji stanowi priorytet).

- Wymagania pamięciowe – wynikają one z wykorzystywanych platform sprzętowych i ich ograniczeń pamięciowych. Przykładem powyższego dylematu jest zastosowanie modeli uczenia maszynowego na platformach mobilnych i platformach systemów wbudowanych, gdzie obecne rozmiary pamięci RAM i pamięci masowej (szczególnie w przypadku platform wbudowanych) potrafią być wyraźnie ograniczone w stosunku do systemów komputerowych.

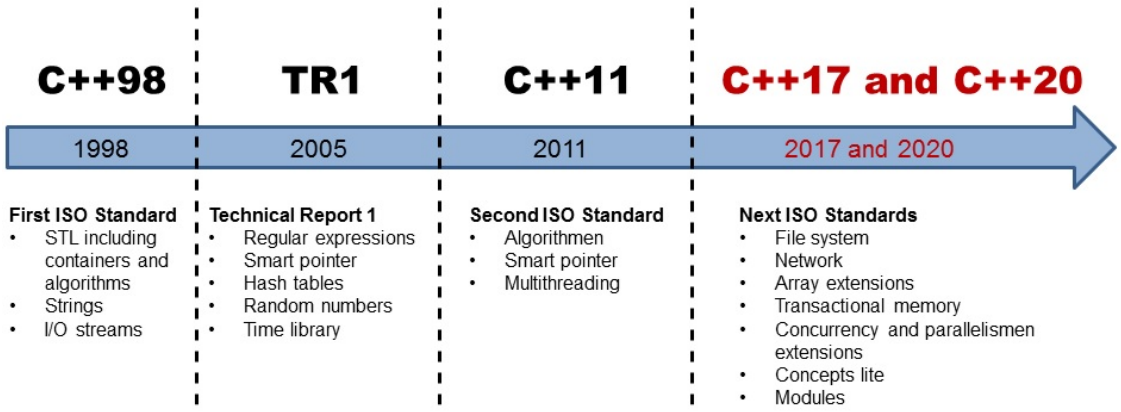
W trakcie rozwoju technologii uczenia maszynowego, postawiono stanowcze kroki w kierunku rozwiązywania powyższych problemów, aby sprostać narastającym wymaganiom związanym z coraz to nowymi i bardziej skomplikowanymi zastosowaniami sztucznej inteligencji. Dokonywano tego poprzez między innymi optymalizację algorytmów, dobór platform sprzętowych o wysokim taktowaniu, możliwym zrównolegleniu operacji, oraz wykorzystaniu wysoko wydajnych języków programowania, w szczególności języków mających możliwość wykorzystania wsparcia ze strony niskopoziomowych operacji.

2.2. Język C++ jako narzędzie do rozwiązania problemów uczenia maszynowego

Dostępne są różne języki i środowiska wspierające uczenie maszynowe, począwszy od języków takich jak Python, C++, Java czy Matlab. Jednak spośród wymienionych kandydatów szczególnie istotnym wyborem jest język C++.

C++ to język imperatywny charakteryzujący się silnym typowaniem, łączący programowanie niskopoziomowe dla konkretnych architektur z wysokopoziomowym programowaniem, w związku z czym oferuje programistom dużą kontrolę nad wykorzystaniem pamięci i możliwość optymalizacji w postaci m.in. dostosowywania wykorzystanych typów danych do wymagań funkcjonalnych tworzonej sieci, kontroli lokalizacji zmiennych (programista decyduje czy zmienna lub struktura znajdzie się na stosie czy stercie) oraz optymalizację czasów wywołań funkcji poprzez sugerowanie kompilatorowi utworzenia funkcji inline. W przeciwieństwie do języków skryptowych których kod jest interpretowany w trakcie wykonywania, takich jak Python i język środowiska Matlab, C++ jest językiem kompilowanym. Oznacza to, że program napisany w C++ przetwarzany jest z postaci tekstu do wykonawczego kodu binarnego dostosowanego do wybranej architektury procesora. Usuwa to całkowicie nadmiar złożoności obliczeniowej wykonywanego programu związanej z interpretacją poleceń i tłumaczeniem ich na język procesora danej platformy w trakcie wykonywania programu, gdyż jest to wykonywane tylko raz, na etapie kompilacji, dodatkowo pozwalając na zastosowanie przez kompilator mechanizmów optymalizacji dostępnych dla wybranej platformy.

Część mechanizmów z języka C++, wywodzących się jeszcze z języka C, pozwala na wykorzystanie wstawek kodu źródłowego w języku Assembler dla wybranego procesora, co zwiększa wydajność programu kosztem przenośności kodu. Dodatkowo niektóre platformy oferują API modułów akceleracji sprzętowej (jak np. system Android udostępniający *Neural Networks API, NNA* dla sieci neuronowych), co oferuje dodatkowe przyspieszenie czasu działania programu.



Rysunek 2.2. Multithreading in modern C++ - Modernes C++

Jedną z popularnych technik mających na celu znaczne zwiększenie wydajności modeli sztucznej inteligencji jest zrównoleglenie przetwarzania. Dostępność mechanizmów wielowątkowych dla procesorów (wprowadzonych w standardzie C++11 i

dalej rozwijanych, jak przedstawiono na rys. 2.2), oraz kompatybilność języka C++ z językiem CUDA pozwala wykonywać wiele obliczeń równolegle poprzez wykorzystanie wielu rdzeni lub oddelegowaniu części przetwarzania do karty (lub wielu kart) graficznej (gdzie ilość procesorów GPU znacząco przewyższa ilość rdzeni CPU). Dodatkowym atutem wykorzystania języka C++ przy tworzeniu modelu sztucznej inteligencji jest łatwa integracja z programami dedykowanymi do wysokiej wydajności, napisanymi w tym języku.

Wymienione wyżej mechanizmy i cechy charakterystyczne języka umożliwiają programistom znaczną optymalizację przygotowywanych rozwiązań sztucznej inteligencji, co przekłada się na bardziej efektywne zużycie pamięci, zabezpieczenie przed przeładowaniem stosu procesora, oraz krótsze czasy propagacji utworzonych modeli.

2.3. Cel powstania bibliotek

Implementacja mechanizmów pozwalających na tworzenie rozwiązań sztucznej inteligencji, z racji na swoją złożoność, wymagania dotyczące kompetencji twórców oraz konieczność optymalizacji jest czasochłonna i kosztowna. Tu z pomocą przychodzą biblioteki utworzone przez korporacje oraz społeczność programistów *open source*. Stanowią one gotowe zbiory mechanizmów (najczęściej pisane w sposób obiektowy, a więc ubrane w klasy posiadające określone zestawy metod), które są na bieżąco optymalizowane przez grupy programistów wykorzystujące je w prywatnych projektach lub pracy zawodowej. Oferują one możliwość wykorzystania gotowych modeli utworzonych w innych technologiach, a czasem także bezpośrednie przygotowanie modelu na podstawie odpowiednio sformatowanego i odpowiednio przystosowanego zestawu danych.

Użycie gotowych bibliotek nie tylko oszczędza koszta i przyspiesza tworzenie pożądanego rozwiązania sztucznej inteligencji, lecz także zapewnia większą niezawodność, gdyż elementy zawarte w bibliotece są implementowane, dokładnie testowane i poprawiane przez programistów o wysokich kompetencjach, jak m.in. w przypadku biblioteki TensorFlow posiadającej wsparcie od pracowników Google.

Więszkość bibliotek przeznaczonych do uczenia maszynowego, nawet wykorzystywanych w językach takich jak Python, napisana jest w języku C++, oferując API dostępne dla określonych języków docelowych. Niestety nie wszystkie biblioteki napisane w ten sposób oferują dostęp do całego API w języku C++ dla wykorzystujących je programów zewnętrznych, lub bywa on utrudniony i skomplikowany, co sprawia że w powszechnej praktyce część bibliotek dedykowanych dla języka C++ operuje na modelach przygotowanych w ramach innej, lub czasem nawet tej samej biblioteki, napisanch w innym języku. Częstym przypadkiem jest tutaj wykorzystanie właśnie języka Python do utworzenia grafu modelu lub modelu w formacie ONNX (ang. *Open Neural Network Exchange*).

W ramach analizy porównawczej w niniejszej pracy, porównywane będą biblioteki oferujące zarówno tworzeie modeli w ramach języka C++, jak i wymagające wykorzystania modeli z innego źródła.

Rozdział 3

Inżynieria danych eksperymentalnych i testowe szablony modeli

3.1. Omówienie danych eksperymentalnych

W celu zestawienia funkcjonalnego bibliotek uczenia maszynowego w języku C++ i przedstawienia przykładów konieczne było wybranie danych eksperymentalnych możliwych do wykorzystania jako porównawczy punkt odniesienia. W tym celu, dla pełnego przetestowania wybranych funkcjonalności przygotowano zestaw danych do problemu klasyfikacji binarnej oraz zadania regresji.

3.1.1. Dane klasyfikacyjne

Jako dane klasyfikacyjne wybrano bazę dotyczącą diagnostyki raka piersi „*Wisconsin Diagnostic Breast Cancer*" z listopada 1995 roku, w której zamieszczono wyniki obrazowania określone w sposób liczbowy. Autorami zestawu są Dr. Wiliam H. Wolberg, W. Nick Street oraz Olvi L. Mangasarian z Uniwersytetu Wisconsin [1]. Baza ta jest dostępna do pobrania z repozytorium Uniwersytetu Californii [2]. Dane mają następującą strukturę:

1) ID - numer identyfikacyjny pacjentki;
2) Diagnosis [*Malignant - M / Benign - B*] - charakter nowotworu, **zmienna odpowiedzi**;
3) Dane klasyfikujące:
    a) *Radius* - średnica guza;
    b) *Texture* - tekstura guza;
    c) *Perimeter* - obwód guza;
    d) *Area* - pole guza;
    e) *Smoothness* - gładkość, miara lokalnych różnic w promieniu guza;

f) *Compactness* - zwartość, wykorzystywana do oceny stadium guza;

g) *Concavity* - stopień wklęsłości miejsc guza;

h) *Concave points* - punkty wklęsłości guza;

i) *Symmetry* - symetria guza, pomagająca w ocenie charakteru przyrostu guza.

j) *Fractal dimention („coastline approximation" - 1)* - wymiar fraktalny pozwalający na ilościowy opis złożoności komórek nerwowych, umożliwiający stwierdzenie nowotworzenia się zbioru komórek.

Dla każdej ze zmiennych odpowiedzi została zebrana średnia wartość, odcyhelenie standardowe oraz średnia trzech największych pomiarów, gdzie każdy zestaw ustawiony jest sekwencyjnie (np. kolumna 3 - średni promień, kolumna 12 - odchylenie standardowe promienia, kolumna 22 - średnia trzech największych pomiarów promienia). Każda ze zmiennych ma charakter ciągły. Zredukowany zestaw danych, zawierający jedynie zmienne decyzyjne informujące o średnich wartościach znaleźć można jako dodatek do książki „*Biostatistics Using JMP: A Practical Guide*" autorstwa Trevora Bihla [3].

3.1.2. Dane regresyjne

Do demonstracji problemu regresji wykorzystano zestaw danych „IronGlutathione" dołączony do książki „Biostatistics Using JMP - A Practical Guide" autorstwa Trevora Bihla [3], dotyczące badań nad związkiem między zawartością żelaza, a α- i π-glutathonine-s-transferase w organiźmie człowieka. Obserwacje pochodzą z badań z 2012 roku. Zestaw posiada 90 obserwacji i składa się z 10 zmiennych:

1. *Age* - wiek badanej osoby;
2. *Gender* - płeć osoby;
3. *Alpha GST (ng/L)* - zawartość transferazy glutatoninowej typu α;
4. *pi GST (mg/L)* - zawartość transferazy glutatoninowej typi π;
5. *transferrin (mg/mL)* - zawartość transferyny;
6. *sTfR (mg/mL)* - zawartość rozpuszczalnego receptora transferyny;
7. *Iron (mg/dL)* - zawartość żelaza;
8. *TIBC (mg/dL)* - całkowita zdolność wiązania żelaza;
9. *%ISAF (Iron / TIBC)* - współczynnik nasycenia transferyny;
10. *Ferritin (ng/dL)* - zawartość ferrytyny;

Z racji na większą swobode w wyborze zmiennej odpowiedzi w przypadku danych regresyjnych, zdecydowano się na wybór ostatniej zmiennej (*Ferritin (ng/dL)*) jako przewidywaną zmienną odpowiedzi.

3.2. Charakterystyka i przetwarzenie danych

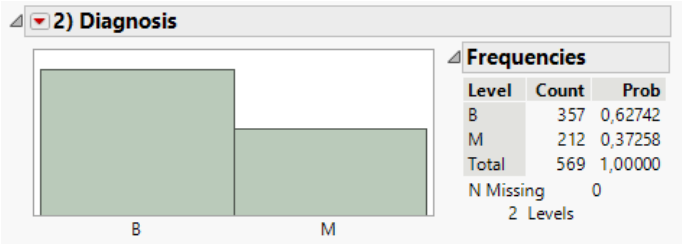
W celu przeprowadzenia procesu uczenia maszynowego, jednym z najistotniejszych kroków jakie należy podjąć jest wstępne zaznajomienie się z zestawem danych i jego analiza pod kątem rozkładu poszczególnych zmiennych oraz prawdopodobieństw. W tym celu wykorzystane zostało oprogramowanie JMP.

3.2.1. Dane klasyfikacyjne

3.2.1.1. Analiza rozkładu danych

Proces analizy rozkładu rozpoczęty został od przyjrzenia się zmiennej odpowiedzi (*Diagnosis*). Rysunek 3.1 przedstawia uzyskany histogram, wraz z tabelą określającą ilość obserwacji danej klasy i współczynnik prawdopodobieństwa przynależności odpowiedzi do danej klasy. Zauważyć można, że dla użytego zestawu danych ilość zarejestrowano 357 obserwacji łagodnego raka piersi, a jego prawdopodobieństwo przynależności do klasy *Benign* wynosi $\approx$ 62,7%, natomiast do klasy *Malignant* przynależało 212 obserwacji z prawodpodobieństwem $\approx$ 37,3%.

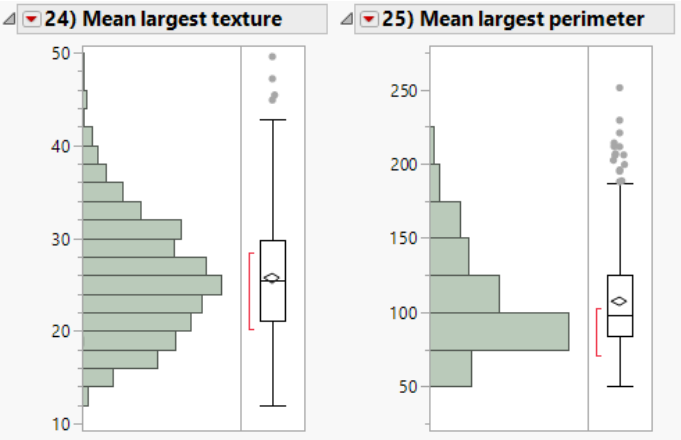
Podczas analizy histogramów zmiennych decyzyjnych, stwierdzono że znaczna ilość ma charakter prawostronnie skośny oraz występują dla nich obserwacje odstające, o czym informuje znajdujący się po prawej stronie histogramu wykres okienkowy (ang. *box graph*), co przedstawiono na rysunku 3.2. Wyjątkiem okazała się zmienna *Mean Largest Concave Points*), która mimo lekkiej skośności, okazała się nie posiadać obserwacji odstających. Na podstawie tych informacji stwierdzono, że aby przygotować dane w odpowiedni sposób do procesu uczenia należy przeprowadzić ich czyszczenie oraz normalizację rozkładu.



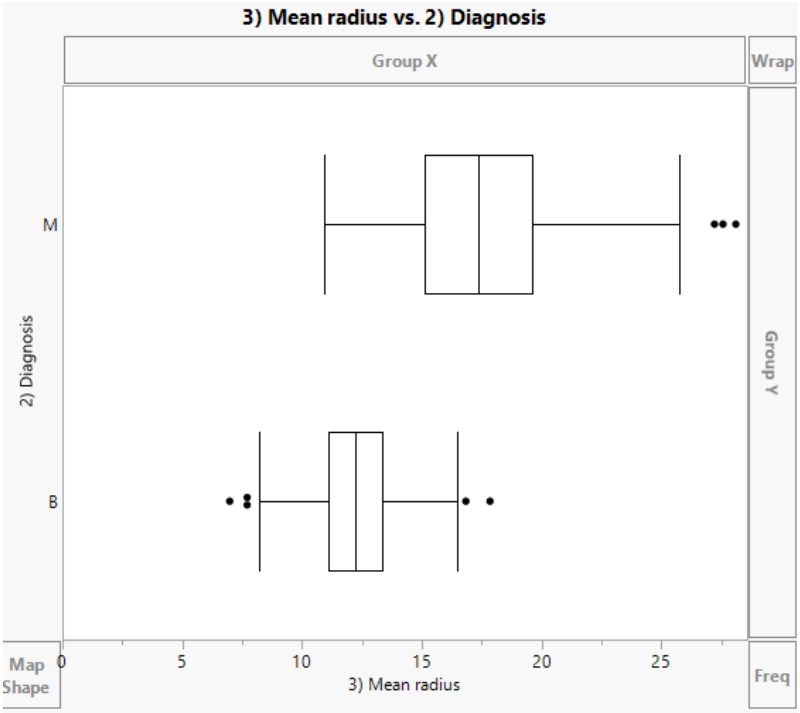
Rysunek 3.1. Histogram rozkładu zmiennej odpowiedzi

3.2.1.2. Czyszczenie i normalizacja rozkładu danych

Na pełny zestaw danych składa się 569 obserwacji. Podczas wstępnej analizy stwierdzono istnienie 13 brakujących wartości dla regresora *Std err concave points*, dla których przyjęto wartość średnią z całej kolumny. Głównym problemem okazały się obserwacje odstające oraz skośności rozkładu. Do analizy obserwacji odstających wykorzystano wykresy okienkowe, gdzie oś Y reprezentowała zmienną odpowiedzi, natomiast oś X czyszczoną zmienną decyzyjną. Przykładowy wykres został przedstawiony na rysunku 3.3. Ze względu na bardzo małą ilość obserwacji zdecydowano się rozpocząć proces przystosowywania danych do uczenia poprzez normalizację ich rozkładu, aby zminimalizować lub wyeliminować konieczność usunięcia danych odstających.



Rysunek 3.2. Przykłady histogramów zmiennych decyzyjnych

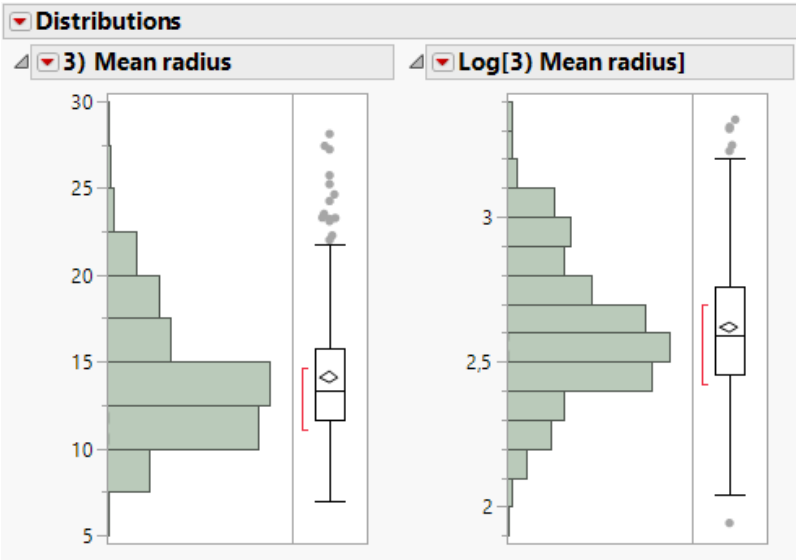


Rysunek 3.3. Przykład analizy obserwacji odstających dla poszczególnych klas zmiennej odpowiedzi

W pierwszym podejściu zdecydowano się na zastosowanie transformacji logarytmicznej dla wszystkich zmiennych decyzyjnych i porównanie charakterystyk uzyskanych rozkładów z oryginalnymi. Zmienna *Mean largest concave points* okazała się posiadać rozkład bardzo zbliżony do standardowego, w związku z czym wyłączono ją z dalszej analizy normalizacji. Przykładowe wyniki przedstawiono na rysunku 3.4. Transformacja ta okazała się skutecznym rozwiązaniem jedynie dla następujących zmiennych:

1. *Mean radius*;
2. *Mean texture*;
3. *Mean perimeter*,

4. *Mean area*;
5. *Mean smoothness*;
6. *Mean symmetry*;
7. *Std err texture*;
8. *Std err smoothness*;
9. *Std err compactness*;
10. *Std err concave points*;
11. *Mean largest texture*;
12. *Mean largest smoothness*;
13. *Mean largest compactness*.

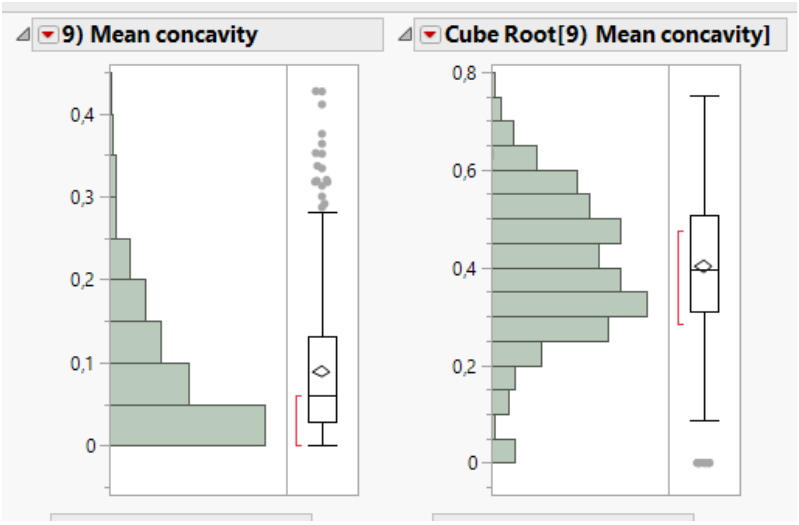


Rysunek 3.4. Porównanie rozkładu danych przed i po transformacji logarytmicznej.

W drugim kroku podjęto próbę wykorzystania transformacji pierwiastkiem sześciennym dla pozostałych zmiennych decyzyjych, ze względu na jej skuteczność dla danych o rozkładzie prawoskośnym. Rysunek 3.5. przedstawia porównanie rozkładu zmiennej *Mean concavity* przed i po transfromacji pierwiastkiem sześciennym. Pomyślnie znormalizowano rozkład następujących zmiennych:

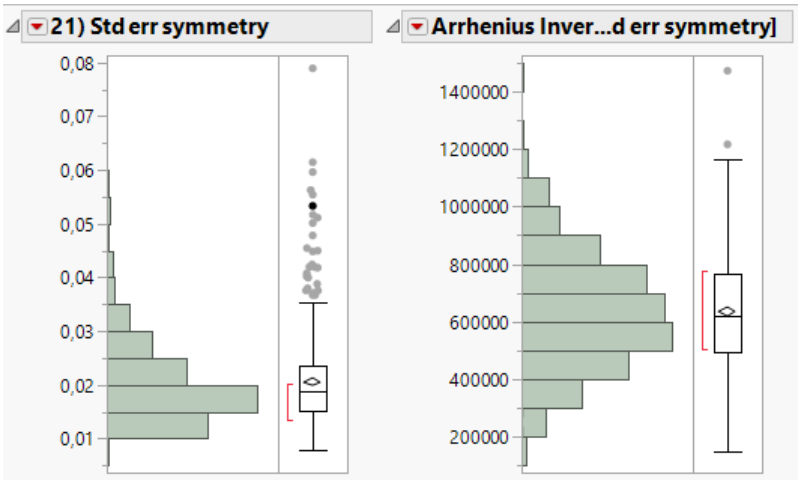
1. *Mean compactness*;
2. *Mean concavity*;
3. *Mean concave points*;
4. *Std err concavity*;
5. *Mean largest radius*;

6. *Mean largest perimeter*;
7. *Mean largest concavity*;
8. *Mean largest symmetry.*



Rysunek 3.5. Porównanie rozkładów danych przed i po zastosowaniu transformacji pierwiastkiem sześciennym.

Ostatecznym krokiem okazało się zastosownie odwrotnej transformacji Arrheniusa, Niestety część z uzyskanych zmodyfikowanych zmiennych decyzyjnych zachowała częściowy skośny rozkład, jednak inne przetestowane transformacje, jak m.in. pierwiastek kwadratowy, potęga kwadratowa, logarytm x+1, logarytm dziesiętny, funkcja potęgowa, funkcja wykładnicza, przyniosły rezultaty porównywalne lub gorsze od uzyskanego w wyniku w/w odwrotnej transformacji Arrheniusa. Rysunek 3.6 przedstawia porównanie uzyskanych rozkładów.



Rysunek 3.6. Porównanie uzyskanych rozkładów danych przed i po odwrotnej transformacji Arrheniusa.

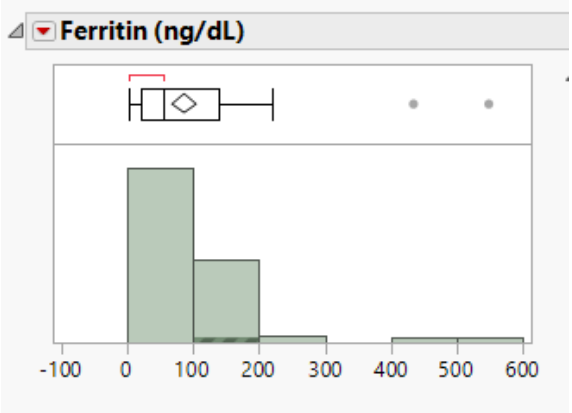
Ze względu na bardzo małą ilość obserwacji, zdecydowano się na zachowanie wszystkich obserwacji odstających, aby zapobiec utracie informacji i zmianie uzyskanych w procesie normalizacji rozkładów. W celu zachowania kompatybilności z

bibliotekami omawianymi w niniejszej pracy, przekodowano zmienną odpowiedzi na wartości liczbowe.

3.2.2. Dane regresyjne

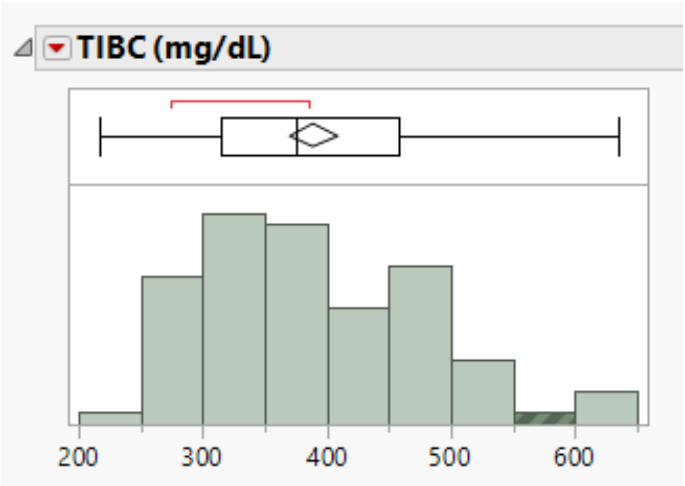
3.2.2.1. Analiza rozkładu danych

Podobnie jak w przypadku danych klasyfikacyjnych, analizę rozpoczęto od zapoznania się z rozkładem wybranej zmiennej odpowiedzi. Zauważono że posiada ona rozkład skrajnie prawostronny, co przedstawiono na rysunku 3.7.

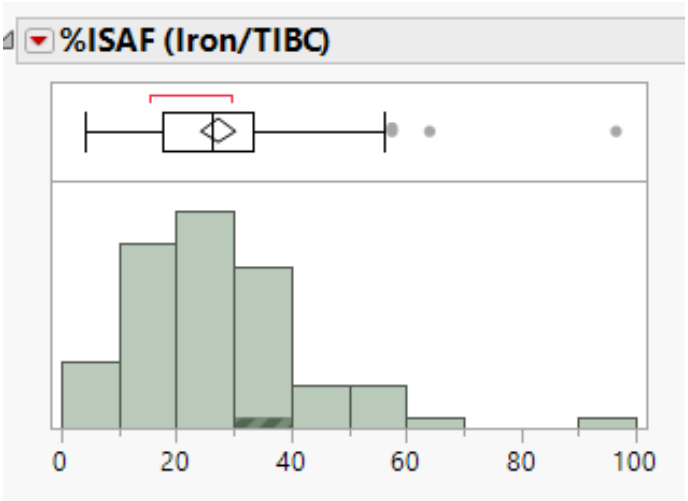


Rysunek 3.7. Wykres rozkładu zmiennej odpowiedzi dla zestawu regresyjnego

Na wykresie okienkowym zawartym nad histogramem rozkładu zauważyć można wystąpienie dwóch obserwacji odstających. Podczas dalszej analizy, spostrzeżono podobny problem w przypadku zmiennych *%ISAF*, *Iron*, *sTfR*, *Transferrin* oraz szczególnie *Alpha GST*. Pozostałe zmienne charakteryzują się rozkładem zbliżonym do krzywej Gaussa, nie posiadając obserwacji zaklasyfikowanych jako odstające. Rysunek 3.8 przedstawia dwa przykładowe histogramy rozkładów zmiennych decyzyjnych.

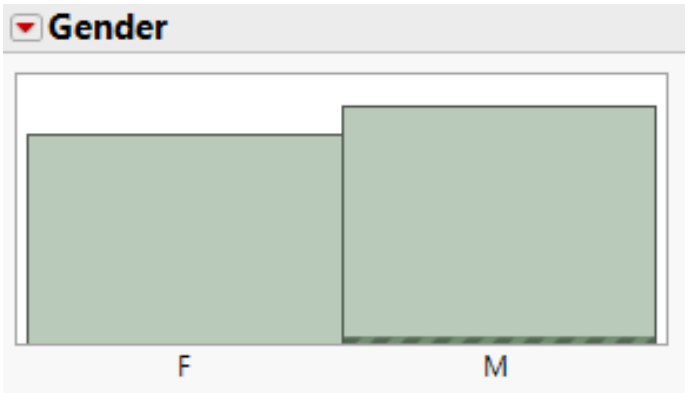


Rysunek 3.8. Przykłady rozkładu zmiennej decyzyjnej cz. 1



Rysunek 3.9. Przykład rozkładu zmiennej decyzyjnej cz. 2

Pojedyncza zmienna - *Gender* - posiada dychotomiczny charakter, aczkolwiek okazuje się relatywnie bardzo zrównoważona, posiadając rozkład na poziomie przynależności w 46,7% do klasy F reprezentującej kobiety oraz 53,3% do klasy M odpowiadającej mężczyznom. Rozkład tej zmiennej przedstawiono na rysunku 3.10.



Rysunek 3.10. Rozkład zmiennej Gender

3.2.2.2. Czyszczenie i normalizacja rozkładu danych

W trakcie przeglądu obserwacji, zauważono pojedynczą obserwację z brakującą wartością zmiennej *Ferritin*. Ze względu na wystąpienie tylko jednego takiego wpisu na 85 obserwacji, zdecydowano się na usunięcie jej. Pozostałymi kwestiami wymagającymi zaadresowania okazała się normalizacja rozkładu części zmiennych i decyzja o działaniu względem wartości odstających.

W celu zmniejszenia ilości obserwacji odstających, postanowiono rozpocząć następny etap od problemu normalizacji rozkładu. Następujące zmienne, ze względu na ich obecną charakterystykę nie zostały poddane żadnym przekształceniom:

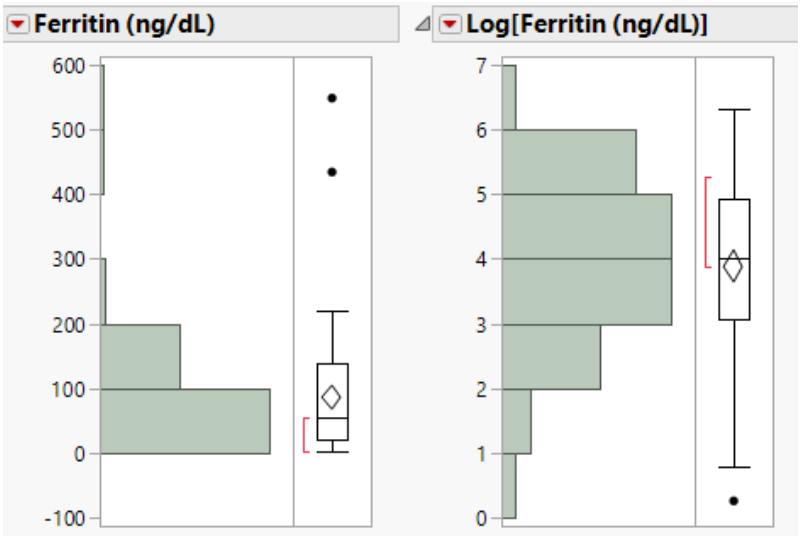
1. *Age*;
2. *Gender*;

3. *TIBC*.

Dla pozostałych zmiennych decyzyjnych zastosowano trzy rodzaje transformacji opisanych w sekcji omawiających dane klasyfikacyjne. Pierwszą z nich była obliczenie logarytmu z wartości zmiennej, które przyniosło zadowalający efekt dla zmiennych:

1. *alpha GST*;
2. *pi GST*;
3. *sTfR*;
4. *Ferritin* (zmienna odpowiedzi).

Rysunek 3.11 przedstawia przykład uzyskanej zmiany rozkładu dla zmiennej odpowiedzi.

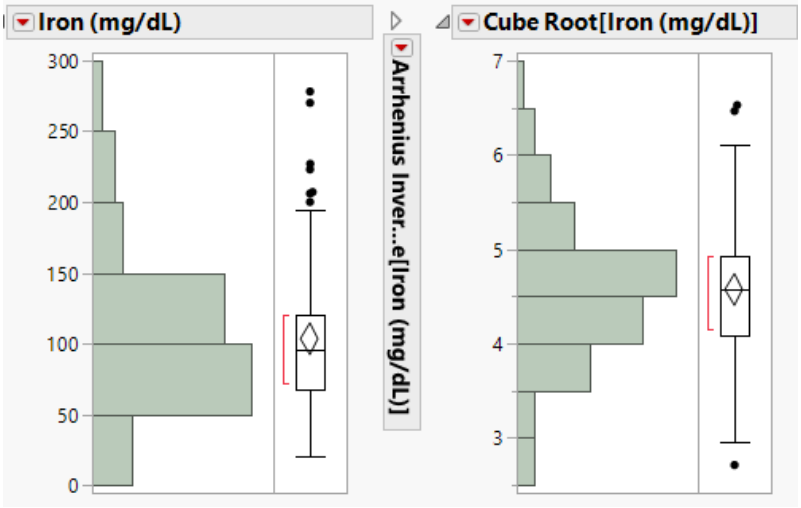


Rysunek 3.11. Wpływ transformacji logarytmicznej na rozkład zmiennej odpowiedzi

Drugą transformacją o zadowalających wynikach okazało się zastosowanie pierwiastka kwadratowego, co pokazano na rysunku 3.12. Wykorzystano ją do normalizacji rozkładu zmiennych:

1. *Transferrin*;
2. *Iron*;
3. *%ISAF*.

Transformacja odwrotnym wzorem Arrheniusa okazała się nieskuteczna na wszystkich zmiennych, uzyskując gorsze efekty niż pozostałe przedstawione wyżej metody. Z racji na niewielką ilość obserwacji, oraz stosunkowo małą liczbę wartości odstających, zdecydowano się na pozostawienie ich do procesu uczenia. W celu kompatybilności z omawianymi bibliotekami, przekodowano zmienną dychotomiczną na wartości liczbowe.



Rysunek 3.12. Normalizacja rozkładu za pomocą transformacji pierwiastkiem kwadratowym.

3.3. Szablony docelowych modeli dla zadanych danych eksperymentalnych

W trakcie analizy zestawów danych wybrany został przedstawiony poniżej zestaw metod dla których wykonano i podsumowano testy praktyczne. Szablony struktury rozwiązań, takie jak np. wybór zmiennych uczestniczących w procesie uczenia, lub struktura sieci neuronowej zostały ustalone w sposób empiryczny z wykorzystaniem programu do uczenia maszynowego JMP.

3.3.1. Regresja logistyczna

Badanie zależności w modelu regresji logistycznej odbyło się z wykorzystaniem wykresu wpływu zmiennej decyzyjnej na zmienną odpowiedzi opartego o p-wartość. Jako próg pozwalający na odrzucenie hipotezy zerowej (hipotezy o braku wypływu zmiennej na odpowiedź) przyjęto 0,05 jednostek. Rysunek 3.13 przedstawia w/w wykres wraz z p-wartościami dla poszczególnych zmiennych. Zauważyć można, że dla części zmiennych nie została wyznaczona p-wartość – oznacza to, że część zmiennych jest ze sobą skorelowanych.

Pierwszym krokiem w wybraniu istotnych zmiennych było usunięcie zmiennych skorelowanych, drugim natomiast stopniowe usuwanie zmiennych o p-wartości powyżej określonego progu. Rysunek 3.14 przedstawia listę wraz z wykresem kolumnowym istotnych regresorów. Ich lista, wraz z odpowiadającymi im p-wartościami została umieszczona w tabeli 3.1.

Nazwa zmiennej	p-wartość
Log mean largest texture	0,00000
Log mean largest compactness	0,00000
Cube root mean largest symmetry	0,00001
Arrhenius inverse std err symmetry	0,00005
Arrhenius inverse std err radius	0,00018
Cube root mean concave points	0,00056
Cube root mean largest concavity	0,00069
Log std err texture	0,00252
Cube root mean largest perimeter	0,00526
Log mean smoothness	0,04867
Log mean radius	0,04884

Tabela 3.1. Lista istotnych regresorów

Source	LogWorth	PValue
AI Mean largest area	951,928	0,00000
QR Mean concavity	610,420	0,00000
AI Std err area	476,593	0,00000
Log Std err smoothness	358,978	0,00000
AI Mean fractal dimention	218,578	0,00000
AI Mean largest fractal dimention	.	0,00000
QR Mean largest symmetry	.	.
Mean largest concave points	.	.
QR Mean largest concavity	.	.
Log Mean largest compactness	.	.
Log Mean largest smoothness	.	.
QR Mean largest perimeter	.	.
Log Mean largest texture	.	.
QR Mean largest radius	.	.
AI Std err fractal dimention	.	0,00000
AI Std err symmetry	.	0,00000
Log Std err concave points	.	0,00000
QR Std err concavity	.	.
Log Std err compactness	.	0,00000
AI Std Err perimeter	.	0,00000
Log Std err texture	.	0,00000
AI Std err radius	.	0,00000
Log Mean symmetry	.	0,00000
QR Mean concave points	.	.
QR Mean compactness	.	.
Log Mean smoothness	.	.
Log Mean area	.	.
Log Mean perimeter	.	.
Log Mean texture	.	0,00000
Log Mean radius	.	0,00000

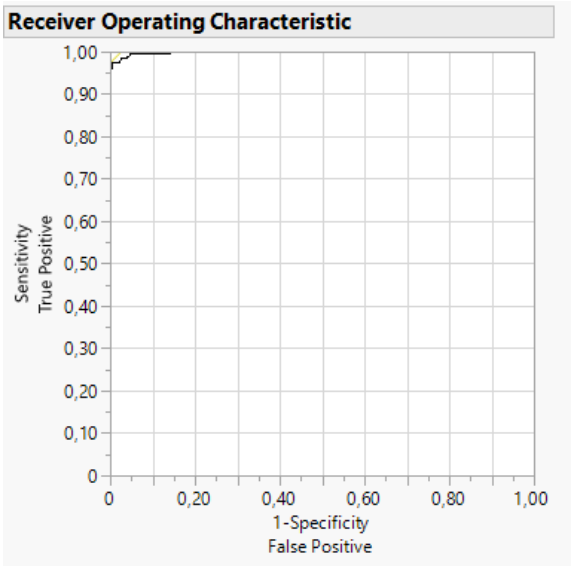
Rysunek 3.13. Wykres p-wartości dla całego zestawu zmiennych decyzyjnych.

Dla wybranego zestawu zmiennych model osiągnął dokładność na poziomie R^2 = 0.9401. Zgodnie z macierzą pomyłek, 207 obserwacji typu *Malignant* oraz 335 obserwacji *Benign* zostało zaklasyfikowanych poprawnie. Oznacza to, że model uzy-

skał tylko 2 wyniki typu *false-positive* (prawdopodobieństwo 0,6%) i 5 wyników typu *false-negative* (prawdopodobieństwo 2,4%) dla danych treningowych. Ze względu na mały zestaw obserwacji, ryzyko przeuczenia jest znikome, w związku z czym nie wytypowano zestawu danych walidacyjnych. Rysunek 3.15 przedstawia krzywą charakterystyczną odbiornika dla modelu.

Source	LogWorth	PValue
Log Mean largest texture	10,190	0,00000
Log Mean largest compactness	6,684	0,00000
QR Mean largest symmetry	4,873	0,00001
AI Std err symmetry	4,304	0,00005
AI Std err radius	3,748	0,00018
QR Mean concave points	3,255	0,00056
QR Mean largest concavity	3,159	0,00069
Log Std err texture	2,598	0,00252
QR Mean largest perimeter	2,279	0,00526
Log Mean smoothness	1,313	0,04867
Log Mean radius	1,311	0,04884

Rysunek 3.14. Wykres i p-wartości istotnych zmiennych decyzyjnych



Rysunek 3.15. Krzywa charakterystyczna odbiornika (ROC) dla modelu regresji logistycznej

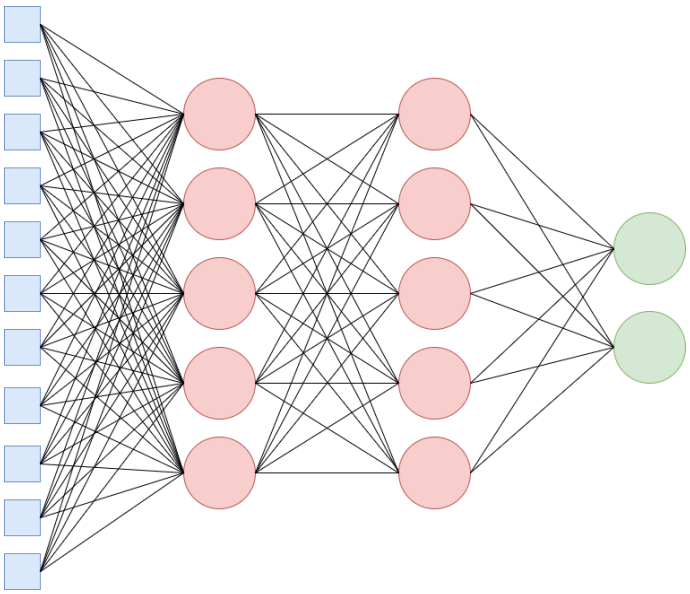
3.3.2. Głęboka sieć neuronowa

Do przygotowania sieci neuronowej wykorzystano zestaw zmiennych zawartych w tabeli 3.1. Dane zostały losowo podzielony na dane uczące i walidacyjne w propocji 80% do 20%. W wyniku prób i błędów, optymalny model uzyskano przy strukturze przedstawionej w tabeli 3.2. Graficzny schemat struktury został także przedstawiony na rysunku 3.16.

Środowisko JMP nie udostępnia informacji o funkcji aktywacji warstwy wyjściowej, w związku z czym w tabeli 3.2 została ona pominięta. Dla ziarna o wartości 1234 uzyskano model którego statystyka R^2 dla danych treningowych wyniosła 0.966268, natomiast dla danych testowych 0.9924547. Trafność dla losowo wybranego zestawu

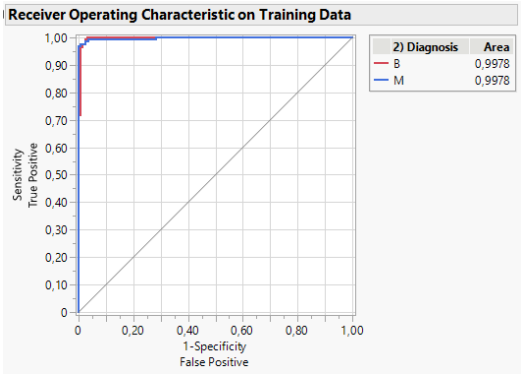
Typ warstwy	ilość neuronów	aktywacja
ukryta	5	tangens hiperboliczny
ukryta	5	tangens hiperboliczny
wyjściowa	2	—

Tabela 3.2. Struktura modelu sieci neuronowej

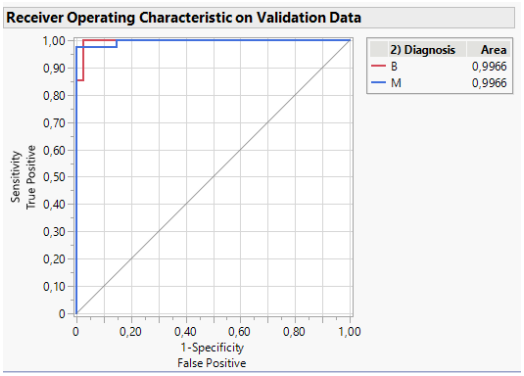


Rysunek 3.16. Schemat struktury sieci

testowego wyniosła 100%, natomiast dla danych uczących napotkano 5 przypadków *false-negative* (prawdopodobieństwo 3%) oraz 1 przypadek *false-positive* (prawdopodobieństwo 0,4%). Rysunki 3.17 oraz 3.18 przedstawiają krzywe charakterystyczne odbiornika dla zestawu testowego i walidacyjnego.



Rysunek 3.17. Krzywa charakterystyczna odbiornika dla zestawu testowego



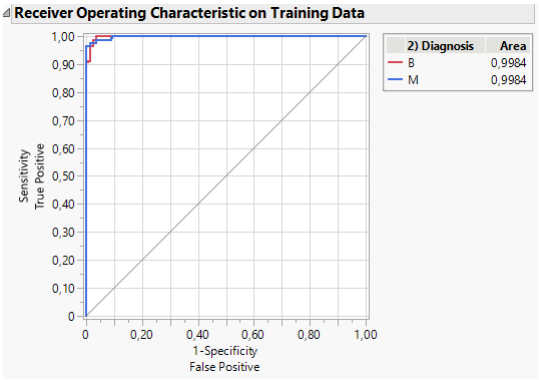
Rysunek 3.18. Krzywa charakterystyczna odbiornika dla danych walidacyjnych

3.3.3. Maszyna wektorów nośnych

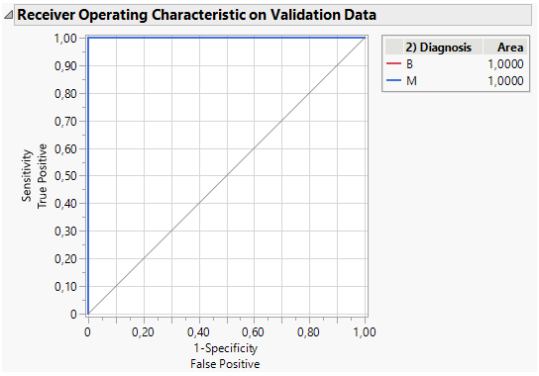
Do predykcji diagnozy wykorzystano ten sam zestaw regresorów, zawartych w tabeli 3.1. Ponownie w celu walidacji użyto metody wybrania losowego zestawu walidacyjnego spośród dostarczonych danych, w proporcji 80% obserwacji uczących i 20% testowych, z użyciem wartości 1234 dla ziarna generatora liczb pseudolosowych. Jako funkcję jądra maszyny wektorów nośnych (ang. Support Vector Machine, SVM) wybrano *Radial Basis Function*, która jest domyślnym wyborem dla SVM w środowisku JMP.

Zmienna decyzyjna	wartość X
Log mean largest texture	3,217
Log mean largest compactness	-1,5504
Cube root mean largest symmetry	0,65891
Arrhenius inverse std err symmetry	635100
Arrhenius inverse std err radius	38170
Cube root mean concave points	0,33665
Cube root mean largest concavity	0,5951
Log std err texture	0,1049
Cube root mean largest perimeter	4,7045
Log mean smoothness	-2,3502
Log mean radius	2,6191

Tabela 3.3. Wartości składowych X modelu dla poszczególnych zmiennych decyzyjnych



Rysunek 3.19. Krzywa charakterystyczna odbiornika dla danych uczących modelu SVM

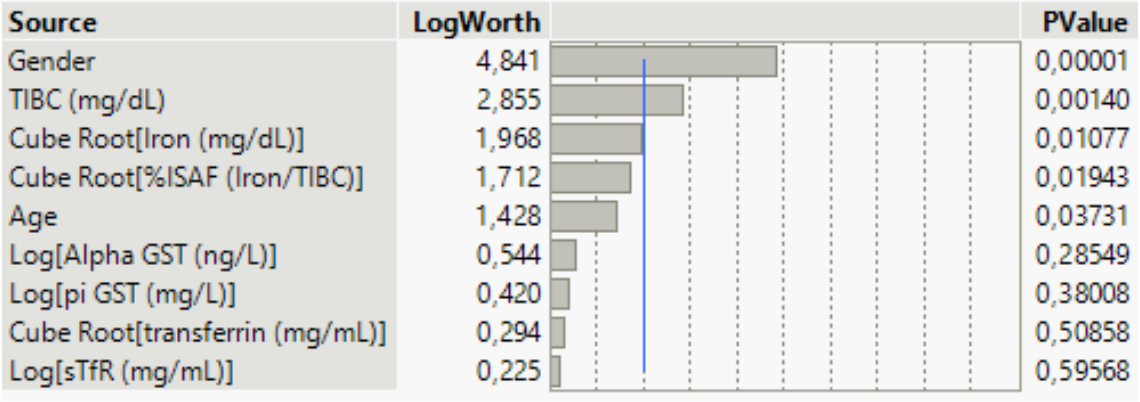


Rysunek 3.20. Krzywa charakterystyczna odbiornika dla danych walidacyjnych modelu SVM

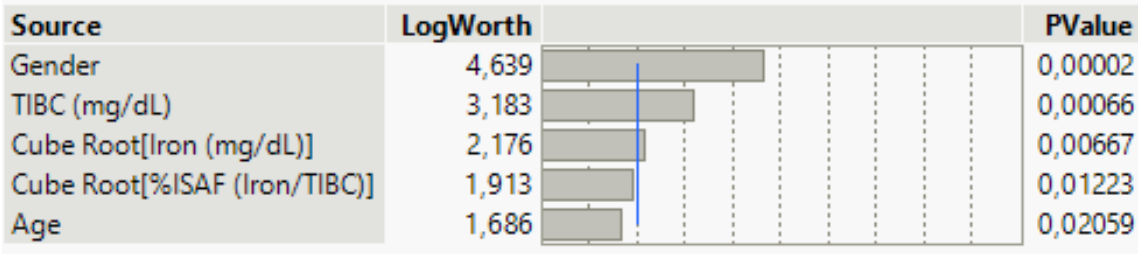
Utworzony w ten sposób model posiada generalizowaną statystykę R^2 na poziomie 0.97161 dla zestawu walidacyjnego, i uzyskał wskaźnik błędnej klasyfikacji wynoszący 0% dla danych testowych, oraz 1,3% dla danych uczących. Tabela 3.3 przedstawia wartości X dla poszczególnych regresorów. Rysunki 3.19 oraz 3.20 przedstawiają krzywe charakterystyczne odbiornika dla uzyskanego modelu.

3.3.4. Regresja liniowa

Podobnie jak w przypadku regresji logistycznej, do ustalenia które z regresorów mają największy wpływ na zmienną odpowiedzi zastosowano wykres wpływu poszczególnych zmiennych oparty o p-wartość. Jako próg zaakceptowania parametru do procesu uczenia zdecydowano się na wybranie p-wartości wynoszących poniżej 0,05 jednostek. Rysunek 3.21 przedstawia wykres dla wszystkich zmiennych, natomiast rysunek 3.22 ukazuje wykres zawierający jedynie wybrane zmienne.



Rysunek 3.21. Wykres p-wartości dla wszystkich zmiennych



Rysunek 3.22. Wykres p-wartości dla zmiennych wybranych do procesu uczenia

W procesie eliminacji regresorów o p-wartości przekraczającej wybrany próg akceptacji, wybrano następujące zmienne do procesu uczenia:

Nazwa zmiennej	p-wartość
Gender	0.00002
TIBC	0.00066
Cube Root Iron	0.00667
Cube Root %ISAF	0.01223
Age	0.02059

Tabela 3.4. Wybrane zmienne decyzyjne i ich p-wartości

Nazwa zmiennej	waga
Intercept	8,498959
Age	0.0201022
Gender[F - M]	-0.959795
Cube Root Iron	2,9461861
TIBC	-0.014182
Cube Root %ISAF	-4,356345

Tabela 3.5. wartości wag zmiennych decyzyjnych

W wyniku uczenia uzyskano model którego wartość metryki R^2 wyniosła 0.4654525, co sugeruje że dane są słabo aproksymowalne liniowo. Tabela 3.5 przedstawia nauczone wagi poszczególnych regresorów.

Rozdział 4

Biblioteka Shogun

4.1. Wprowadzenie

Shogun to darmowa biblioteka do uczenia maszynowego o otwartym źródle, napisana w C++ i udostępniana według licencji *BSD 3-clause* [4]. Posiada ona interfejsy dla różnych języków, w tym Python, Ruby czy C#, jednak pozwala ona na jej użycie także w jej natywnym języku. Skupia się ona na problemach klasyfikacji oraz regresji.

4.2. Formaty źródeł danych

Podstawową klasą pozwalającą na załadowanie danych do biblioteki Shogun jest klasa *std::vector* z standardowej biblioteki szablonowej (ang. *Standard Template Library, STL*) języka C++. W związku z tym, do pobrania danych dla programu realizującego nauczanie i pracę z modelem możliwe jest wykorzystanie dowolnego mechanizmu (np. odczytu z pliku, pobranie danych z sieci czy innego urządzenia) które finalnie przetworzy je do postaci wektora, lecz należy ten mechanizm dostarczyć we własnym zakresie. Popularnym wyborem do przechowywania informacji uczących jest plik o ustrukturyzowanym formacie CSV, dla którego biblioteka Shogun posiada dedykowane wsparcie [5]. Obwarowane jest ono jednak pewnymi wymaganiami:

- **Plik musi zawierać jedynie dane numeryczne** - wprzypadku występowania wartości tekstowych, należy wykonać przetwarzanie wstępne mające na celu ich zamianę na wartości liczbowe (np. w przypadku klas decyzyjnych zmiennej odpowiedzi sugerowane jest zastosowanie kodowania *one-hot*). Niestety ten wymóg nie pozwala na przechowywanie etykiet wraz z danymi.

- **Jako separator należy użyć przecinka** - mimo iż sam format, jak i wiele programów komercyjnych do pracy z danymi, jak np. Microsoft Excel, JMP, itp., pozwalają na zastosowanie innych separatorów, takich jak średnik, dla biblioteki Shogun należy zastosować w formie separatora przecinek;

- **Liczby rzeczywiste powinny być zapisywane z użyciem kropki jako separatora dziesiętnego** - wynika to ze specyfiki języka C++ (jak i wielu

innych języków), że domyślne mechanizmy wymuszają użycie kropki jako separatora dziesiętnego, i oczekują jej w przypadku parsowania liczby rzeczywistej z postaci ciągu znakowego odczytanego z pliku, do postaci wartości liczbowej.

Do odczytu i parsowania danych z pliku CSV wykorzystywana jest klasa *shogun::CCSVFile*, której wynik następnie ładowany jest do klasy *shogun::SGMatrix*. Ze względu na zapis odczytanych danych w kolejności według kolumn. do wykorzystania ich w procesie uczenia konieczna jest transpozycja, a następnie rozdzielenie macierzy na dwie części, z których jedna zawiera regresory, a druga wartości zmiennej odpowiedzi. Przykładowy fragment kodu realizujący to zadanie zamieszczony został na listingu 4.1. Po prawidłowym rozgraniczeniu fragmentów danych, należy przeprowadzić ponowną transpozycję do postaci akceptowalnej przez algorytmy uczenia, oraz obudować dane klasami na których operują docelowe metody uczenia, takimi jak *CDenseFeatures*, *CMulticlassLabels* czy *CRegressionLabels*, co zostało ukazane na listingu 4.2.

Listing 4.1. Przykładowa funkcja do odczytu i przygotowania danych z pliku CSV dla biblioteki Shogun

```
1  #pragma once
2
3  #include <shogun/base/init.h>
4  #include <shogun/base/some.h>
5  #include <shogun/io/File.h>
6
7  // pomocnicze pośrednie opakowanie na zestaw danych
8  struct Dataset
9  {
10     shogun::SGMatrix<float64_t> inputs;
11     shogun::SGMatrix<float64_t> outputs;
12 };
13
14 // pomocnicza struktura określająca pozycję zmiennej odpowiedzi
15 enum class LabelPos
16 {
17     FIRST,
18     LAST
19 };
20
21 inline Dataset readShogunCsvData(std::string filename, LabelPos labelPos)
22 {
23     using namespace shogun;
24     using Matrix = SGMatrix<float64_t>;
25
26     Dataset ret;
27
28     // odczytanie surowej zawartości pliku csv i sparsowanie jej do
29     // macierzy
30     auto csvFile = some<CCSVFile>(filename);
31     Matrix data;
32     data.load(csvFile);
33     // transpozycja do postaci docelowej dla człowieka
34     // (działanie na kolumnach)
35     Matrix::transpose_matrix(data.matrix, data.num_rows, data.num_cols);
36     // podział macierzy na część regresorów i zmiennej odpowiedzi
37     switch(labelPos)
```

```
38      {
39          case FIRST:
40              ret.inputs = data.submatrix(1, data.num_cols).clone();
41              ret.outputs = data.submatrix(0, 1).clone();
42              break;
43          case LAST:
44              ret.inputs = data.submatrix(0, data.num_cols - 1).clone();
45              ret.outputs =
46                  data.submatrix(data.num_cols - 1, data.num_cols).clone();
47              break;
48      };
49      // ponowna transpozycja do positaci docelowej dla algorytmów uczących
50      // (operowanie na wierszach)
51      Matrix::transpose_matrix(ret.inputs.matrix, ret.inputs.num_rows,
52                               ret.inputs.num_cols);
53      return ret;
54  }
```

Listing 4.2. Funkcja przepakowujące dane do kontenerów docelowych

```
1   #pragma once
2
3   #include <inc/shogun/csv.hpp>
4   #include <inc/shogun/linear.hpp>
5   #include <inc/shogun/logistic.hpp>
6   #include <inc/shogun/svm.hpp>
7   #include <inc/shogun/neural.hpp>
8
9   #include <shogun/base/init.h>
10
11  inline void shogunModels()
12  {
13      using namespace shogun;
14
15      init_shogun_with_defaults();
16
17      // odczytanie danych we własnym pośrednim typie danych
18      auto classificationDatasetTemp =
19          readShogunCsvData("wdbc_data_with_labels.csv", LabelPos::FIRST);
20      auto regressionDatasetTemp =
21          readShogunCsvData("IronGlutathione.csv", LabelPos::LAST);
22      // rozdzielenie danych na regresory i zmienne odpowiedzi
23      auto classificationFeatures =
24          some<CDenseFeatures<float64_t>>(
25              classificationDatasetTemp.inputs);
26      auto classificationLabels =
27          some<CBinaryLabels>(
28              classificationDatasetTemp.outputs);
29      auto regressionFeatures =
30          some<CDesneFeatures<float64_t>>(
31              regressionDatasetTemp.inputs);
32      auto regressionLabels =
33          some<CRegressionLabels>(
34              regressionDatasetTemp.outputs);
35
36      // wywołanie modeli
37      shogunLinear(regressionFeatures, regressionLabels);
38      shogunLogistic(classificationFeatures, classificationLabels);
```

```
39      shogunSVM(classificationFeatures, classificationLabels);
40      sharkNeural(classificationFeatures, classificationLabels);
41
42      exit_shogun();
43  }
```

4.3. Metody przetwarzania i eksploracji danych

4.3.1. Normalizacja

Biblioteka dostarcza możliwość normalizacji typu min-max, zapewniając że dane mieścić się będą w przedziale jednostkowym, za pomocą klasy *shogun::CRescaleFeatures*. Klasa pozwala na ponowne wykorzystanie dla danych o tych samych nauczonych zmiennych statystycznych. Posiada ona dwie główne metody:

- *fit()* - pozwalającą na nauczenie normalizatora statystyk danych;
- *transform()* - pozwalającą na normalizację obserwacji.

W przypadku niektórych algorytmów oferowanych przez Shogun, normalizacja jest jednym z pierwszych wykonywanych kroków, w związku z czym nie zawsze jest potrzeba wykonania jej we wstępnym przetwarzaniu. Informacja o takim przypadku powinna być zawarta w dokumentacji danej metody. Listing 4.3 pokazuje jak wykorzystać wyżej wspomnianą klasę do zrealizowania normalizacji. Zarówno przedstawiona klasa jak i funkcja zawarta na listingu realizują normalizację w miejscu zapisu macierzy regresorów, w związku z czym nie ma potrzeby nadpisywania elementy ją przechowującego.

Listing 4.3. Przykład funkcji wykonującej normalizację

```
1  #pragma once
2
3  #include <shogun/preprocessor/RescaleFeatures.h>
4
5  inline void normalize(auto& inputs)
6  {
7      using namespace shogun;
8
9      // utworzenie normalizera
10     auto scaler = wrap(new CRescaleFeatures);
11     // nauka normalizera oraz przeprowadzenie normalizacji
12     scaler->fit(inputs);
13     scaler->transform(inputs);
14 }
```

4.3.2. Redukcja wymiarowości

Shark udostępnia użytkownikowi kilka rodzajów algorytmów redukcji wymiarowości, realizowane przez następujące klasy [5]:

- **PCA** - klasa *CPCA*;
- **Kernel PCA** - klasa *CKernelPCA*;
- **MDS** - klasa *MultidimensionalScaling*;
- **IsoMap** - klasa *CIsoMap*;
- **ICA** - klasa *CFastICA*;
- **Factor analysis** - klasa *CFactorAnalysis*;
- **t-SNE** - klasa *CTDistributedStochasticNeighborEmbedding*.

Każda z powyższych klas operuje poprzez poprzednie nauczenie się parametrów danych uczących metodą *fit()* oraz ustawienie docelowej ilości wymiarów (z wyjątkiem ICA). Nauczony obiekt reduktora można wykorzystać do redukcji wymiarowości danych poprzez metodę *apply_to_feature_vector()* zwracającą przetworzony wektor, lub w przypadku ICA, Analizy Składowych oraz t-SNE metodę *transform()*, której wynik należy zrzutować na wskaźnik na CDenseFeatures. Niestety wykorzystanie któregokolwiek z reduktorów wiąże się z koniecznością utworzenia nowej kopii obiektu w procesie transformacji, zamiast wykonania przekształceń w miejscu. Listing 4.4 przedstawia sposób wykonania redukcji na przykładzie klasy *CKernelPCA*.

Listing 4.4. Przykład redukcji wymiarowości z wykorzystaniem metody Kernel PCA [5]

```
1  #pragma once
2
3  inline void KernelPCA(
4      shogun::Some<shogun::CDenseFeatures<float64_t>> inputs,
5      const int target_dim)
6  {
7      using namespace shogun;
8
9      // utworzenie jądra
10     auto gaussKernel = some<CGaussianKernel>(inputs, inputs, 0.5);
11     // utworzenie obiektu reduktora wymiarowości
12     auto pca = some<CKernelPCA>();
13     // konfiguracja reduktora
14     pca->set_kernel(gaussKernel.get());
15     pca->set_target_dim(target_dim);
16     // nauczenie reduktora
17     pca->fit(inputs);
18     // zastosowanie redukcji
19     auto featureMatrix = inputs->get_feature_matrix();
20     for (index_t i = 0; i < inputs->get_num_vectors(); ++i)
21     {
22         auto vector = featureMatrix.get_column(i);
23         auto newVector = pca->apply_to_feature_vector(vector);
24     }
25 }
```

4.3.3. Regularyzacja L1 i L2

W przypadku biblioteki Shogun, regularyzacja stanowi integralną część modelu, co oznacza że występuje ona zawsze podczas wykorzystania danego typu modelu uczenia maszynowego, oraz nie ma możliwości zmiany typu regularyzacji używanej przez docelowy model.

4.4. Modele uczenia maszynowego

4.4.1. Regresja liniowa

Jednym z podstawowych algorytmów uczenia maszynowego udostępnianych przez bibliotekę Shogun jest regresja liniowa, realizowana za pomocą klasy *CLinearRidgeRegression*. Polega ona na dekompozycji macierzy Choleskiego [5], wykorzystując podejście nie-iteracyjne. Jak wskazuje nazwa, metoda ta posiada wbudowaną regularyzację L2 (Ridge), której konfiguracja odbywa się podczas tworzenia obiektu modelu. Listing 4.5 przedstawia sposób wykonania regresji liniowej z pomocą Shogun.

Listing 4.5. Przykład regresji liniowej w Shogu

```
#pragma once

#include <iostream>

#include <inc/shogun/verify.hpp>

#include <shogun/base/some.h>
#include <shogun/features/DenseFeatures.h>
#include <shogun/labels/RegressionLabels.h>
#include <shogun/regression/LinearRidgeRegression.h>

inline void shogunLinear(
    shogun::Some<shogun::CDenseFeatures<float64_t>>& inputs,
    shogun::Some<shogun::CRegressionLabels>& outputs)
{
    using namespace shogun;

    // utworzenie modelu
    float64_t tauRegularization = 0.0001;
    auto linear =
        some<CLinearRidgeRegression>(tauRegularization, nullptr, nullptr);
    // podział danych na testowe i uczące
    auto testSamples = static_cast<int>(0.8*inputs.num_cols());
    auto trainInputs = inputs.submatrix(0, testSamples).clone();
    auto trainOutputs = outputs.submatrix(0, testSamples).clone();
    auto testInputs =
        inputs.submatrix(testSamples, inputs.num_cols()).clone();
    auto testOutputs =
        outputs.submatrix(testSamples, outputs.num_cols()).clone();
    // nauczanie
    linear->set_labels(trainOutputs);
    linear->train(trainInputs);
    // weryfikacja modelu
```

```
34      std::cout << "-----␣Shogun␣Linear␣-----" << std::endl;
35      std::cout << "Train␣data:␣" << std::endl;
36      auto predictions = wrap(linear->apply_regression(trainInputs));
37      shogunVerifyModel(predictions, trainOutputs, Task::REGRESSION);
38
39      std::cout << "Test␣data:␣" << std::endl;
40      predictions = wrap(linear->apply_regression(testInputs));
41      shogunVerifyModel(predictions, testOutputs, Task::REGRESSION);
42  }
```

4.4.2. Regresja logistyczna

Biblioteka Shogun zawiera implementację wieloklasowej regresji logistycznej w postaci gotowego obiektu klasy *CMulticlassLogisticRegression*. Posiada ona wbudowaną konfigurowalną regularyzację. Listing 4.6 przedstawia sposób użycia wspomnianej klasy.

Listing 4.6. Przykład regresji logistycznej

```
1   #pragma once
2
3   #include <inc/shogun/verify.hpp>
4
5   #include <iostream>
6   #include <shogun/base/some.h>
7   #include <shogun/features/DenseFeatures.h>
8   #include <shogun/labels/MulticlassLabels.h>
9   #include <shogun/multiclass/MulticlassLogisticRegression.h>
10
11
12  inline void shogunLogistic(
13      shogun::Some<shogun::CDenseFeatures>& inputs,
14      shogun::Some<shogun::CBinaryLabels>& outputs)
15  {
16      using namespace shogun;
17
18      // podział danych na testowe i uczące
19      auto testSamples = static_cast<int>(0.8*inputs.num_cols());
20      auto trainInputs = inputs.submatrix(0, testSamples).clone();
21      auto trainOutputs = outputs.submatrix(0, testSamples).clone();
22      auto testInputs =
23          inputs.submatrix(testSamples, inputs.num_cols()).clone();
24      auto testOutputs =
25          outputs.submatrix(testSamples, outputs.num_cols()).clone();
26
27      // utworzenie modelu
28      auto logReg = some<CMulticlassLogisticRegression>();
29
30      // nauka modelu
31      logReg->set_labels(trainOutputs);
32      logReg->train(trainInputs);
33
34      // ewaluacja modelu
35      std::cout << "-----␣Shogun␣Logistic␣-----" << std::endl;
36      std::cout << "Train:" << std::endl;
37      auto prediction = wrap(logReg->apply_multiclass(trainInputs));
```

```
38      shogunVerifyModel(prediction, trainOutputs, Task::CLASSIFICATION);
39
40      std::cout << "Test:" << std::endl;
41      prediction = wrap(logReg->apply_multiclass(testInputs));
42      shogunVerifyModel(prediction, testOutputs, Task::CLASSIFICATION);
43  }
```

4.4.3. Maszyna wektorów nośnych

Podobnie jak w przypadku regresji logistycznej, w bibliotece Shogun dostępna jest implementacja wieloklasowej klasyfikacji z wykorzystaniem maszyny wektorów nośnych, w postaci klasy *CMulticlassLibSVM*. Posiada ona szereg dostępnych do konfiguracji parametrów, i umożliwia wybór zastosowanego jądra użytkownikowi. Listing 4.7 prezentuje jak wykorzystać wymienioną klasę.

Listing 4.7. Przykład użycia maszyny wektorów nośnych

```
1   #pragma once
2
3   #include <inc/shogun/verify.hpp>
4
5   #include <iostream>
6   #include <shogun/base/some.h>
7   #include <shogun/features/DenseFeatures.h>
8   #include <shogun/labels/MulticlassLabels.h>
9   #include <shogun/kernel/GaussianKernel.h>
10  #include <shogun/multiclass/MulticlassLibSVM.h>
11
12  inline void shogunSVM(shogun::Some<shogun::CDenseFeatures>& inputs,
13                        shogun::Some<shogun::CBinaryLabels>& outputs)
14  {
15      using namespace shogun;
16
17      // podział danych na testowe i uczące
18      auto testSamples = static_cast<int>(0.8*inputs.num_cols());
19      auto trainInputs = inputs.submatrix(0, testSamples).clone();
20      auto trainOutputs = outputs.submatrix(0, testSamples).clone();
21      auto testInputs =
22          inputs.submatrix(testSamples, inputs.num_cols()).clone();
23      auto testOutputs =
24          outputs.submatrix(testSamples, outputs.num_cols()).clone();
25
26      // utworzenie jądra
27      auto kernel = some<CGaussianKernel>(trainInputs, trainInputs, 5);
28      // utworzenie i konfiguracja modelu
29      auto svm = some<CMulticlassLibSVM>();
30      svm->set_kernel(kernel);
31
32      // trening
33      svm->set_labels(trainOutputs);
34      svm->train(trainInputs);
35
36      // ewaluacja modelu
37      std::cout << "-----␣Shogun␣SVM␣-----" << std::endl;
38      std::cout << "Train:" << std::endl;
39      auto prediction = wrap(svm->apply_binary(trainInputs));
```

```
40      shogunVerifyModel(prediction, trainOutputs, Task::CLASSIFICATION);
41
42      std::cout << "Test:" << std::endl;
43      prediction = wrap(svm->apply_binary(testInputs));
44      shogunVerifyModel(prediction, testOutputs, Task::CLASSIFICATION);
45  }
```

4.4.4. Algorytm K najbliższych sąsiadów

Algorytm K najbliższych sąsiadów dostępny jest pod postacią klasy *CKNN*. Umożliwia on wybranie sposobu obliczania dystansu poprzez przekazanie obiektu odpowiedniej klasy, oraz ilości najbliższych sąsiadów. Głównymi z dostępnych typów dystansów są dystans Euklidesa, Hamminga, Manhatanu oraz podobieństwo kosinusowe. W porównaniu do poprzednich metod, nie wymaga on ustawiania hiperparametrów, dzięki czemu można z niego bezproblemowo korzystać bez sprawdzianu krzyżowego. Listing 4.8 pokazuje przykład konfiguracji i użycia algorytmu kNN z użyciem dystansu Euklidesa.

Listing 4.8. Przykład algorytmu kNN w Shogun

```
1   #pragma once
2
3   #include <inc/shogun/verify.hpp>
4
5   #include <iostream>
6   #include <shogun/base/some.h>
7   #include <shogun/features/DenseFeatures.h>
8   #include <shogun/labels/MulticlassLabels.h>
9   #include <shogun/multiclass/KNN.h>
10  #include <shogun/distance/EuclideanDistance.h>
11
12  inline void shogunKNN(shogun::Some<shogun::CDenseFeatures>& inputs,
13                        shogun::Some<shogun::CMulticlassLabels>& outputs)
14  {
15      using namespace shogun;
16
17      // podział danych na testowe i uczące
18      auto testSamples = static_cast<int>(0.8*inputs.num_cols());
19      auto trainInputs = inputs.submatrix(0, testSamples).clone();
20      auto trainOutputs = outputs.submatrix(0, testSamples).clone();
21      auto testInputs =
22          inputs.submatrix(testSamples, inputs.num_cols()).clone();
23      auto testOutputs =
24          outputs.submatrix(testSamples, outputs.num_cols()).clone();
25      // przygotowanie dystansu
26      auto distance = some<CEuclideanDistance>(trainInputs, trainInputs);
27      // przygotowanie modelu
28      std::int32_t k = 3;
29      auto knn = some<CKNN>(k, distance, trainOutputs);
30      // ewaluacja modelu
31      std::cout << "-----␣Shogun␣KNN␣-----" << std::endl;
32      std::cout << "Train:" << std::endl;
33      auto prediction = wrap(knn->apply_multiclass(trainInputs));
34      shogunVerifyModel(prediction, trainOutputs, Task::CLASSIFICATION);
35
```

```
36      std::cout << "Test:" << std::endl;
37      prediction = wrap(knn->apply_multiclass(testInputs));
38      shogunVerifyModel(prediction, testOutputs, Task::CLASSIFICATION);
39  }
```

4.4.5. Algorytm zbiorowy

4.4.5.1. Wzmacnianie gradientu

Implementacja algorytmu zbiorowego z wykorzystaniem metody wzmacniania gradientu przystosowana jest do działania jedynie z modelami wykonującymi zadanie regresji. Klasa odpowiedzialna za jego realizację to *CStochasticGBMachine*. Pozwala ona na konfigurację szeregu parametrów, do których należą:

- bazowy algorytm;
- funkcja straty;
- liczba iteracji;
- współczynnik uczenia;
- ułamek wektorów do losowego wybrania w każdej iteracji.

Listing 4.9 przedstawia sposób implementacji powyższej metody z wykorzystaniem binarnego drzewa decyzyjnego regresji i klasyfikacji (implementowanego przez klasę *CCARTree*) jako algorytm bazowy.

Listing 4.9. Przykład użycia metody wzmacniania gradientu

```
1  #pragma once
2
3  #include <inc/shogun/verify.hpp>
4
5  #include <iostream>
6  #include <shogun/base/some.h>
7  #include <shogun/features/DenseFeatures.h>
8  #include <shogun/labels/MulticlassLabels.h>
9  #include <shogun/loss/SquaredLoss.h>
10 #include <shogun/lib/SGMatrix.h>
11 #include <shogun/lib/SGVector.h>
12 #include <shogun/machine/StochasticGBMachine.h>
13 #include <shogun/multiclass/tree/CARTree.h>
14
15 inline void shogunGradientBoost(
16     shogun::Some<shogun::CDenseFeatures>& inputs,
17     shogun::Some<shogun::CMulticlassLabels>& outputs)
18 {
19     using namespace shogun;
20
21     // podział danych na testowe i uczące
22     auto testSamples = static_cast<int>(0.8*inputs.num_cols());
23     auto trainInputs = inputs.submatrix(0, testSamples).clone();
24     auto trainOutputs = outputs.submatrix(0, testSamples).clone();
```

```
25      auto testInputs =
26          inputs.submatrix(testSamples, inputs.num_cols()).clone();
27      auto testOutputs =
28          outputs.submatrix(testSamples, outputs.num_cols()).clone();
29      // oznaczenie regresorów jako ciągłe
30      SGVector<bool> featureType(1);
31      featureType.set_const(false);
32      // utworzenie binarnego drzewa decyzyjnego
33      auto tree = some<CCARTree>(featureType, PT_REGRESSION);
34      tree->set_max_depth(3);
35      // utworzenie funkcji straty
36      auto loss = some<CSquaredLoss>();
37      // utworzenie i konfiguracja modelu
38      constexpr int iterations = 100;
39      constexpr int learningRate = 0.1;
40      constexpr int subsetFraction = 1.0;
41      auto model = some<CStochasticGBMachine>(tree,
42                                              loss,
43                                              iterations,
44                                              learningRate,
45                                              subsetFraction);
46      // trening
47      model->set_labels(trainOutputs);
48      model->train(trainInputs);
49      // ewaluacja modelu
50      std::cout << "-----␣Shogun␣Gradient␣Boost␣-----" << std::endl;
51      std::cout << "Train:" << std::endl;
52      auto prediction = wrap(model->apply_multiclass(trainInputs));
53      shogunVerifyModel(prediction, trainOutputs, Task::CLASSIFICATION);
54
55      std::cout << "Test:" << std::endl;
56      prediction = wrap(model->apply_multiclass(testInputs));
57      shogunVerifyModel(prediction, testOutputs, Task::CLASSIFICATION);
58  }
```

4.4.5.2. Losowy las

Metoda losowego lasu dostępna jest w bibliotece Shogun poprzez użycie klasy *CRandomForest*. W przeciwieństwie do wzmacniania gradientu, implementacja tej metody pozwala także na wykonywanie klasyfikacji. Do głównych konfigurowalnych parametrów należą:

- ilość drzew;
- liczba zbiorów na które powinny zostać podzielone dane;
- algorytm wybrania końcowego wyniku;
- typ rozwiązywanego problemu;
- ciągłość wartości regresorów.

Listing 4.10 pokazuje jak utworzyć i skonfigurować model losowego lasu do wykonania zadania aproksymacji funkcji kosinus.

Listing 4.10. Przykład użycia metody losowego lasu

```
#pragma once

#include "inc/shogun/verify.hpp"

#include <shogun/base/some.h>
#include <shogun/ensemble/MajorityVote.h>
#include <shogun/labels/RegressionLabels.h>
#include <shogun/lib/SGMatrix.h>
#include <shogun/lib/SGVector.h>
#include <shogun/machine/RandomForest.h>

inline void shogunRandomForest(
    shogun::Some<shogun::CDenseFeatures<DataType>> inputs,
    shogun::Some<shogun::CRegressionLabels> outputs)
{
    using namespace shogun;

    // podział danych na testowe i uczące
    auto testSamples = static_cast<int>(0.8*inputs.num_cols());
    auto trainInputs = inputs.submatrix(0, testSamples).clone();
    auto trainOutputs = outputs.submatrix(0, testSamples).clone();
    auto testInputs =
        inputs.submatrix(testSamples, inputs.num_cols()).clone();
    auto testOutputs =
        outputs.submatrix(testSamples, outputs.num_cols()).clone();
    // utworzenie i konfiguracja modelu
    constexpr std::int32_t numRandFeats = 1;
    constexpr std::int32_t numBags = 10;
    auto randForest =
        some<CRandomForest>(numRandFeats, numBags);
    auto vote = some<CMajorityVote>();
    randForest->set_combination_rule(vote);
    // oznaczenie danych jako ciągłe
    SGVector<bool> featureType(1);
    featureType.set_const(false);
    randForest->set_feature_type(featureType);
    // trening
    randForest->set_labels(trainOutputs);
    randForest->set_machine_problem_type(PT_REGRESSION);
    randForest->train(trainInputs);
    // ewaluacja modelu
    std::cout << "-----␣Shogun␣Random␣Forest␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = wrap(randForest->apply_regression(trainInputs));
    shogunVerifyModel(predictions, trainOutputs, Task::REGRESSION);

    std::cout << "Test␣data:" << std::endl;
    predictions = wrap(randForest->apply_regression(testInputs));
    shogunVerifyModel(predictions, testOutputs, Task::REGRESSION);
}
```

4.4.6. Sieć neuronowa

Pierwszym krokiem tworzenia sieci neuronowej dla niniejszej biblioteki jest skonfigurowanie architektury sieci za pomocą obiektu klasy *CNeuralLayers*. Posiada ona

szereg metod, które tworzą odpowiednio skonfigurowane warstwy z wybraną funkcją aktywacji:

- *input()* - warstwa wejściowa z określoną ilością wymiarów;
- *logistic()* - warstwa w pełni połączona z sigmoidalną funkcją aktywacji;
- *linear()* - warstwa w pełni połączona z liniową funkcją aktywacji;
- *rectified_linear()* - warstwa w pełni połączona z funkcją aktywacji ReLU;
- *leaky_rectified_linear* - warstwa w pełni połączona z funkcją aktywacji Leaky ReLU;
- *softmax* - warstwa w pełni połączona z funkcją aktywacji softmax.

Kolejność wywoływania powyższych metod jest istotna, ponieważ decyduje ona o kolejności warstw w modelu. Po zakończeniu konfiguracji, możliwe jest utworzenie obiektu zatwierdzonej architektury za pomocą funkcji *done()*, a następnie wykorzystanie go do inicjalizacji klasy *CNeuralNetwork*. W celu połączenia warstw, należy wywołać na obiekcie sieci neuronowej funkcję *quick_connect* oraz zainicjalizować wagi metodą *initialize_neural_network*. Może ona przyjąć parametr określający rozkład Gaussa używany do inicjalizacji parametrów.

Następnym krokiem jest skonfigurowanie optymalizatora za pomocą metody *set_optimization*. Klasa *CNeuralNetwork* wspiera optymalizację z wykorzystaniem metody spadku gradientu oraz Broydena-Fletchera-Goldfarba-Shannona. Sieć neuronowa posiada wbudowaną regularyzację L2, którą można skonfigurować, podobnie jak pozostałe parametry takie jak współczynnik uczenia, ilość epok, kryterium zbieżności dla funkcji straty, czy wielkość zestawów *batch*. Niestety, niemożliwy jest wybór funkcji straty, gdyż jest on dokonywany automatycznie na podstawie typu zmiennej odpowiedzi. Listing 4.11 przedstawia pełny proces budowania, konfiguracji oraz uczenia sieci. Niestety ze względu na brak implementacji warstwy neuronu o aktywacji w postaci tangensa hiperbolicznego, na potrzeby prezentacji zastosowania zdecydowano się na wykorzystanie funkcji rectified linear, co ma wpływ na uzyskane wyniki.

Listing 4.11. Przykład użycia sieci neurnowej

```
1  #pragma once
2
3  #include "inc/shogun/verify.hpp"
4
5  #include <shogun/features/DenseFeatures.h>
6  #include <shogun/labels/MulticlassLabels.h>
7  #include <shogun/neuralnets/NeuralLayers.h>
8  #include <shogun/neuralnets/NeuralNetwork.h>
9  #include <shogun/base/some.h>
10
11 inline void sharkNeural(
12     shogun::Some<shogun::CDenseFeatures<float64_t>> inputs,
13     shogun::Some<shogun::CBinaryLabels> outputs)
14 {
15     using namespace shogun;
```

```
16
17      // podział danych na testowe i uczące
18      auto testSamples = static_cast<int>(0.8*inputs.num_cols());
19      auto trainInputs = inputs.submatrix(0, testSamples).clone();
20      auto trainOutputs = outputs.submatrix(0, testSamples).clone();
21      auto testInputs =
22          inputs.submatrix(testSamples, inputs.num_cols()).clone();
23      auto testOutputs =
24          outputs.submatrix(testSamples, outputs.num_cols()).clone();
25      // konstrukcja architektury sieci
26      auto dimensions = trainInputs.get_num_features();
27      auto layers = some<CNeuralLayers>();
28      layers = wrap(layers->input(dimensions));
29      layers = wrap(layers->rectified_linear(5));
30      layers = wrap(layers->rectified_linear(5));
31      layers = wrap(layers->logistic(1));
32      auto allLayers = layers->done();
33      // utworzenie sieci
34      auto network = some<CNeuralNetwork>(allLayers);
35      network->quick_connect();
36      network->initialize_neural_network();
37      // konfiguracja sieci
38      network->set_pptimization_method(NNOM_GRADIENT_DESCENT);
39      network->set_gd_mini_batch_size(64);
40      network->set_l2_coefficient(0.0001);
41      network->set_max_num_epochs(500);
42      network->set_epsilon(0.0);
43      network->set_gd_learning_rate(0.01);
44      network->set_gd_momentum(0.5);
45      // trening
46      network->set_labels(trainOutputs);
47      network->train(trainInputs);
48      // walidacja
49      std::cout << "-----␣Shogun␣Neural␣Network␣-----" << std::endl;
50      std::cout << "Train␣data:" << std::endl;
51      auto predictions = network->apply_binary(trainInputs);
52      shogunVerifyModel(predictions, trainOutputs, Task::CLASSIFICATION);
53
54      std::cout << "Test␣data:" << std::endl;
55      predictions = network->apply_binary(testInputs);
56      shogunVerifyModel(predictions, testOutputs, Task::CLASSIFICATION);
57  }
```

4.5. Metody analizy modeli

4.5.1. Błąd średniokwadratowy

Obliczenie błędu średniokwadratowego w bibliotece Shogun sprowadza się do utworzenia obiektu wykorzystującego typ *CMeanSquaredError* jako argument szablonu funkcji *some<>()*. Jest on zwracany pod postacią wskaźnika. W celu otrzymania wartości błędu dla posiadanych danych, należy wywołać z jego pomocą funkcję evaluate, do której przekazany zostaje zestaw predykcji oraz oczekiwanych wartości. Listing 4.12 ukazuje sposób użycia wspomnianego mechanizmu.

Listing 4.12. Przykład obliczenia wartości błędu średniokwadratowego [5]

```
using namespace shogun;

// [...]

auto mse_error = some<CMeanSquaredError>();
auto mse = mse_error->evaluate(predictions, train_labels);
```

4.5.2. Średni błąd absolutny

Realizacja obliczania średniego błędu absolutnego dla biblioteki Shogun dokonywana jest za pomocą klasy *CMeanAbsoluteError* pełniącej rolę ewaluatora. Tworzona jest ona poprzez wykorzystanie szablonu *some<>* a następnie wykorzystywana do obliczeń wołając jej metodę *evaluate* przekazując uzyskane oraz oczekiwane wyniki regresji lub klasyfikacji. Listing 4.13 przedstawia sposób użycia wyżej wymienionej klasy.

Listing 4.13. Przykład obliczenia wartości średniego błędu absolutnego [5]

```
using namespace shogun;
// [...]
auto mae_error = some<CMeanAbsoluteError>();
auto mae = mae_error->evaluate(predictions, train_labels);
```

4.5.3. Logarytmiczna funkcja straty

Logarytmiczna funkcja straty jest możliwa do obliczenia z wykorzystaniem biblioteki Shogun przy użyciu klasy *CLogLoss*, jednak udostępniane przez nią metody wskazują że powinna być wykorzystywana przez model, a nie bezpośrednio przez użytkownika. Udostępnia ona metodę *get_square_grad()* pozwalającą na obliczenie kwadratu gradientu między zadaną predykcją a docelowym wynikiem. Sposób użycia tej metody zaprezentowano na listingu 4.14

Listing 4.14. Przykład użycia klasy *CLogLoss*

```
using namespace shogun;
// [...]
auto logLoss = some<CLogLoss>();
auto sqareGradient = logLoss->get_square_grad(prediction, label);
```

4.5.4. Metryka R^2

Biblioteka Shogun nie posiada bezpośredniej implementacji dla metryki R^2 w związku z czym, pomimo możliwości wykorzystania wbudowanej metody obliczania błędu średniokwadratowego, wariancja potrzebna dla uzyskania wyniku musi zostać uzyskana przez własny mechanizm użytkownika. Listing 4.15 przedstawia funkcję weryfikującą poprawność modeli opisanych w poprzednich punktach, obliczającą wartość metryki R^2.

Listing 4.15. Przykład obliczenia metryki R^2

```
#pragma once

#include <iostream>
#include <cmath>

#include <shogun/evaluation/MeanSquaredError.h>
#include <shogun/evaluation/CMeanAbsoluteError.h>
#include <shogun/evaluation/ROCEvaluation.h>

enum class Task
{
    CLASSIFICATION,
    REGRESSION
};

inline auto shogunVerifyModel(auto predictions, auto targets, Task task)
{
    using namespace shogun;

    // błąd średniokwadratowy
    auto mseError = some<CMeanSquaredError>();
    auto mse = mseError->evaluate(predictions, targets);
    std::cout << "MSE␣=␣" << mse << std::endl;
    // metryka R^2
    float64_t avg = 0.0;
    float64_t sum = 0.0;
    // obliczenie średniej i wariancji
    for (index_t i = 0; i < targets.num_labels(); i++)
    {
        avg += (targets.get_label(i));
    }
    avg /= targets.num_labels();
    for (index_t i = 0; i < targets.num_labels(); i++)
    {
        sum += std::pow(targets.get_label(i) - avg, 2);
    }
    auto variance = (1.0/targets.num_labels()) * sum;
    // obliczenie metryki R^2
    auto r_square = 1.0 - mse / variance;
    std::cout << "R^2␣=␣" << r_square << std::endl << std::endl;
    if (task == CLASSIFICATION)
    {
        auto roc = some<CROCEvaluation>();
        roc->evaluate(predictions, targets);
        std::cout << "AUC␣ROC␣=␣" << roc->get_auROC() << std::endl;
    }
}
```

4.5.5. Dokładność

Do obliczenia dokładności w przypadku zadań regresji, biblioteka udostępnia klasę *CMulticlassAccuracy*. Pozwala ona nie tylko na samą klasyfikację, lecz także oferuje metodę pobrania macierzy misklasyfikacji. Nie znaleziono natimiast klasy *CAccuracyMeasure* wspomnianej przez autora „Hands-On Machine Learning with C++”[5],

co sugerowałoby jej usunięcie z biblioteki. Listing 4.16 pokazuje w jaki sposób należy użyć klasy *CMulticlassAccuracy*.

Listing 4.16. Przykład obliczenia dokładności modelu

```
using namespace shogun;
// [...]
auto acc_measure = some<CMulticlassAccuracy>();
auto acc = acc_measure->evaluate(predictions, train_labels);
auto confusionMatrix = acc_measure->get_confusion_matrix(
    predictions, train_labels);
```

4.5.6. Precyzja i pamięć (recall), oraz metryka F-score

W książce [5] wspomniane zostały klasy *CRecallMeasure* oraz *CF1Measure* mające pozwolić obliczyć odpowiednio pamięć modelu oraz metrykę F-score, jednak w trakcie pracy z biblioteką nie znaleziono definicji tych klas, lub jakichkolwiek innych pełniących te funkcje, w związku z czym założono brak implementacji tych metod dla biblioteki Shogun. Zdecydowano się na wspomnienie o tym fakcie, w celu podkreślenia możliwoscy wystąpienia rozbieżności między źródłami wiedzy, a aktualnym stanem biblioteki.

4.5.7. Pole pod wykresem krzywej operacyjnej

Biblioteka Shogun posiada implementację obliczania pola pod wykresem krzywej charakterystycznej odbiornika, w postaci klasy *CROCEvaluation*. Listing 4.17 przedstawia sposób jej użycia.

Listing 4.17. Przykład obliczenia pola pod wykresem funkcji ROC dla Shogun

```
using namespace shogun;
// [...]
auto roc = some<CROCEvaluation>();
roc->evaluate(predictions, targets);
std::cout << "AUC␣ROC␣=␣" << roc->get_auROC() << std::endl;
```

4.5.8. Sprawdzian krzyżowy K-krotny

Sprawdzian krzyżowy stanowi w bibliotece Shogun złożony mechanizm, do którego wykorzystania należy przygotować drzewo decyzyjne parametrów, reprezentowane przez klasę *CModelSelectionParameters*. Użytkownik może wybrać model oraz kryterium ewaluacji modelu poprzez utworzenie odpowiednich klas, a następnie przekazanie ich w konstruktorze obiektu sprawdzianu krzyżowego, będącego instancją klasy *CCrossValidation*. Kolejnym krokiem jest utworzenie instancji klasy *CGridSearchModelSelection* która dokona wyboru parametrów. Ostatnim etapem jest konfiguracja docelowego modelu i przeprowadzenie procesu uczenia. Dokłady wygląd całego mechanizmu został przedstawiony na listingu 4.18.

Listing 4.18. Przygotowanie modelu wieloklasowej regresji liniowej z wykorzystaniem sprawdzianu krzyżowego

```
#pragma once

inline void shogunCrossValidLogistic(
    shogun::Some<shogun::CDenseFeatures<float64_t>> inputs,
    shogun::Some<shogun::CMulticlassLabels> outputs)
{
    using namespace shogun;

    // podział danych na testowe i uczące
    auto testSamples = static_cast<int>(0.8*inputs.num_cols());
    auto trainInputs = inputs.submatrix(0, testSamples).clone();
    auto trainOutputs = outputs.submatrix(0, testSamples).clone();
    auto testInputs =
        inputs.submatrix(testSamples, inputs.num_cols()).clone();
    auto testOutputs =
        outputs.submatrix(testSamples, outputs.num_cols()).clone();
    // utworzenie drzewa parametrów
    auto root = some<CModelSelectionParameters>();
    // współczynnik regularyzacji
    CModelSelectionParameters* z = new CModelSelectionParameters("m_z");
    root->append_child(z);
    z->build_values(0.2, 1.0, R_LINEAR, 0.1);
    // utworzenie strategii podziału drzewa decyzyjnego
    index_t k = 3;
    CStatifiedCrossValidationSplitting* splitting =
        new CStatifiedCrossValidationSplitting(labels, k);
    // utworzenie kryterium ewaluacji dla drzewa decyzyjnego
    auto evalCriterium = some<CMulticlassAccuracy>();
    // utworzenie modelu regresji logistycznej
    auto logReg = some<CMulticlassLogisticRegression>();
    // utworzenie obiektu sprawdzianu krzyżowego
    auto cross = some<CCrossValidation>(logReg, trainInputs, trainOutputs,
                                        splitting, evalCriterium);
    cross->set_num_runs(1);
    auto modelSelection = some<CGridSearchModelSelection>(cross, root);
    // wybranie parametrów dla modelu
    CParameterCombination* bestParams =
        wrap(modelSelection->select_model(false));
    // zaaplikowanie parametrów dla modelu
    bestParams->apply_to_machine(logReg);
    // wyświetlenie drzewa decyzyjnego
    bestParams->print_tree();
    // trenowanie docelowego modelu
    logReg->set_labels(trainOutputs);
    logReg->train(trainInputs);

    // ewaluacja modelu
    std::cout << "-----␣Shogun␣CV␣Logistic␣-----" << std::endl;
    std::cout << "Train:" << std::endl;
    auto prediction = wrap(logReg->apply_multiclass(trainInputs));
    shogunVerifyModel(prediction, trainOutputs, Task::CLASSIFICATION);

    std::cout << "Test:" << std::endl;
    prediction = wrap(logReg->apply_multiclass(testInputs));
    shogunVerifyModel(prediction, testOutputs, Task::CLASSIFICATION);

```

```
57      delete splitting;
58  }
```

4.6. Dostępność dokumentacji i źródeł wiedzy

Internetowe źródła informacji w postaci forów społecznościowych skupiają się na wykorzystaniu biblioteki Shark w innych językach, jak np. Python, lecz wraz z jej kodem źródłowym na platformie GitHub [4] możliwe jest wygenerowanie przykładów jej wykorzystania także w języku C++ w folderze examples. Przykłady te należy zbudować za pomocą odpowiednego skryptu Pythona zawartego w repozytorium, powodując wygenerowanie listingów kodów w docelowym języku w plikach JSON. Niestety, okazują się one obrazować użycie biblioteki w nienaturalny, proceduralnie generowany sposób, sprawiając, że przy faktycznej próbie skorzystania z API projektu, stają się one bezużyteczne. Dodatkowo, jedyna forma dokumentacji projektu ogranicza się do komentarzy w kodzie, zmuszając użytkownika do błądzenia po repozytorum w poszukiwaniu potrzebnych informacji.

Shogun jest jedną z bibliotek opisaną w książce „*Hands On Machine Learning with C++*" autorstwa Kirilla Kolodiazhnyi [5], wprowadzającej czytelnika zarówno do podstawowych funkcjonalności Shogun, jak i podsumowującej podstawy teorii uczenia maszynowego w kontekście ich zastosowania. Większość z przykładów realizacji poszczególnych typów modeli w tej książce posiada przedstawione główne fragmenty listingów dla biblioteki Shogun. Warto jednak zaznaczyć, że różnią się one od przykładów generowanych przez skrypt budujący, obrazując wykorzystanie faktycznie udostępnianego API biblioteki. W toku pracy nad niniejszym rozdziałem, książka ta okazała się jedynym wartościowym źródłem wiedzy jego temat.

Rozdział 5

Biblioteka Shark-ML

5.1. Wprowadzenie

Shark-ML to biblioteka uczenia maszynowego dedykowana dla języka C++. Posiada ono otwarte źródło, i udostępniana jest na podstawie licencji *GNU Lesser General Public License*. Głównymi aspektami na których skupia się ta biblioteka są problemy liniowej i nieliniowej optymalizacji (w związku z czym posiada ona część funkcjonalności biblioteki do algebry liniowej), maszyny jądra (np. maszyna wektorów nośnych) i sieci neuronowe. [6] Podmiotami udostępniającymi bibliotekę jest Uniwersytet Kopenhagi w Danii, oraz Instytut Neuroinformatyki z Ruhr-Universitat Bochum w Niemczech.

5.2. Formaty źródeł danych

Biblioteka operuje na własnych reprezentacjach macierzy i wektorów, które tworzone są poprzez opakowywanie surowych tablic za pomocą specjalnych adapterów, jak np. *remora::dense_matrix_adaptor<>()* lub za pomocą kontenerów biblioteki standardowej C++ i funkcji *createDataFromRange()*. Mechanizm ten jest identyczny jak w przypadku pozostałych z omawianych bibliotek, co daje użytkownikowi dużą dowolność co do sposobu przechowywania danych i mechanizmu ich odczytywania. Posiada ona także dedykowany parser dla plików w formacie CSV, lecz zakłada on obecność w pliku jedynie danych numerycznych. Do jego użycia należy użyć klasy kontenera *ClassificationDataset* lub *RegressionDataset* oraz metody *importCSV* która zapisuje odczytane dane do wcześniej wspomnianego obiektu poprzez mechanizm zwracania przez parametr. Jeden z argumentów funkcji określa która z kolumn zawiera zmienną decyzyjną, dzięki czemu biblioteka jest w stanie od razu oddzielić dane wejściowe od kolumny oczekiwanych wartości. Artykuł „Classification with Shark-ML machine learning library”[7] dostępny na platformie GitHub pokazuje także, jak pobrać dane w postaci formatu CSV z internetu z pomocą API biblioteki *curl*, i przetworzyć je do formy akceptowanej przez Shark-ML. W aktualnej wersji biblioteki znalazły się także wbudowane funkcje pobierania danych współpracujące z protokołem HTTP. Listing 5.1 ukazuje jak odczytać dane z pliku .csv znajdującego się na dysku użytkownika.

Listing 5.1. Odczytanie danych z pliku CSV

```
#pragma once

#include <inc/shark/printEvaluation.hpp>

inline auto sharkReadCsvData(std::string filePath, constexpr Task task)
{
    using namespace shark;

    // odczytaj zawartość pliku
    std::ifstream file(filePath);
    std::string trainDataString(std::istreambuf_iterator<char>(file),
                                std::istreambuf_iterator<char>());

    if constexpr (task == Task::CLASSIFICATION)
    {
        ClassificationDataset trainData;
        csvStringToData(trainData, trainDataString, FIRST_COLUMN);
        return trainData;
    }
    else
    {
        RegressionDataset trainData;
        csvStringToData(trainData, trainDataString, LAST_COLUMN);
        return trainData;
    }
}
```

W celu opakowania danych zawartych w kontenerach biblioteki standardowej języka C++ do obiektów akceptowanych przez bibliotekę Shark-ML, konieczne jest wykorzystanie specjalnych funkcji adaptorowych, do których przekazywany jest wskaźnik na dane w postaci surowej tablicy, wraz z oczekiwanymi wymiarami wynikowej macierzy / wektora. Sposób opakowania danych pokazano na listingu 5.2

Listing 5.2. Sposób opakowywania danych do przetwarzania przez Shark-ML [5]

```
// przykładowe dane zawarte w kontenerze std::vector biblioteki
// standardowej C++
std::vector<float> data{1, 2, 3, 4};

// opakowanie danych do postaci macierzy 2 x 2
auto m = remora::dense_matrix_adaptor<float>(data.data(), 2, 2);

// opakowanie danych do postaci wektora 1 x 4
auto v = remora::dense_vector_adaptor<float>(data.data(), 4);
```

5.3. Metody przetwarzania i eksploracji danych

5.3.1. Normalizacja

Biblioteka Shark-ML implementuje normalizacje jako klasy treningowe dla modelu *Normalizer*, udostępniając użytkownikowi trzy możliwe do wykorzystania klasy:

- *NormalizeComponentsUnitInterval* - przetwarza dane tak aby mieściły się w przedziale jednostkowym;
- *NormalizeComponentsUnitVariance* - przelicza dane aby uzyskać jednostkową wariancję, i niekiedy także średnią wynoszącą 0.
- *NormalizeComponentsWhitening* - dane przetwarzane są w sposób zapewniający średnią wartość wynoszącą zero oraz określoną przez użytkownika wariancję (domyślnie wariancja jednostkowa).

Opierają się one o użycie metody *train()* na obiekcie normalizera, aby odpowiednio go skonfigurować do przetwarzania zarówno danych testowych, jak i wszystkich innych danych które użytkownik ma zamiar wprowadzić do modelu. Dodatkowymi funkcjami jest możliwość przemieszania danych, i wydzielenia fragmentu jako dane testowe za pomocą metody *shuffle()* klasy *ClassificationDataset* oraz funkcji *splitAtElement()*. Listing **??** pokazuje przykład wstępnego przetwarzania danych z wykorzystaniem normalizacji.

Listing 5.3. Wstępne przetwarzanie danych do uczenia [7]

```
#pragma once

inline auto sharkPreprocess(auto& trainData)
{
    using namespace shark;

    // przemieszanie danych
    trainData.shuffle();
    // utworzenie normalizera
    using Trainer = NormalizeComponentsUnitVariance<RealVector>;
    bool removeMean = true;
    Normalizer<RealVector> normalizer;
    Trainer normalizingTrainer(removeMean);
    // nauczenie normalizera średniej i wariancji danych treningowych
    normalizingTrainer.train(normalizer, trainData.inputs());
    // transformacja danych uczących
    return transformInputs(trainData, normalizer);
}
```

5.3.2. Redukcja wymiarowości

5.3.2.1. PCA

Algorytm redukcji wymiarowości PCA implementowany jest w bibliotece Shark za pośrednictwem klasy *PCA*. Wykorzystuje ona obiekt modelu liniowego w formie enkodera oraz przyjmuje oprócz niego w metodzie *encoder* docelowy wymiar zestawu danych. Wynikiem działania wymienionej metody jest konfiguracja modelu liniowego do tworzenia zestawu danych o zredukowanym wymiarze. Listing 5.4 przedstawia sposób wykorzystania klasy PCA.

Listing 5.4. Redukcja wymiarowości danych z wykorzystaniem klasy PCA i enkodera

```
// [...]

// utworzenie trenera PCA
shark::PCA pca(data);
shark::LinearModel<> enc;

// konfiguracja enkodera do redukcji wymiarów
constexpr int nbOfDim = 2;
pca.encoder(enc, nbOfDim);
auto encoded_data = enc(data);
```

5.3.2.2. Liniowa analiza dyskryminacyjna

Liniowa analizy dyskryminacyjnej (ang. *Linear Discriminant Analysis, LDA*) w przypadku biblioteki Shark-ML opiera się o rozwiązanie analityczne, poprzez konfigurację klasy modelu *LinearClassifier* przez klasę treningową *LDA*, wykorzystując funkcję *train()*. Możliwe jest także wykorzystanie LDA do zadania klasyfikacji, uzyskując predykcje dla zestawu danych za pomocą wywołania obiektu liniowego klasyfikatora jak funkcji (użycie operatora ()) przekazując mu dane uzyskane z ClassificationDataset za pomocą metody *inputs()*. Szczegóły implementacyjne dla redukcji wymiarowości danych zamieszczone zostały na listingu 5.5, natomiast listing **??** przedstawia sposób użycia LDA do zadania klasyfikacji.

Listing 5.5. Przykład redukcji zestawu danych z wykorzystaniem modelu LDA [5]

```
using namespace shark;

void LDAReduction(const UnlabeledData<RealVector>& data,
                  const UnlabeledData<RealVector>& labels,
                  std::size_t target_dim)
{
    // utworzenie obiektów LDA
    LinearClassifier<> encoder;
    LDA lda;
    // utworzenie zestawu danych
    LabeledData<RealVector, unsigned int> dataset(
        labels.numberOfElements(), InputLabelPair<RealVector, unsigned int>(
            RealVector(data.element(0).size()), 0));
    // wypełnienie zbioru danymi
    for (std::size_t i = 0; i < labels.numberOfElements(); ++i)
    {
        // zmiana indeksów klas aby zaczynały się od 0
        dataset.element(i).label =
            static_cast<unsigned int>(labels.element(i)[0]) - 1;
        dataset.element[i].input = data.element(i)
    }
    // trening enkodera
    lda.train(encoder, dataset);
    // utworzenie zredukowanego zestawu danych
    auto new_labels = encoder(data);
    auto new_data = encoder.decisionFunction()(data);
}
```

5.3.3. Regularyzacja L1

Biblioteka Shark, w przeciwieństwie do Shogun nie posiada ściśle określonych mechanizmów regularyzacji dla danych metod uczenia maszynowego. Zamiast tego, istnieje możliwość umieszczenia obiektu wykonującego regularyzację w obiekcie klasy trenera, za pomocą metody *setRegularization()*. W celu zastosowania metody Lasso, należy umieścić w wybranym trenerze obiekt klasy *shark::OneNormRegularizer*, a następnie przeprowadzić proces uczenia.

5.3.4. Regularyzacja L2

Podobnie jak w przypadku metody Lasso, wykorzystanie regularyzacji L2 w trenowanym modelu opiera się na wstrzyknięciu obiektu regularyzatora do obiektu klasy trenera. Dla metody L2 jest to obiekt klasy *shark::TwoNormRegularizer*.

5.4. Modele uczenia maszynowego

5.4.1. Regresja liniowa

Jednym z podstawowych modeli oferowanych przez niniejszą bibliotekę jest regresja liniowa. Do celów jej reprezentacji dostępna jest klasa *LinearModel*, oferująca rozwiązanie problemu w sposób analityczny za pomocą klasy trenera *LinearRegression*, lub podejście iteracyjne implementowane przez klasę trenera *LinearSAGTrainer*, wykorzystujące iteracyjną metodę gradientu średniej statystycznej (ang. *Statistic Averagte Gradient, SAG*). W przypadku bardziej skompilowanych regresji, gdzie może nie istnieć rozwiązanie analityczne, istnieje możliwość zastosowania podejścia iteracyjnego z użyciem optymalizatora wybranego przez użytkownika. Metoda ta sprowadza się to uczenia optymalizatora z wykorzystaniem funkcji straty, a następnie załadowanie uzyskanych wag do modelu regresji. Parametry modelu możliwe są do odczytania z wykorzystaniem metod *offset()* i *matrix()* lub metody *parameterVector()*. Na listingu 5.6 ukazane zostało wykorzystanie podejścia iteracyjnego, natomiast listing 5.7 przedstawia metodę analityczną.

Listing 5.6. Przykład regresji liniowej z wykorzystaniem optymalizatora spadku gradientowego

```
1  #pragma once
2
3  #include <iostream>
4
5  #include <inc/shark/printEvaluation.hpp>
6
7  inline void sharkLinear(const shark::RegressionDataset& trainData,
8                          const shark::RegressionDataset& testData)
9  {
10     using namespace shark;
11
12     // przygotowanie modelu
13     LinearModel<> model(inputDimension(trainData), labelDimension(trainData));
14     SquaredLoss<> loss;
15     ErrorFunction errorFunction(trainData, &model, &loss);
```

```
16      // przygotowanie i wyszkolenie optymalizatora
17      CG optimizer;
18      errorFunction.init();
19      optimizer.init(errorFunction);
20      for (int i = 0; i < 100; ++i)
21      {
22          optimizer.step(errorFunction);
23      }
24      // zastosowanie wytrenowanych parametrów modelu
25      model.setParameterVector(optimizer.solution().point);
26      // ewaluacja
27      std::cout << "-----␣Shark␣Linear␣-----" << std::endl;
28      std::cout << "Train␣data:" << std::endl;
29      auto predictions = model(trainData.inputs());
30      printSharkModelEvaluation(
31          trainData.outputs(), predictions, Task::REGRESSION);
32
33      std::cout << "Test␣data:" << std::endl;
34      predictions = model(testData.inputs());
35      printSharkModelEvaluation(
36          trainData.outputs(), predictions, Task::REGRESSION);
37  }
```

Listing 5.7. Przykład regresji liniowej z wykorzystaniem trenera analitycznego [5]

```
1   #pragma once
2
3   inline void sharkLinear(const shark::RegressionDataset& trainData,
4                           const shark::RegressionDataset& testData)
5   {
6       using namespace shark;
7
8       // utworzenie modelu
9       LinearRegression trainer;
10      LinearModel<> model;
11      // trening modelu
12      trainer.train(model, trainData);
13      // odczytanie parametrów modelu
14      std::cout << "intercept:␣" << model.offset() << std::endl;
15      std::cout << "matrix:␣" << model.matrix() << std::endl;
16      // ewaluacja
17      std::cout << "-----␣Shark␣Linear␣-----" << std::endl;
18      std::cout << "Train␣data:" << std::endl;
19      auto predictions = model(trainData.inputs());
20      printSharkModelEvaluation(
21          trainData.outputs(), predictions, Task::REGRESSION);
22
23      std::cout << "Test␣data:" << std::endl;
24      predictions = model(testData.inputs());
25      printSharkModelEvaluation(
26          trainData.outputs(), predictions, Task::REGRESSION);
27  }
```

5.4.2. Regresja logistyczna

Mechanizm regresji logistycznej dostępny w bibliotece Shark-ML z natury rozwiązuje problem regresji binarnej. Istnieje jednak możliwość przygotowania wielu klasyfikatorów, w ilości wyrażonej wzorem:

$$\frac{N(N-1)}{2} \tag{5.1}$$

gdzie N oznacza ilość klas występujących w problemie. Utworzone klasyfikatory następnie są złączane w jeden za pomocą odpowiedniej konfiguracji obiektu *OneVersusOneClassifier*, rozwiązując problem klasyfikacji wieloklasowej. W tym celu zestaw danych należy iteracyjnie podzielić na podproblemy o charakterystyce binarnej za pomocą wbudowanej funkcji *binarySubProblem()* przyjmującej zestaw danych i klasy. Nauczanie poszczególnych modeli realizowane jest poprzez klasę trenera *LogisticRegression*. Po zakończeniu trenowania okreslonej partii pomniejszych modeli, są one ładowane do głównego modelu. Wykorzystanie gotowego klasyfikatora wieloklasowego nie różni się od sposobu użycia modelu uzyskanego np. w klasyfikacji liniowej. Listing 5.8 prezentuje funkcję budującą model logistycznej regresji wieloklasowej, natomiast listing 5.9 pokazuje sposób utworzenia prostego modelu dla problemu binarnego.

Listing 5.8. Przykład funkcji tworzącej model wieloklasowej regresji logistycznej [5]

```
1  using namespace shark;
2
3  // [...]
4
5  void LRClassification(const ClassificationDataset& train,
6                        const ClassificationDataset& test,
7                        unsigned int num_classes)
8  {
9      // utworzenie obiektu docelowego klasyfikatora oraz tablicy
10     // klasyfikatorów składowych
11     OneVersusOneClassifier<RealVector> ovo;
12     auto pairs = num_classes * (num_classes - 1) / 2;
13     std::vector<LinearClassifier<RealVector>> lr(pairs);
14
15     // iteracyjne konfigurowanie klasyfikatorów składowych
16     for (std::size_t n = 0, cls1=1; cls1 < num_classes; ++cls1)
17     {
18         using BinaryClassifierType =
19             OneVersusOneClassifier<RealVector>::binary_classifier_type;
20         std::vector<BinaryClassifierType*> ovo_classifiers;
21         for (std::size_t cls2 = 0; cls2 < cls1; ++cls2, ++n)
22         {
23             // pobranie binarnego podproblemu
24             ClassificationDataset binary_cls_data =
25                 binarySubProblem(train, cls2, cls1);
26
27             // trening modelu składowego
28             LogisticRegression<RealVector> trainer;
29             trainer.train(lr[n], binary_cls_data);
30
31             // załadowanie modelu składowego do serii
```

```
32              ovo_classifiers.push_back(&lr[n]);
33          }
34          // podłączenie serii do głównego klasyfikatora
35          ovo.addClass(ovo_classifiers);
36      }
37
38      // użycie modelu
39      auto predictions = ovo(test.inputs());
40      // [...]
41  }
```

Listing 5.9. Przykład prostej binarnej regresji logistycznej

```
1   #pragma once
2
3   #include <iostream>
4
5   inline void sharkLogistic(const shark::ClassificationDataset& trainData,
6                             const shark::ClassificationDataset& testData)
7   {
8       using namespace shark;
9
10      // utworzenie modelu
11      LinearClassifier<RealVector> logisticModel;
12      LogisticRegression<RealVector> trainer;
13      // trening
14      trainer.train(logisticModel, trainData);
15      // ewaluacja
16      std::cout << "-----Shark␣Logistic␣Regression-----" << std::endl;
17      std::cout << "Train␣data␣model␣evaluation:" << std::endl;
18      auto predictions = logisticModel(trainData.inputs());
19      printSharkModelEvaluation(
20          trainData.labels(), predictions, Task::CLASSIFICATION);
21
22      std::cout << "Test␣data␣model␣evaluation:" << std::endl;
23      predictions = logisticModel(testData.inputs());
24      printSharkModelEvaluation(
25          testData.labels(), predictions, Task::CLASSIFICATION);
26  }
```

5.4.3. Maszyna wektorów nośnych

Jednym z bardzo istotnych z perspektywy zastosowania biblioteki Shark-ML oferowanych przez nią metod uczenia maszynowego jest maszyna wektorów nośnych stanowiąca rodzaj tzw. modeli jądra (ang. *kernel model*). Opiera się ona na wykonaniu regresji liniowej w przestrzeni cech określonych przez wykorzystany kernel. Podobnie jak w przypadku regresji logistycznej, API biblioteki umożliwia wykonanie klasyfikacji dla przypadku binarnego, natomiast rozwiązanie przy jej użyciu problemu wieloklasowego wymaga kombinacji instancji maszyn wektorów nośnych w model złożony, czego można dokonać przy pomocy klasy *OneVersusOneClassifier* oraz ilości klas wyrażonej wzorem 5.1. Zgodnie z charakterystyczną cechą tej biblioteki, użycie metody podzielone jest na utworzenie instancji modelu oraz obiektu klasy trenera, która go konfiguruje w procesie uczenia. W tym celu dostępne są dla użytkownika klasy:

- *GaussianRbfKernel* - odpowiada za obliczenie podobieństwa między zadanymi cechami wykorzystując funkcję bazową *ang. Radial Basis Function, RBF*;
- *KernelClassifier* - funkcja realizująca regresję liniową wewnątrz przestrzeni określonej przez jądro;
- *CSvmTrainer* - klasa trenera realizująca uczenie w oparciu o skonfigurowane parametry;

Do parametrów pozwalających na konfigurację modelu należą m.in.:

- przepustowość modelu - podawana w konstruktorze *GaussianRbfKernel* jako liczba z przedziału $\langle 0; 1\rangle$;
- regularyzacja - podawana jako liczba rzeczywista w konstruktorze *CSvmTrainer*, domyślnie maszyna wektorów nośnych używa kary typu *1-norm penalty* za przekroczenie docelowej granicy;
- bias - flaga binarna (bool) określająca czy model ma używać biasu, podawana w konstruktorze *CSvmTrainer*;
- *sparsify* - parametr określający czy model ma zachować wektory które nie są nośne, dostępny przez metodę *sparsify()* trenera;
- minimalna dokładność zakończenia nauczania - pozwala wyspecyfikować precyzję modelu, jest dostępna jako pole struktury zwracane przez metodę *stoppingCondition()* klasy trenera;
- wielkość cache - ustawiana za pomocą funkcji *setCacheSize()* trenera;

Sposób użycia modelu jest identyczny jak w przypadku pozostałych modeli, poprzez operator wywołania funkcji - (). Listing 5.10 ukazuje przykład utworzenia i skonfigurowania modelu na podstawie wpisów dostępnych w dokumentacji biblioteki, natomiast listing 5.10 przedstawia sposób utworzenia maszyny wektorów nośnych dla problemów wieloklasowych wewnątrz funkcji przyjmującej zestawy danych uczących i testowych.

Listing 5.10. Przykład maszyny wektorów nośnych dla problemu binarnego

```
1  #pragma once
2
3  inline void sharkSVM(const shark::ClassificationDataset& trainData,
4                       const shark::ClassificationDataset& testData)
5  {
6      using namespace shark;
7
8      // utworzenie jądra
9      double gamma = 0.5;
10     GaussianRbfKernel<> kernel(gamma);
11     KernelClassifier<RealVector> svm;
12     double regularization = 1000.0;
13     bool bias = true;
14     // utworzenie i konfiguracja modelu
15     CSvmTrainer<RealVector, double> trainer(
```

```
16          &kernel, regularization, bias);
17      trainer.sparsify() = false;
18      trainer.stopCondition().minAccuracy=1e-6;
19      trainer.setCacheSize(0x1000000);
20      // trening
21      trainer.train(svm, trainData);
22      // ewaluacja
23      std::cout << "-----Shark␣SVM-----" << std::endl;
24      std::cout << "Train␣data:" << std::endl;
25      auto predictions = svm(trainData.inputs());
26      printSharkModelEvaluation(
27          trainData.labels(), predictions, Task::CLASSIFICATION);
28
29      std::cout << "Test␣data:" << std::endl;
30      predictions = svm(testData.inputs());
31      printSharkModelEvaluation(
32          testData.labels(), predictions, Task::CLASSIFICATION);
33  }
```

Listing 5.11. Przykład maszyny wektorów nośnych dla problemu wieloklasowego [5]

```
1   using namespace shark;
2
3   void SVMClassification(const ClassificationDataset& train,
4                          const ClassificationDataset& test,
5                          unsigned int num_classes)
6   {
7       double gamma = 0.5;
8       GaussianRbfKernel<> kernel(gamma);
9       // utworzenie obiektu modelu docelowego
10      OneVersusOneClassifier<RealVector> ovo;
11      // utworzenie kontenera na poszczególne podproblemy
12      unsigned int pairs = num_classes * (num_classes - 1) / 2;
13      std::vector<KernelClassifier<RealVector>> svm(pairs);
14      for (std::size_t n = 0, cls1 = 1; cls1 < num_classes; cls1++)
15      {
16          // utworzenie zestawu klasyfikatorów podproblemów dla danej klasy
17          using BinaryClassifierType =
18              OneVersusOneClassifier<RealVector>::binary_classifier_type;
19          std::vector<BinaryClassifierType*> ovo_classifiers;
20          for (std::size_t cls2 = 0; cls2 < cls1; cls2++, n++)
21          {
22              // utworzenie podproblemu binarnego
23              ClassificationDataset binary_cls_data =
24                  binarySubProblem(train, cls2, cls1);
25              // trenowanie modelu składowego
26              double c = 10.0;
27              CSvmTrainer<RealVector> trainer(&kernel, c, false);
28              trainer.train(svm[n], binary_cls_data);
29              ovo_classifiers.push_back(&svm[n]);
30          }
31          // dołożenie zestawu klasyfikatorów do głównego modelu
32          ovo.addClass(ovo_classifiers);
33      }
34      // użycie modelu
35      auto predictions = ovo(test.inputs());
36  }
```

5.4.4. Algorytm K najbliższych sąsiadów

Jedną z metod klasyfikacji oferowanych przez bibliotekę Shark-ML jest model najbliższych sąsiadów, który można wyposażyć w różne algorytmy, w tym w algorytm kNN (ang. *K Nearest Neighbours*). Do reprezentacji modelu stworzona została klasa *NearestNeighborModel*. Biblioteka umożliwia wykorzystanie rozwiązania naiwnego (ang. *brute-force*) lub bazującego na podejściu drzew dzielnych (ang. *space partitioning tree*) poprzez użycie klas *KDTree* i *TreeNearestNeighbors*. W przeciwieństwie do poprzednio wskazanych metod, wykonanie klasyfikacji wieloklasowej w tym przypadku nie wymaga tworzenia złożonych modeli lub podawania modelowi ilości klas. Jest on automatycznie konfigurowany na podstawie danych uczących. Listing 5.12 przedstawia sposób przygotowania klasyfikatora kNN.

Listing 5.12. Przykład utworzenia klasyfikatora kNN

```
1  #pragma once
2
3  #include <iostream>
4
5  inline void sharkKNN(const shark::ClassificationDataset& trainData,
6                       const shark::ClassificationDataset& testData)
7  {
8      using namespace shark;
9
10     // utworzenie i konfiguracja drzewa oraz algorytmu
11     KDTree<RealVector> tree(trainData.inputs());
12     TreeNearestNeighbors<RealVector,unsigned int> algorithm(
13         trainData, &tree);
14
15     // konfiguracja modelu
16     const unsigned int K = 2; // ilość sąsiadów dla algorytmu kNN
17     NearestNeighborModel<RealVector, unsigned int> KNN(&algorithm, K);
18
19     // ewaluacja modelu
20     std::cout << "-----Shark␣KNN-----" << std::endl;
21     std::cout << "Train␣data␣:" << std::endl;
22     auto predictions = KNN(trainData.inputs());
23     printSharkModelEvaluation(
24         trainData.labels(), predictions, Task::CLASSIFICATION);
25
26     std::cout << "Test␣data:" << std::endl;
27     predictions = KNN(testData.inputs());
28     printSharkModelEvaluation(
29         testData.labels(), predictions, Task::CLASSIFICATION);
30 }
```

5.4.5. Algorytm zbiorowy

Biblioteka Shogun-ML oprócz powszechnie znanych algorytmów udostępnia także bardziej złożone struktury, jak np. model algorytmów złożonych (ang. *ensemble*), bazujący na wykorzystaniu wielu składowych algorytmów bazujących na fragmentach przestrzeni cech, aby później połączyć uzyskane wyniki, osiągając w ten sposób zwiększenie precyzji predykcji. Niestety jedynym występującym w tej bibliotece

mechanizmem wykorzystującym tą technikę jest losowy las (ang. *Random Forest*) złożony z drzew decyzyjnych, umożliwiający jedynie zadanie klasyfikacji (nie jest dostępna możliwość przeprowadzenia z jego użyciem regresji). Klasycznie dla omawianej biblioteki, implementacja odbywa się poprzez utworzenie obiektu klasy trenera, w tym przypadku *RFTrainer*, umożliwiającego konfigurację parametrów, a następnie nauczenie modelu, reprezentowanego przez klasę *RFClassifier*. Oprócz algorytmu Random Forest, istnieje możliwość wykorzystania biblioteki do utworzenia modelu w technice składania (ang. *stacking*), jednak z racji nie występowania tej opcji domyślnie, leży ona poza zakresem niniejszej pracy. Listing 5.13 przedstawia sposób utworzenia i użycia modelu losowego lasu.

Listing 5.13. Utworzenie modelu algorytmu złożonego losowego lasu [5]

```
1  using namespace std;
2
3  void RFClassification(const shark::ClassificationDataset& train,
4                        const shark::ClassificationDataset& test)
5  {
6      using namespace shark;
7
8      // utworzenie i konfiguracja trenera
9      RFTrainer<unsigned int> trainer;
10     trainer.setNTrees(100);
11     trainer.setMinSplit(10);
12     trainer.setMaxDepth(10);
13     trainer.setNodeSize(5);
14     trainer.minImpurity(1.e-10);
15     // utworzenie klasyfikatora
16     RFClassifier<unsigned int> rf;
17     trainer.train(rf, train);
18     // ewaluacja
19     ZeroOneLoss<unsigned int> loss;
20     auto predictions = rf(test.inputs());
21     double accuracy = 1. - loss.eval(test.labels(), predictions);
22     std::cout << "Random␣Forest␣accuracy␣=␣" << accuracy << std::endl;
23 }
```

5.4.6. Sieć neuronowa

Skonstruowanie sieci neuronowej w bibliotece Shark-ML wykorzystuje pewne mechanizmy oferowane przez klasę *LinearModel<>*. Pozwala ona na określenie typu i ilości wejść, wyjść, oraz zastosowania biasu. Każda warstwa składa się z pojedynczego obiektu modelu liniowego, gdzie ilość wyjść określa liczbę neuronów zawartych w warstwie. Konfiguracja funkcji aktywacji neuronu odbywa się na etapie przekazania typów do szablonu modelu. Pełną listę dostępnych funkcji aktywacji znaleźć można w dokumentacji biblioteki [8]. Kolejnym krokiem jest przygotowanie obiektu klasy *ErrorFunction<>* w oparciu o jedną z dostępnych funkcji strat, która zostanie skonfigurowana do wykorzystania przez optymalizator przeprowadzający uczenie. Po przygotowaniu funkcji straty, należy zainicjować sieć losowymi wagami i utworzyć oraz skonfigurować wybrany obiekt klasy optymalizatora. Na tym etapie, sieć jest gotowa do przeprowadzenia procesu uczenia. Polega ono na iteracyjnym wykonywaniu kroków za pomocą funkcji *step()* obiektu optymalizatora. W międzyczasie

możliwe jest także pobranie wartości funkcji straty na każdej epoce uczenia. Z racji konieczności użycia zwykłej pętli zdefiniowanej przez użytkownika, istnieje możliwość określenia własnych warunków stopu ewaluowanych po każdej epoce, jak np. liczba epok lub przekroczenie określonego progu przez uzyskaną wartość funkcji straty. Wewnątrz pętli iterującej po epokach należy umieścić kolejną pętlę, której zadaniem będzie przejście przez wszystkie batche, wykonując na nich krok optymalizatora. Po zakończeniu uczenia, należy skonfigurować obiekt modelu przekazując mu wagi ustalone przez optymalizer, uzyskując w ten sposób gotową instancję wyszkolonej sieci neuronowej. Listing 5.14 przedstawia kod realizujący cały proces, stanowiący przykład z książki „Hands On Machine Learning with C++".

Listing 5.14. Przykład sieci neuronowej o dwóch warstwach ukrytych do zadania klasyfikacji

```
#pragma once

inline void sharkNN(const shark::ClassificationDataset& trainData,
                    const shark::ClassificationDataset& testData)
{
    using namespace shark;

    // zdefiniowanie warstw sieci
    using DenseTanhLayer = LinearModel<RealVector, TanhNeuron>;
    using DenseLinearLayer = LinearModel<RealVector>;
    using DenseLogisticLayer = LinearModel<RealVector, LogisticNeuron>;
    DenseLinearLayer layer1(inputDimension(trainData), 5, true);
    DenseTanhLayer layer2(5, 5, true);
    DenseTanhLayer layer3(5, 5, true);
    DenseLogisticLayer output(5, 1, true);
    // połączenie warstw
    auto network = layer1 >> layer2 >> layer3 >> output;
    // utworzenie i konfiguracja funkcji straty
    DiscreteLoss<> loss;
    ErrorFunction<> error(trainData, &network, &loss, true);
    TwoNormRegularizer<> regularizer(error.numberOfVariables());
    double weightDecay = 0.0001;
    error.setRegularizer(weightDecay, &regularizer);
    error.init();
    // inicjalizacja wag sieci
    initRandomNormal(network, 0.001);
    // utworzenie i konfiguracja optymalizatora
    SteepestDescent<> optimizer;
    optimizer.setMomentum(0.5);
    optimizer.setLearningRate(0.1);
    optimizer.init(error);
    // przeprowadzenie procesu uczenia
    std::size_t epochs = 1000;
    std::size_t iterations = trainData.numberOfBatches();
    // pętla przechodząca poszczególne epoki
    for (std::size_t epoch = 0; epoch != epochs; ++epoch)
    {
        double avgLoss = 0.0;
        // pętla operująca na pojedynczych batch'ach
        for (std::size_t i = 0; i != iterations; ++i)
        {
            // wykonanie kroku optymalizatora
            optimizer.step(error);
```

```
44              // zapisanie częściowej wartości funkcji straty
45              if (i % 100 == 0)
46              {
47                  avgLoss += optimizer.solution().value;
48              }
49          }
50          // wyliczenie średniej wartości funkcji straty
51          avgLoss /= iterations;
52          std::cout << "Epoch " << epoch << " | Avg. loss " << avgLoss
53                    << std::endl;
54      }
55      // konfiguracja modelu docelowego
56      network.setParameterVector(optimizer.solution().point);
57      // walidacja
58      std::cout << "-----Shark Neural -----" << std::endl;
59      std::cout << "Train data:" << std::endl;
60      auto predictions = network(trainData.inputs());
61      printSharkModelEvaluation(
62          trainData.labels(), predictions, Task::CLASSIFICATION);
63
64      std::cout << "Test data:" << std::endl;
65      predictions = network(testData.inputs());
66      printSharkModelEvaluation(
67          testData.labels(), predictions, Task::CLASSIFICATION);
68  }
```

5.5. Metody analizy modeli

5.5.1. Funkcje straty

Biblioteka Shark-ML oferuje szereg funkcji straty pozwalających na wymierną weryfikację dokładności modelu. Należą do nich [9]:

- **średni błąd absolutny** - realizowany za pomocą klasy *AbsoluteLoss*;
- **błąd średniokwadratowy** - realizowany za pomocą klasy *SquaredLoss*;
- **błąd typu zero-one** - realizowany za pomocą klasy *ZeroOneLoss*;
- **błąd dyskretny** - realizowany za pomocą klasy *DiscreteLoss*;
- **entropia krzyżowa** - realizowana za pomocą klasy *CrossEntropy*;
- **błąd typu hinge** - realizowany za pomocą klasy *HingeLoss*;
- **średniokwadratowy błąd typu hinge** - realizowany za pomocą klasy *SquaredHingeLoss*;
- **błąd typu hinge epsilon** - realizowany za pomocą klasy *EpsilonHingeLoss*;
- **średniokwadratowy błąd typu hinge epsilon** - realizowany za pomocą klasy *SquaredEpsilonHingeLoss*;
- **funkcja straty Hubera** - realizowana za pomocą klasy *HuberLoss*;

- **funkcja straty Tukeya** - realizowana za pomocą klasy *TukeyBiweightLoss*.

Każda z powyższych klas używana jest w schematyczny sposób, poprzez wcześniejsze utworzenie obiektu klasy wybranej funkcji straty, a następnie wywołanie jej jako funkcji przekazując wartości oczekiwane oraz otrzymane predykcje modelu. Listing 5.15 przedstawia omówiony sposób użycia na przykładzie błędu średniokwadratowego.

Listing 5.15. Użycie funkcji straty na przykładzie błędu średniokwadratowego

```
using namespace shark;

// [...]

SquaredLoss<> mse_loss;
auto mse = mse_loss(train_data.labels(), predictions);
auto rmse = std::sqrt(mse);
```

5.5.2. Metryka R^2 i adjusted R^2

Biblioteka Shark-ML nie oferuje bezpośredniej klasy reprezentującej metrykę R^2 jak w przypadku funkcji strat, jednak udostępnia użytkownikowi funkcję obliczania wariancji danych, co umożliwia bardzo łatwą samodzielną implementację obu metryk. Listing 5.16 przedstawia sposób ich wyliczenia, posiadając wartość błędu średniokwadratowego.

Listing 5.16. Implementacja metryk R^2 oraz adjusted R^2

```
using namespace shark;

// [...]

// błąd średniokwadratowy
SquaredLoss<> mse_loss;
auto mse = mse_loss(train_data.labels(), predictions);

// metryka R^2
auto var = variance(train_data.labels());
auto r_squared = 1 - mse / var(0);

// metryka adjusted R^2
auto adj_r_squared = 1 - (1 - r_squared)((num_regressors - 1)/
                     (num_regressors - data_size - 1));
```

5.5.3. Pole pod wykresem krzywej charakterystycznej odbiornika

Pole pod wykresem krzywej charakterystycznej odbiornika stanowi jedną z często wykorzystywanych metryk poprawności predykcji modelu, w związku z czym nie mogło jej zabraknąć w bibliotece Shark-ML. Jest ona dostępna za pośrednictwem klasy *NegativeAUC*, wykorzystywanej w taki sam sposób jak pozostałe omówione wcześniej funkcje straty. W przeciwieństwie do standardowego podejścia, wspomniana

klasa oblicza odwróconą wartość pola pod wykresem funkcji ROC, aby umożliwić wykorzystanie jej jako minimalizowanego celu w procesie uczenia. Listing 5.17 przedstawia sposób obliczenia wartości wymienionej metryki.

Listing 5.17. Przykład obliczenia pola pod wykresem funkcji ROC dla Shark-ML

```
using namespace std;

// [...]

constexpr bool invertToPositiveROC = true;
NegativeAUC<> roc(invertToPositiveROC);
auto auc_roc = roc(train_data.labels(), predictions);
```

5.5.4. Sprawdzian krzyżowy K-krotny

Proces poszukiwania najlepszych wartości hiperparametrów w Shark-ML uwzględnia przeprowadzenie wewnętrznie uczenia danego modelu, lecz skupia się na porównaniu uzyskiwanych wyników, w związku z czym jego opis zamieszczony został w tej sekcji. Użycie implementacji metody sprawdzianu krzyżowego K-fold w wymaga wykorzystania trzech klas. Pierwszą z nich stanowi *CVFolds*, której zadaniem jest przechowanie zestawu danych podzielonego na odpowiednią ilość fragmentów. Drugą jest klasa *CrossValidationError* stanowiąca szablon przyjmujący typ modelu, dla którego określany będzie błąd walidacji, oraz obiekt klasy błędu, który ma zostać wyliczony. Ostatnią klasą jest *GridSearch*, którego zadaniem jest iteracyjny wybór fragmentów do uczenia i wyliczenie wartości hiperparametrów dla modelu. Wynikiem procesu jest uzyskanie najlepszego zestawu hiperparametrów do procedury szkolenia - użytkownik musi zawołać metodę *step()* klasy *GridSearch* tylko jeden raz. Listing 5.18 przedstawia przykład zawarty w książce „Hands On Machine Learning with C++" [5], w którym autor przedstawia proces wykorzystania powyższych klas na własnoręcznie zaimplementowanym modelu regresji wielomianowej.

Listing 5.18. Przykład realizacji sprawdzianu krzyżowego K-fold w Shark-ML

```
using namespace shark;

// [...]

// przetworzenie danych uczących
const unsigned int num_folds = 5;
CVFolds<RegressionDataset> folds =
    createCVSameSize<RealVector, RealVector>(train_data, num_folds);

// przygotowanie parametrów dla docelowego modelu
double regularization_factor = 0.0;
double polynomial_degree = 8;
int num_epochs = 300;

// konfiguracja docelowego modelu
PolynomialModel<> model;
PolynomialRegression trainer(regularization_factor, polynomial_degree,
                             num_epochs);

```

```
20 // utworzenie obiektu błędu oraz sprawdzianu krzyżowego
21 AbsoluteLoss<> loss;
22 CrossValidationError<PolynomialModel<>, RealVector> cv_error(
23     folds, &trainer, &model, &trainer, &loss);
24
25 // utworzenie siatki
26 GridSearch grid;
27 std::vector<double> min(2);
28 std::vector<double> max(2);
29 std::vector<std::size_t> sections(2);
30
31 // regularyzacja
32 min[0] = 0.0;
33 max[0] = 0.00001;
34 sections[0] = 6;
35
36 // stopień wielomianu
37 min[1] = 4;
38 max[1] = 10.0;
39 sections[1] = 6;
40 grid.configure(min, max, sections);
41
42 // proces uczenia i konfiguracja modelu
43 grid.step(cv_error);
44
45 trainer.setParameterVector(grid.solution().point);
46 trainer.train(model, train_data);
```

5.6. Dostępność dokumentacji i źródeł wiedzy

Biblioteka Shark-ML posiada skróconą dokumentację dostępną na głównej stronie internetowej projektu, wraz z przykładowymi plikami źródłowymi dołączonymi do repozytorium. Jest ona także wspomniana w książce „Hands-On Machine Learning with C++", przedstawiającej sposoby użycia wybranych funkcjonalności. Kwestią wyróżniającą ją natomiast na tle pozostałych bibliotek omówionych w ramach niniejszej pracy jest fakt, że jest ona dedykowana dla języka C++, w związku z czym dużo łatwiej dostępne są wątki społecznościowe i artykuły omawiające realizację różnorodnych typów modeli z jej użyciem, oraz oferując przykładowy kod źródłowy. Strona główna projektu [10] posiada rozbudowaną, przejrzystą i opatrzoną przykładami dokumentację, znacznie ułatwiając korzystanie z dostępnego API.

Rozdział 6

Biblioteka Dlib

6.1. Wprowadzenie

Jest to biblioteka do uczenia maszynowego napisana w nowoczesnym C++, o zastosowaniu przemysłowym oraz naukowym [11]. Podobnie jak poprzednio omawiane biblioteki, posiada ona otwarte źródło na licencji Boost Software Licence [12]. Do dziedzin wykorzystujących wyżej wspomnianą bibliotekę należą robotyka, systemy wbudowane, telefonia komórkowa oraz śrowodiska o dużej wydajności obliczeniowej. Kod źródłowy biblioteki opatrzony jest testami jednostkowymi, co pozwala na łatwiejsze utrzymanie jakości dostarczanego rozwiązania. Ciekawym aspektem jest fakt, że Dlib stanowi nie tylko bibliotekę, lecz zestaw narzędzi, oferujący funkcjonalności wykraczające także poza dziedzinę uczenia maszynowego.

6.2. Formaty źródeł danych

Do reprezentacji wektora w bibliotece Dlib wykorzystywane są kontenery z biblioteki szablonów STL języka C++. Dodatkowo, istnieje możliwość ich inicjalizacji za pomocą operatora przecinka, oraz opakowania surowej tablicy (ang. *raw array*). Oznacza to, że podobnie jak w przypadku biblioteki Shogun, dane mogą być przekazywane do programu wykorzystującego Dlib w dowolny sposób zapewniający umieszczenie ich np. w surowej tablicy do późniejszego przetworzenia na obiekty akceptowane przez bibliotekę. Metoda ta działa także z kontenerami biblioteki STL, które pozwalają na dostęp do surowych danych przy użyciu metody *data()*. Tak samo jak poprzednio, występuje tu wsparcie dla formatu CSV obwarowanego tymi samymi ograniczeniami co dla Shogun. Za wspomniane wsparcie odpowiada przeładowany operator strumienia współpracujący z klasą *std::ifstream* biblioteki standardowej C++. Przykładowy kod wykorzystujący opisany mechanizm zamieszczony został na listingu 6.1.

Listing 6.1. Fragment kodu ilustrujący sposób odczytu z pliku w formacie CSV [5]

```
1 #include <Dlib/matrix.h>
2 #include <fstream>
3 #include <iostream>
4
5 using namespace Dlib;
```

```
6
7  // [...]
8
9  matrix<double> data;
10 std::ifstream file("data_file.csv");
11 file >> data;
12 std::cout << data << std::endl;
```

6.3. Metody przetwarzania i eksploracji danych

6.3.1. Normalizacja

Biblioteka udostępnia normalizację danych poprzez standaryzację, realizowaną przez klasę *Dlib::vector_normalizer*. Głównym warunkiem ograniczającym zastosowanie jej jest fakt, że nie można w niej umieścić całego zestawu danych treningowych na raz, co wymusza podział obserwacji na osobne wektory, a następnie umieszczenie ich w kontenerze *std::vector* do dalszego przetwarzania. Przykład funkcji normalizującej przedstawiono na listingu 6.2.

Listing 6.2. Funkcja normalizująca

```
1  #pragma once
2
3  #include <vector>
4  #include <dlib/matrix.h>
5
6  inline std::vector<dlib::matrix<double>> dlibNormalize(
7      const std::vector<dlib::matrix<double>& data)
8  {
9      using namespace dlib;
10     // utworzenie i trening normalizera
11     vector_normalizer<matrix<double>> normalizer;
12     normalizer.train(data);
13     // przetwarzanie danych
14     std::vector<matrix<double>> processedData(data.size());
15     for (auto dataMatrix : data)
16         processedData.emplace_back(normalizer(data));
17     return processedData;
18 }
```

6.3.2. Redukcja wymiarowości

6.3.2.1. PCA

Implementacja metody PCA w bibliotece Dlib oferowana jest za pośrednictwem klasy *dlib::vector_normalizer_pca*, która oprócz samej redukcji wymiarowości wykonuje także wcześniej automatycznie proces normalizacji danych. Bywa to przydatne, gwarantując że redukcja będzie przeprowadzana zawsze na odpowiednio przygotowanych wartościach obserwacji. Listing 6.3 przedstawia funkcję używającą wyżej wymienioną metodę.

Listing 6.3. Przykład redukcji wymiarowości z użyciem metody PCA

```
#pragma once

#include <dlib/matrix.h>
#include <dlib/matrix/matrix_utilities.h>
#include <dlib/statistics.h>
#include <vector>

inline std::vector<dlib::matrix<double>> dlibPCA(
    const std::vector<dlib::matrix<double>>& data,
    std::size_t desiredDimensions)
{
    using namespace dlib;
    // utworzenie i trening reduktora
    vector_normalizer_pca<matrix<double>> pca;
    pca.train(data, desiredDimensions/data[0].nr());
    // przygotowanie kontenera na przetworzone dane
    std::vector<matrix<double>> processedData;
    processedData.reserve(data.size());
    // przetwarzanie danych
    for(auto& sample : data)
    {
        processedData.emplace_back(pca(sample));
    }
    return processedData;
}
```

6.3.2.2. Liniowa analiza dyskryminacyjna

Drugim z oferowanych algorytmów redukcji wymiarowości zawartych w Dlib jest algorytm liniowej analizy dyskryminacyjnej. Jest on dostępny pod postacią funkcji *dlib::compute_lda_transform*, która przekształca macierz zawierającą dane wejściowe do macierzy transformacji danych. Ze względu na nadzorowany charakter algorytmu, konieczne jest także przekazanie wartości zmiennych odpowiedzi, natomiast same dane, w przeciwieństwie do metody PCA, mogą być zawarte w pojedynczym obiekcie macierzy. Redukcja odbywa się poprzez wymnożenie otrzymanej macierzy przez transponowany wiersz zawierający próbkę. Szczegółowe zastosowanie algorytmu przedstawiono na listingu 6.4

Listing 6.4. Przykład redukcji wymiarowości z użyciem algorytmu LDA

```
#pragma once

#include <dlib/matrix.h>
#include <dlib/matrix/matrix_utilities.h>
#include <dlib/statistics.h>
#include <vector>

inline std::vector<dlib::matrix<double>> dlibLDA(
    const dlib::matrix<double>& data,
    const std::vector<unsigned long>& labels
    std::size_t desiredDimensions)
{
    using namespace dlib;
    // utworzenie obiektow potrzebnych do redukcji
```

```
15      matrix<double, 0, 1> mean;
16      auto transform = data;
17      // przeksztalcenie danych w macierz redukcyjną
18      compute_lda_transform(transform, mean, labels, desiredDimensions);
19      // przygotowanie kontenera na przetworzone dane
20      std::vector<matrix<double>> transformedData;
21      transformedData.reserve(data.nr());
22      // redukcja wymiarowości
23      for (long i = 0; i < data.nr(); ++i)
24      {
25          transformedData.emplace_back(transform * trans(rowm(data, i)) - mean);
26      }
27      // zwrócenie wektora przetworzonych wierszy
28      return transformedData;
29  }
```

6.3.2.3. Mapowanie Sammona

Całość algorytmu mapowania Sammmona implementowana jest za pomocą klasy *dlib::sammon_projection*, i ogranicza się do utworzenia jej instancji. Wykorzystując metodę należy przekazać do utworzonego obiektu za pomocą operatora wywołania funkcji wektor danych, oraz oczekiwaną ilość wymiarów, otrzymując przekształcone dane. W związku z powyższym, funkcja realizująca redukcję z użyciem wyżej wymienionej metody sprowadza się do wykorzystania dwóch linii. Dokładny sposób jej użycia pokazano na listingu **??**

Listing 6.5. Przykład redykcji wymiarowości z użyciem mapowania Sammmona

```
1   #pragma once
2
3   #include <dlib/matrix.h>
4   #include <dlib/matrix/matrix_utilities.h>
5   #include <dlib/statistics.h>
6   #include <vector>
7
8   inline std::vector<matrix<double>> dlibSammon(
9       const std::vector<matrix<double>>& data,
10      long desiredDimensions)
11  {
12      using namespace dlib;
13      // utworzenie obiektu mapowania
14      dlib::sammon_projekction sammon;
15      // redukcja wymiarowości
16      return sammon(data, desiredDimensions);
17  }
```

6.4. Modele uczenia maszynowego

6.4.1. Regresja liniowa

Biblioteka Dlib posiada pośrednią realizację modelu regresji liniowej. Wykorzystuje ona technikę brzegowej regresji jądra, przekazując w formie kernela jądro liniowe.

Następnie przeprowadzany jest trening, zapisując docelowy model w postaci funkcji decyzyjnej. Listing 6.6 przedstawia szczegóły powyższego mechanizmu.

Listing 6.6. Przykład regresji liniowej w Dlib

```
#pragma once

#include <dlib/matrix.h>
#include <dlib/svm.h>

#include <inc/dlib/eval.hpp>

inline void dlibLinear(std::vector<dlib::matrix<double>> data,
                       std::vector<double> labels)
{
    using namespace dlib;
    // utworzenie oraz konfiguracja trenera i jądra
    using linearKernel = linear_kernel<matrix<double>>;
    krr_trainer<matrix<double>> trainer;
    trainer.set_kernel(linearKernel());
    // podział danych
    auto dataSplit = data.begin() + data.size() * 0.8;
    auto trainData = std::vector<matrix<double>>(
        data.begin(), dataSplit);
    auto testData = std::vector<matrix<double>>(
        dataSplit, data.end());
    auto labelSplit = labels.begin() + labels.size() * 0.8;
    auto trainLabels = std::vector<matrix<double>>(
        labels.begin(), labelSplit);
    auto testLabels = std::vector<matrix<double>>(
        labelSplit, labels.end());
    // trening
    decision_function<linearKernel> model = trainer.train(
        trainData, trainLabels);
    // ewaluacja
    std::cout << "-----␣Dlib␣Linear␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = std::vector<double>(trainData.size());
    for (auto& sample : trainData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, trainLabels, Task::REGRESSION);
    predictions.clear();
    std::cout << "Test␣data:" << std::endl;
    for (auto& sample : testData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, testLabels, Task::REGRESSION);
}
```

6.4.2. Maszyna wektorów nośnych

W celu realizacji wieloklasowej klasyfikacji z użyciem maszyny wektorów nośnych, biblioteka Dlib oferuje klasę funkcji decyzyjnej *dlib::one_vs_one_decision_function*.

Przechowuje ona wynikowy model uczenia algorytmem maszyny wektorów nośnych, zawarty w klasie *one_versus_one*, do której przesłany zostaje trener SVM. Dokładny sposób użycia został przedstawiony na listingu 6.7.

Listing 6.7. Przykład użycia maszyny wektorów nośnych w Dlib

```
#pragma once

#include <dlib/matrix.h>
#include <dlib/svm_threaded.h>
#include <inc/dlib/eval.hpp>
#include <vector>

inline void dlibSVM(std::vector<dlib::matrix<double>> data,
                    std::vector<double> labels)
{
    using namespace dlib;
    // podział danych
    auto dataSplit = data.begin() + data.size() * 0.8;
    auto trainData = std::vector<matrix<double>>(
        data.begin(), dataSplit);
    auto testData = std::vector<matrix<double>>(
        dataSplit, data.end());
    auto labelSplit = labels.begin() + labels.size() * 0.8;
    auto trainLabels = std::vector<matrix<double>>(
        labels.begin(), labelSplit);
    auto testLabels = std::vector<matrix<double>>(
        labelSplit, labels.end());

    using OVOTrainer = one_vs_one_trainer<any_trainer<double>>;
    using Kernel = radial_basis_kernel<double>;
    // utworzenie trenera maszyny wektorów nośnych
    svm_nu_trainer<Kernel> svmTrainer;
    svmTrainer.set_kernel(Kernel(0.1));
    // utworzenie trenera klasyfikatora wieloklasowego
    OVOTrainer trainer;
    trainer.set_trainer(svmTrainer);
    // utworzenie modelu
    one_vs_one_decision_function<OVOTrainer> model =
        trainer.train(trainData, trainLabels);
    // ewaluacja
    std::cout << "-----␣Dlib␣SVM␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = std::vector<double>(trainData.size());
    for (auto& sample : trainData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, trainLabels, Task::CLASSIFICATION);
    predictions.clear();
    std::cout << "Test␣data:" << std::endl;
    for (auto& sample : testData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, testLabels, Task::CLASSIFICATION);
}
```

6.4.3. Sieci neuronowe

Konstrukcja sieci neuronowej w przypadku biblioteki Dlib rozpoczyna się od zdefiniowania architektury sieci, za pomocą odpowiedniego łańcucha szablonów. Parametry tworzą sieć w kolejności od najbardziej zagnieżdżonego do najbardziej zewnętrznego. Dlib zapewnia użytkownikowi rozdzielność typu warsty od jej funkcji aktywacji, w związku z czym użytkownik może dokładnie dostosować działanie sieci. Niestety, sama składnia tworzonej architektury jest przez to zaciemniona, co utrudnia jej analizę. Po utworzeniu architektury, należy przygotować i skonfigurować solver. Najpopularniejszym z oferowanych przez bibliotekę jest algorytm stochastycznego spadku gradientowego, zaimplementowanego w postaci klasy *dlib::sgd*. Trzeci krok stanowi konfiguracja trenera głębokich sieci neuronowych, oferowanego przez klasę *dlib::dnn_trainer*, poprzez ustawienie parametrów takich jak:

- współczynnik uczenia;
- współczynnik zmiany szybkości uczenia;
- rozmiar mini-batch'y;
- maksymalna ilość epok.

Obiekt trenera w trakcie jego tworzenia przyjmuje referencję do architektury sieci oraz do obiektu solvera. Proces nauki odbywa się poprzez wywołanie funkcji *train()*, natomiast wynikowy model zapisany zostaje w obiekcie architektury sieci. Listing 6.8 przedstawia przykład budowy sieci neuronowej z wykorzystaniem niniejszej biblioteki.

Listing 6.8. Przykład sieci neuronowej w Dlib

```
1  #pragma once
2
3  #include <dlib/dnn.h>
4  #include <dlib/matrix.h>
5
6  #include <inc/dlib/eval.hpp>
7
8  #include <vector>
9
10 inline void dlibNeural(std::vector<dlib::matrix<double>> data,
11                        std::vector<double> labels)
12 {
13     using namespace dlib;
14     // podział danych
15     auto dataSplit = data.begin() + data.size() * 0.8;
16     auto trainData = std::vector<matrix<double>>(
17         data.begin(), dataSplit);
18     auto testData = std::vector<matrix<double>>(
19         dataSplit, data.end());
20     auto labelSplit = labels.begin() + labels.size() * 0.8;
21     auto trainLabels = std::vector<matrix<double>>(
22         labels.begin(), labelSplit);
23     auto testLabels = std::vector<matrix<double>>(
24         labelSplit, labels.end());
```

```
25      // zdefiniowanie architektury sieci
26      using Architecture = loss_mean_squared<fc <1,
27                               htan<fc<5,
28                               htan<fc<5,
29                               input<matrix<double>>>>>>>>;
30      // utworzenie sieci
31      Architecture model;
32      // utworzenie i konfiguracja algorytmu optymalizacji
33      float weightDecay = 0.0001f;
34      float momentum = 0.5f;
35      sgd solver(weightDecay, momentum);
36      // utworzenie i konfiguracja trenera
37      dnn_trainer<Architecture> trainer(model, solver);
38      trainer.set_learning_rate(0.1);
39      trainer.set_learning_rate_shrink_factor(1);
40      trainer.set_mini_batch_size(64);
41      trainer.set_max_num_epochs(100);
42      trainer.be_verbose();
43      // trening
44      trainer.train(trainData, trainLabels);
45      model.clean();
46      // ewaluacja
47      std::cout << "-----␣Dlib␣Neural␣-----" << std::endl;
48      std::cout << "Train␣data:" << std::endl;
49      auto predictions = model(trainData);
50      dlibEval(predictions, trainLabels, Task::CLASSIFICATION);
51
52      std::cout << "Test␣data:" << std::endl;
53      predictions = model(testData);
54      dlibEval(predictions, testLabels, Task::CLASSIFICATION);
55  }
```

6.4.4. Brzegowa regresja jądra

Przygotowanie wieloklasowego modelu brzegowej regresji jądra (ang. *Kernel Ridge Regression*) w przypadku biblioteki Dlib wygląda prawie identycznie do sposobu realizacji wieloklasowej maszyny wektorów nośnych. Główną różnicą jest wykorzystany trener podstawowy, w tym wypadku stanowiący obiekt klasy *dlib::krr_trainer*. Możliwe jest także wykorzystanie tego samego typu jądra, co w przypadku maszyny wektorów nośnych. Listing 6.9 obrazuje szczegółowy sposób przygotowania modelu.

Listing 6.9. Przykład użycia brzegowej regresji jądra w Dlib

```
1   #pragma once
2
3   #include <dlib/matrix.h>
4   #include <dlib/svm_threaded.h>
5
6   #include <inc/dlib/eval.hpp>
7
8   #include <vector>
9
10  inline void dlibKRR(std::vector<dlib::matrix<double>> data,
11                      std::vector<double> labels)
12  {
```

```
13      using namespace dlib;
14      // podział danych
15      auto dataSplit = data.begin() + data.size() * 0.8;
16      auto trainData = std::vector<matrix<double>>(
17          data.begin(), dataSplit);
18      auto testData = std::vector<matrix<double>>(
19          dataSplit, data.end());
20      auto labelSplit = labels.begin() + labels.size() * 0.8;
21      auto trainLabels = std::vector<matrix<double>>(
22          labels.begin(), labelSplit);
23      auto testLabels = std::vector<matrix<double>>(
24          labelSplit, labels.end());
25
26      using OVOTrainer = one_vs_one_trainer<any_trainer<double>>;
27      using Kernel = radial_basis_kernel<double>;
28      // utworzenie trenera maszyny wektorów nośnych
29      krr_trainer<Kernel> krrTrainer;
30      krrTrainer.set_kernel(Kernel(0.1));
31      // utworzenie trenera klasyfikatora wieloklasowego
32      OVOTrainer trainer;
33      trainer.set_trainer(krrTrainer);
34      // utworzenie modelu
35      one_vs_one_decision_function<OVOTrainer> model =
36          trainer.train(trainData, trainLabels);
37      // ewaluacja
38      std::cout << "-----␣Dlib␣KRR␣-----" << std::endl;
39      std::cout << "Train␣data:" << std::endl;
40      auto predictions = std::vector<double>(trainData.size());
41      for (auto& sample : trainData)
42      {
43          predictions.emplace_back(model(sample));
44      }
45      dlibEval(predictions, trainLabels, Task::CLASSIFICATION);
46      predictions.clear();
47      std::cout << "Test␣data:" << std::endl;
48      for (auto& sample : testData)
49      {
50          predictions.emplace_back(model(sample));
51      }
52      dlibEval(predictions, testLabels, Task::CLASSIFICATION);
53  }
```

6.5. Metody analizy modeli

6.5.1. Pole pod wykresem krzywej charakterystycznej odbiornika

Biblioteka Dlib posiada implementację funkcji wyznaczającej krzywą ROC, jednak wymaga ona pewnego przetwarzania danych przed i po jej użyciu. W celu jej zastosowania należy podzielić dane sklasyfikowane prawidłowo i nieprawidłowo. Wynikiem funkcji jest wektor zawierający współrzędne poszczególnych punktów krzywej charakterystycznej odbiornika, które pozwalają na narysowanie wykresu na płaszczyźnie dwuwymiarowej. Obliczenie wartości pola pod wykresem należy dokonać

ręcznie, wykorzystując np. jedną z numerycznych metod na obliczenie całki. Listing 6.10 przedstawia funkcję ewaluującą predykcje modelu, w tym obliczenie wartości pola pod wykresem krzywej charakterystycznej odbiornika dla zadania klasyfikacji.

Listing 6.10. Obliczenie pola pod wykresem funkcji ROC dla Dlib

```
#pragma once

#include <cmath>
#include <dlib/statistics.h>

enum class Task
{
    CLASSIFICATION,
    REGRESSION
};

inline dlibEval(std::vector<double> predictions,
                std::vector<double> labels,
                Task task)
{
    using namespace dlib;

    if (task == Task::CLASSIFICATION)
    {
        // przygotowanie kontenerów na podzielone dane
        std::vector<double> correct;
        std::vector<double> incorrect;
        correct.reserve(predictions.size());
        incorrect.reserve(predictions.size());
        // przygotowanie wartości detektora
        constexpr int positiveDetectionScore = 1;
        constexpr int negativeDetectionScore = 0;
        // podział danych
        for (int i = 0; i < predictions.size() && i < labels.size(); ++i)
        {
            if (predictions[i] == labels[i])
                correct.emplace_back(positiveDetectionScore);
            else
                incorrect.emplace_back(negativeDetectionScore);
        }
        // obliczenie krzywej roc
        auto roc = compute_roc_curve(correct, incorrect);
        // obliczenie pola pod wykresem roc
        double aucRoc = 0.0;
        for (int i = 0; i < roc.size() - 1; i++)
        {
            auto avg = (roc[i+1].true_positive_rate - roc[i].true_positive_rate)/2.0;
            auto interval = roc[i+1].false_positive_rate - roc[i].false_positive_rate;
            aucRoc += avg * interval;
        }
        std::cout << "AUC␣ROC:␣" << aucRoc << std::endl;
    }
    else
    {
        // obliczenie sumy kwadratów różnic
        auto sum = 0.0;
        for (int i = 0; i < predictions.size() && i < labels.size(); ++i)
```

```
53          {
54              sum += pow(labels[i] - predictions[i], 2);
55          }
56          // obliczenie błędu średniokwadratowego
57          auto mse = sqrt(sum / predictions.size());
58          std::cout << "MSE:␣" << mse << std::endl;
59      }
60  }
```

6.5.2. Sprawdzian krzyżowy K-krotny

Przeprowadzenie sprawdzianu krzyżowego z wykorzystaniem biblioteki Dlib stanowi złożony proces. Jest on realizowany wielofazowo, rozpoczynając od zdefiniowania własnej funkcji obliczającej interesujący użytkownika wynik metryki sprawdzianu, w oparciu o funkcję *dlib::cross_validate_regression_trainer()*. Zewnętrzna część ustalania docelowych wartości hiperparametrów dokonywana jest przez funkcję *dlib::find_min_global()* przyjmującą adres stworzonej funkcji optymalizacyjnej, kontenery przechowujące informacje o minimalnych i maksymalnych dopuszczalnych wartościach poszczególnych hiperparametrów, oraz ilość dozwolonych wywołań funkcji optymalizacyjnej. Aby odczytać wynikowe wartości, należy sięgnąć do kolejnych pół składowej x zwróconej przez funkcję *find_min_global()* struktury. Szczegóły zostały zaprezentowane w oparciu o przykład z książki „Hands-On Machine Learning with C++" [5] na listingu 6.11.

Listing 6.11. Przykład realizacji sprawdzianu krzyżowego

```
1  #pragma once
2
3  #include <dlib/global_optimization.h>
4  #include <dlib/matrix.h>
5  #include <dlib/svm.h>
6
7  #include <cmath>
8
9  inline void dlibCrossValidate(std::vector<dlib::matrix<double>> data,
10                                std::vector<double> labels)
11 {
12     using namespace dlib;
13     // podział danych
14     auto dataSplit = data.begin() + data.size() * 0.8;
15     auto trainData = std::vector<matrix<double>>(
16         data.begin(), dataSplit);
17     auto testData = std::vector<matrix<double>>(
18         dataSplit, data.end());
19     auto labelSplit = labels.begin() + labels.size() * 0.8;
20     auto trainLabels = std::vector<matrix<double>>(
21         labels.begin(), labelSplit);
22     auto testLabels = std::vector<matrix<double>>(
23         labelSplit, labels.end());
24     // utworzenie funkcji sprawdzianu krzyżowego
25     auto crossValidationScore = [&](const double gamma, const double c,
26                                     const double degreeIn)
27     {
28         using namespace dlib;
```

```
29
30          auto degree = std::floor(degreeIn);
31          // zdefiniowanie jądra
32          using Kernel = polynomial_kernel<double>;
33          // przygotowanie i konfiguracja trenera
34          svr_trainer<Kernel> trainer;
35          trainer.set_kernel(Kernel(gamma, c, degree));
36          // obliczenie metryk sprawdzianu krzyżowego
37          matrix<double> result = cross_validate_regression_trainer(
38              trainer, trainData, trainLabels, 10);
39          // zwrócenie metryki błędu średniokwadratowego
40          return result(0, 0);
41      }
42      // przeprowadzenie sprawdzianu
43      auto result = find_min_global(
44          crossValidationScore,
45          {0.01, 1e-8, 5}, // wartości minimalne
46          {0.1, 1, 15},    // wartości maksymalne
47          max_function_calls(50));
48      // odczytanie ustalonych wartości hiperparametrów
49      auto gamma = result.x(0);
50      auto c = result.x(1);
51      auto degree = result.x(2);
52      // utworzenie modelu w oparciu o ustalone hiperparametry
53      using Kernel = polynomial_kernel<double>;
54      svr_trainer<Kernel> trainer;
55      trainer.set_kernel(Kernel(gamma, c, degree));
56      auto model = trainer.train(trainData, trainLabels);
57      // ewaluacja
58      std::cout << "-----␣Dlib␣CrossValidated␣SVM␣-----" << std::endl;
59      std::cout << "Train␣data:" << std::endl;
60      auto predictions = std::vector<double>(trainData.size());
61      for (auto& sample : trainData)
62      {
63          predictions.emplace_back(model(sample));
64      }
65      dlibEval(predictions, trainLabels, Task::REGRESSION);
66      predictions.clear();
67      std::cout << "Test␣data:" << std::endl;
68      for (auto& sample : testData)
69      {
70          predictions.emplace_back(model(sample));
71      }
72      dlibEval(predictions, testLabels, Task::REGRESSION);
73  }
```

6.6. Dostępność dokumentacji i źródeł wiedzy

Dlib posiada zbiór przykładów w postaci listingów kodów źródłowych realizujących poszczególne mechanizmy, dostępnych na stronie głównej projektu [**dlib:home**]. Jest ona także jedną z głównych bibliotek omawianych w ramach wspomnianej wcześniej książki „Hands-On Machine Learning with C++”. Niestety większość forów społecznościowych skupia się na pracy z Dlib z poziomu interfejsu języka Python, co może utrudnić szukanie rozwiązań dla specyficznych przypadków. Warto wspomnieć, że

oprócz funkcjonalności uczenia maszynowego, Dlib realizuje także inne zadania, jak np. networking, co sprawia, że nawigacja po stronie projektu jest lekko utrudniona przez obecność potencjalnie nieinteresujących użytkownika elementów.

Rozdział 7

Zestawienie zbiorcze i podsumowanie

7.1. Oferowane funkcjonalności

7.2. Porównanie wyników dla zadanych przykładów

7.3. Wymagany nakład pracy i jakość źródeł

W procesie pracy z poszczególnymi bibliotekami zauważono, że najmniejszą ilością potrzebnego wkładu pracy charakteryzowała się biblioteka Shark-ML. Wynika to z bardzo przyjaznej dla użytkownika składni, oraz dokładnej dokumentacji dostępnej na stronie internetowej projektu, wraz z przykładami wykorzystania poszczególnych metod. Biblioteka także bez jakichkolwiek problemów została zbudowana i zainstalowana na systemie operacyjnym Ubuntu 22.04 w środowisku WSL, pozwalając bardzo szybko przejść do docelowej pracy.

Drugą biblioteką pod względem koniecznego wkładu czasu okazał się zestaw narzędziowy Dlib. Posiada on stronę projektu z wylistowanymi klasami oraz funkcjami dostępnymi w bibliotece, jednak opis działania poszczególnych metod jest bardzo pobieżny, oraz brakuje dostępnych przykładów. Składnia biblioteki może stanowić wyzwanie dla użytkownika, gdyż nie zawsze jest oczywista, i momentami utrudnia analizę realizowanych przez program operacji.

Jako najbardziej wymagającą bibliotekę uznano Shogun. W chwili pisania niniejszej pracy, zarówno oficjalne repozytorium projektu jak i repozytorium dystrybucji dla systemu operacyjnego Ubuntu okazało się być niekompletne. Uniemożliwiło to zainstalowanie biblioteki za pomocą wbudowanego managera pakietów oraz zbudowanie jej ze względu na nienaprawione zależności do przeniesionych repozytoriów stron trzecich. Mimo przyjaźniejszej składni niż w przypadku Dlib, wspomniany wcześniej mankament sprawia, że w celu pobrania biblioteki konieczne okazało się zainstalowanie specjalnego managera pakietów *nix* posiadającego starszą wersję projektu Shogun dostępną na swoim repozytorium. Z racji braku dostępnej online dokumentacji projektu Shogun oraz faktu, że generowane przykłady nie odnoszą się

do API biblioteki lecz używają jej w kompletnie odrębny, nienaturalny dla projektu sposób, ustalenie funkcjonalności oraz sposobu realizacji poszczególnych zadań uczenia maszynowego musiało zostać oparte praktycznie wyłącznie o materiały dostępne w formie książkowej. Znacznie utrudnia i wydłuża to proces zastosowania biblioteki do jakiegokolwiek projektu.

Bibliografia

[1] Olvi L. Mangasarian Dr William H. Wolberg W. Nick Street. *Wisconsin Diagnostic Breast Cancer (WDBC)*. 1995. URL: https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic).

[2] Dheeru Dua i Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[3] Trevor Bihl. *Biostatistics Using JMP: A Practical Guide*. Cary, NC: SAS Institute Inc., 2017.

[4] shogun toolbox. *Shogun*. 2020. URL: https://github.com/shogun-toolbox/shogun.

[5] Kirill Kolodiazhnyi. *Hands-On Machine Learning with C++*. Packt Publishing, Maj 2020.

[6] Christian Igel, Verena Heidrich-Meisner i Tobias Glasmachers. "Shark". W: *Journal of Machine Learning Research* 9 (2008), s. 993–996.

[7] mlcpp. *Classification with Shark-ML machine learning library*. 2018. URL: https://github.com/Kolkir/mlcpp/tree/master/classification_shark.

[8] The Shark developer team. *Neuron activation functions*. 2018. URL: https://www.shark-ml.org/doxygen_pages/html/group__activations.html.

[9] The Shark developer team. *Loss and Cost Functions*. 2018. URL: http://image.diku.dk/shark/sphinx_pages/build/html/rest_sources/tutorials/concepts/library_design/losses.html.

[10] The Shark developer team. *Shark - Machine Learning*. 2018. URL: https://www.shark-ml.org/.

[11] Davis E. King. "Dlib-ml: A Machine Learning Toolkit". W: *Journal of Machine Learning Research* 10 (2009), s. 1755–1758.

[12] Dlib team. *Dlib License*. 2003. URL: http://dlib.net/license.html.