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

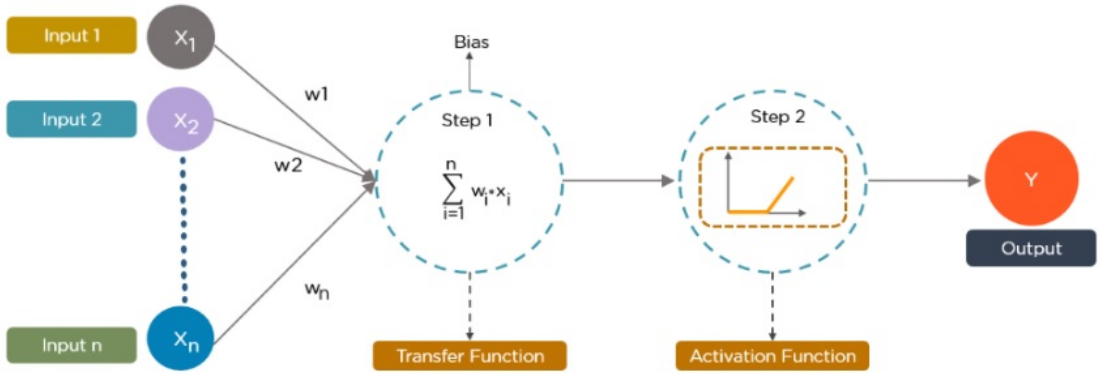
2.1. Problemy współczesnego uczenia maszynowego

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.

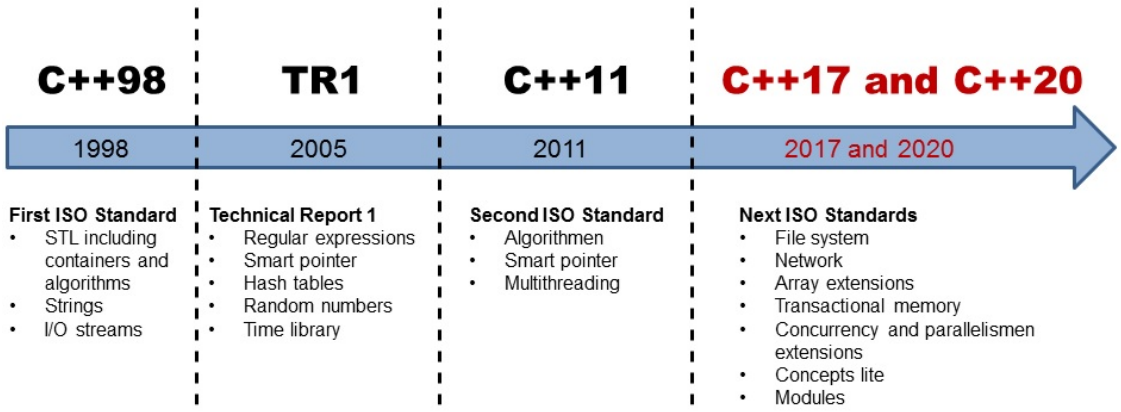
W trakcie rozwoju technologii uczenia maszynowego, postawiono stanowcze kroki w kierunku rozwiązywania powyższych problemów, aby sprostać narastającym wymaganiom związanym z coraz to nowymi i bardziej skomplikowanymi zastosowaniami sztucznej inteligencji. Dokonywano tego poprzez między innymi optymalizację 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

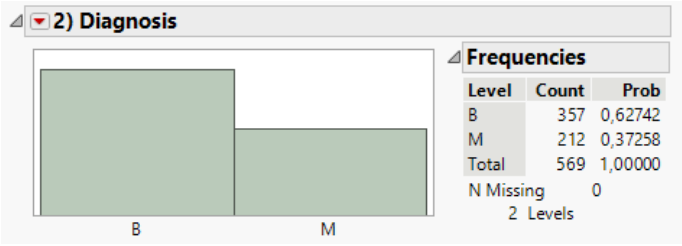
W celu przeprowadzenia procesu uczenia maszynowego, jednym z najistotniejszych kroków jakie należy podjąć jest wstępne zaznajomienie się z zestawem danych i jego analiza pod kątem rozkładu poszczególnych zmiennych oraz prawdopodobieństw. W tym celu wykorzystane zostało oprogramowanie JMP.

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%.

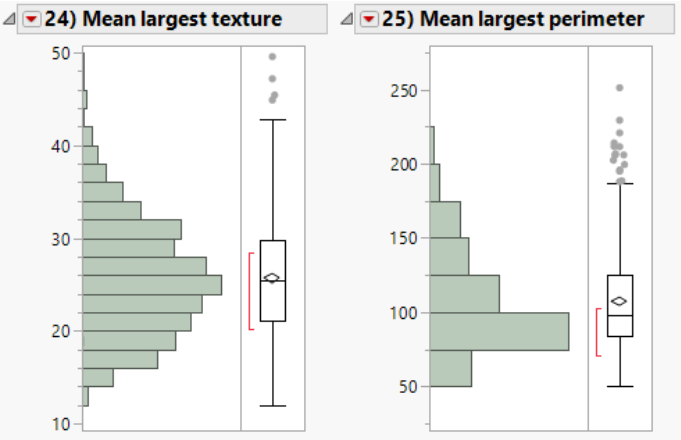
Podczas analizy histogramów zmiennych decyzyjnych, stwierdzono że znaczna ilość ma charakter prawostronnie skośny oraz występują dla nich obserwacje odstające, o czym informuje znajdujący się po prawej stronie histogramu wykres okienkowy (ang. *box graph*), co przedstawiono na rysunku 3.2. Wyjątkiem 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.



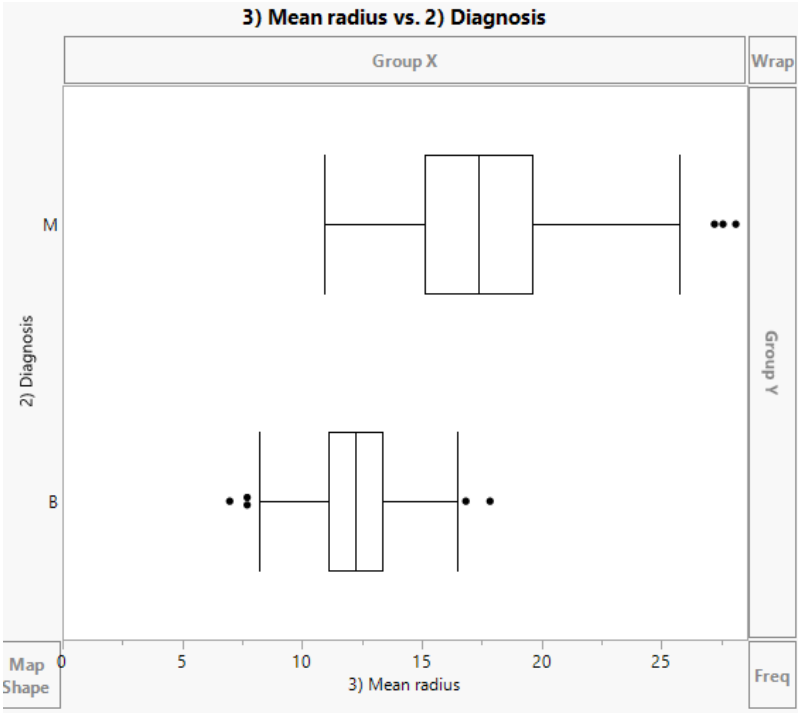
Rysunek 3.1. Histogram rozkładu zmiennej odpowiedzi

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

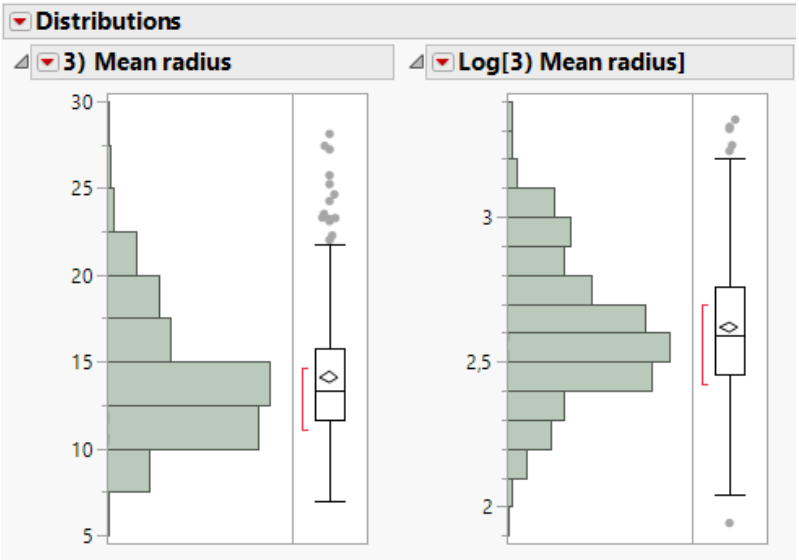


Rysunek 3.3. Przykład analizy obserwacji odstających dla poszczególnych klas zmiennej odpowiedzi

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*.

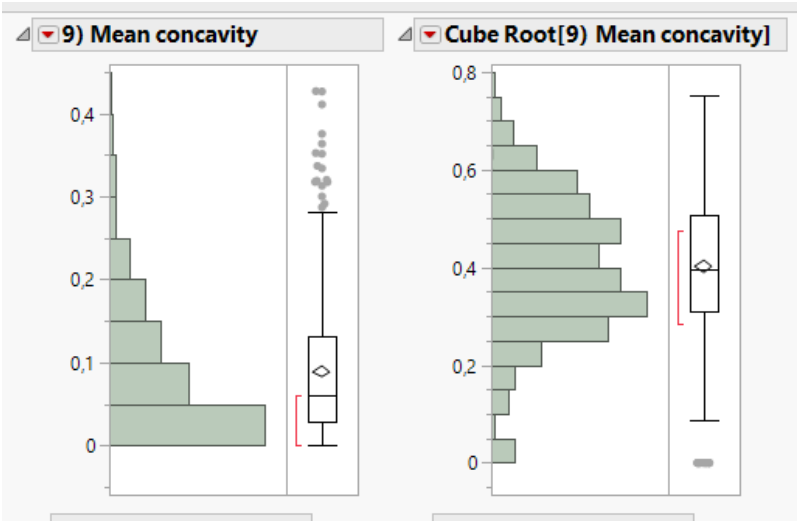


Rysunek 3.4. Porównanie rozkładu danych przed i po transformacji logarytmicznej.

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:

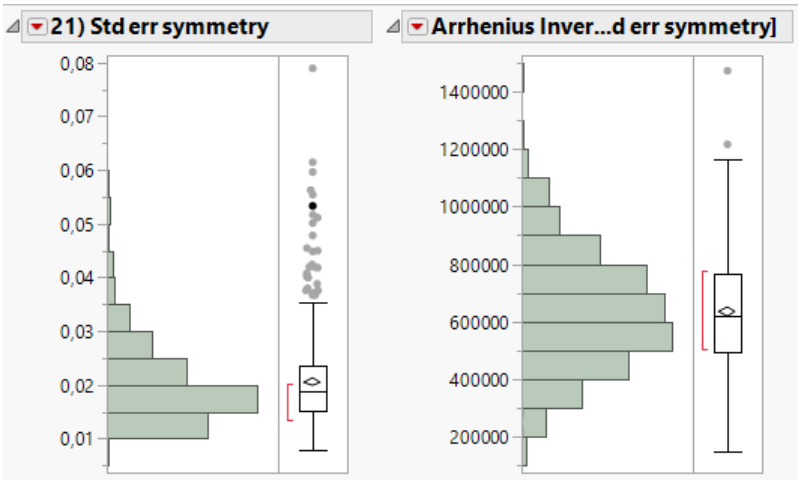
1. *Mean compactness*;
2. *Mean concavity*;
3. *Mean concave points*;
4. *Std err concavity*;
5. *Mean largest radius*;

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.

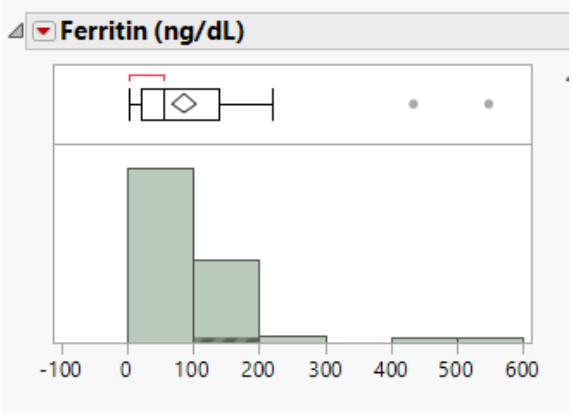
Ze względu na bardzo małą ilość obserwacji, zdecydowano się na zachowanie wszystkich obserwacji odstających, aby zapobiec utracie informacji i zmianie uzyskanych w procesie normalizacji rozkładów. W celu zachowania kompatybilności z

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

3.2.2. Dane regresyjne

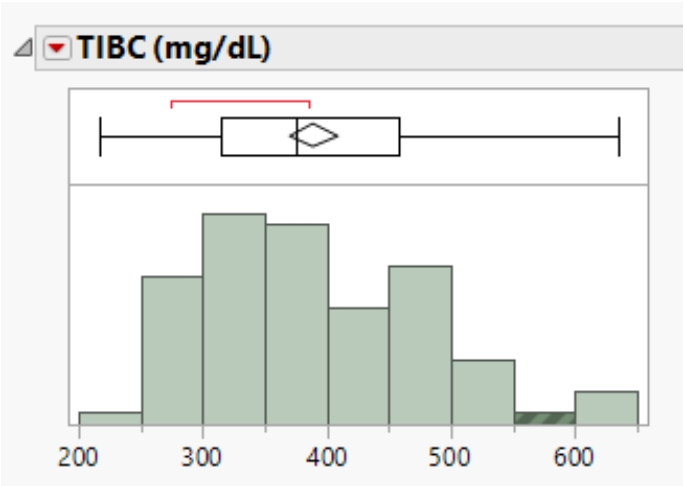
3.2.2.1. Analiza rozkładu danych

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.

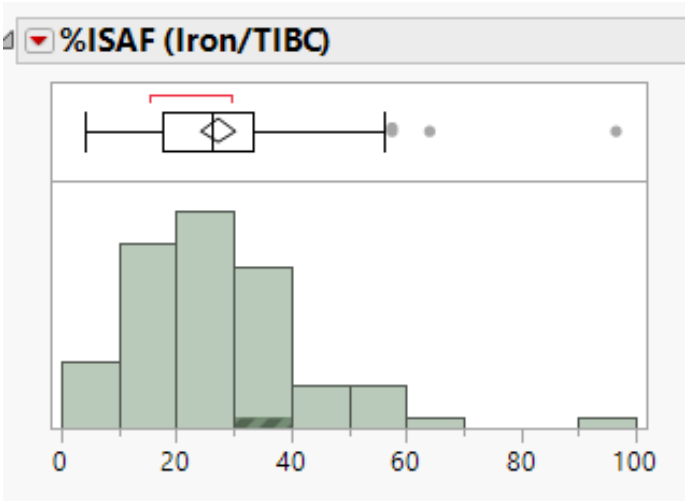


Rysunek 3.7. Wykres rozkładu zmiennej odpowiedzi dla zestawu regresyjnego

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.

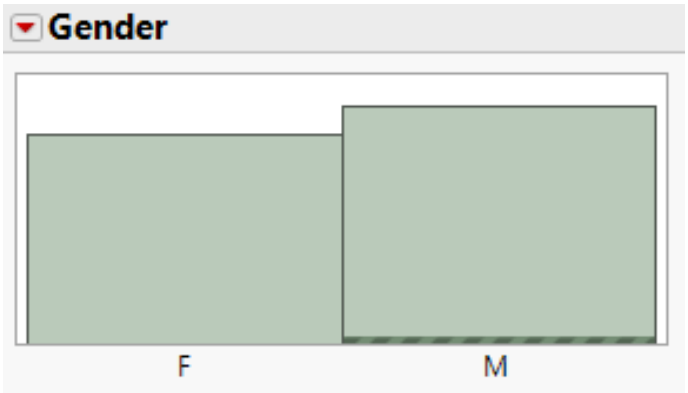


Rysunek 3.8. Przykłady rozkładu zmiennej decyzyjnej cz. 1



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:

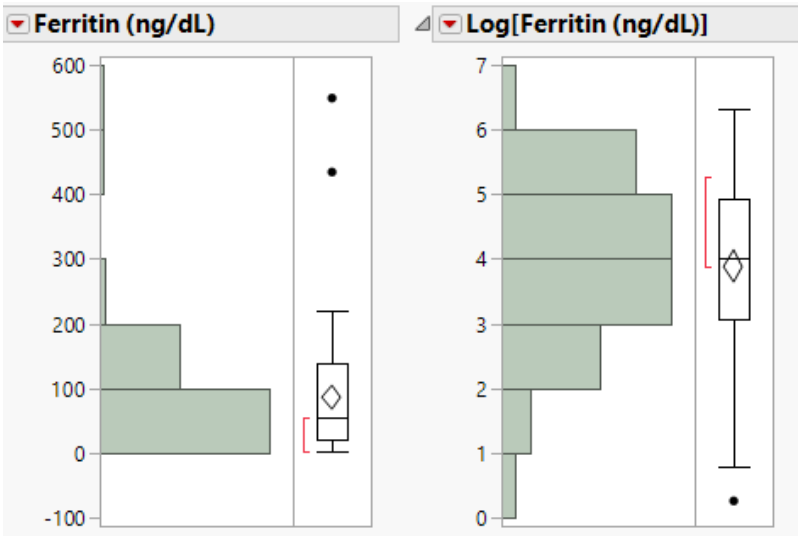
1. *Age*;
2. *Gender*;

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.

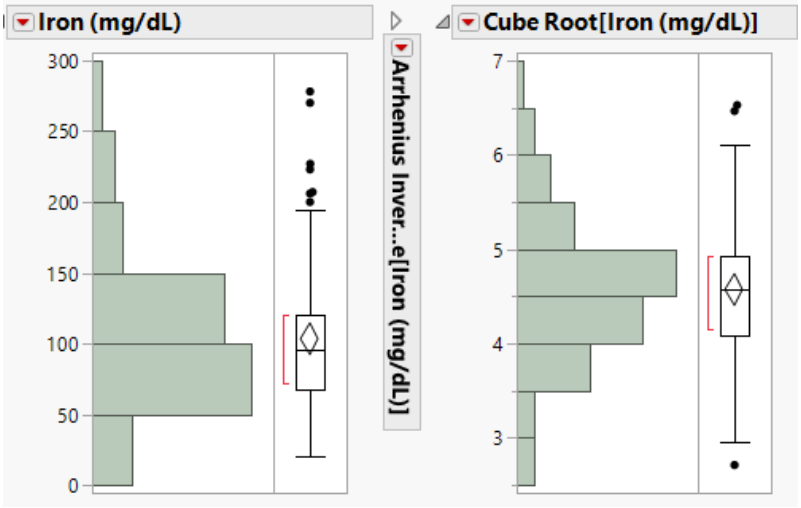


Rysunek 3.11. Wpływ transformacji logarytmicznej na rozkład zmiennej odpowiedzi

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

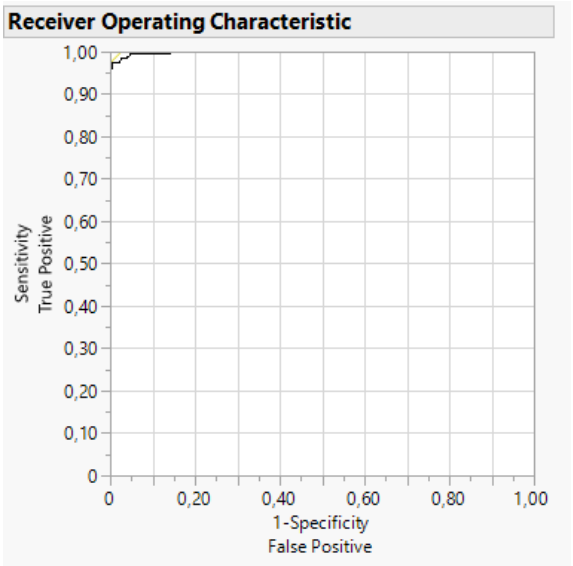
Rysunek 3.13. Wykres p-wartości dla całego zestawu zmiennych decyzyjnych.

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

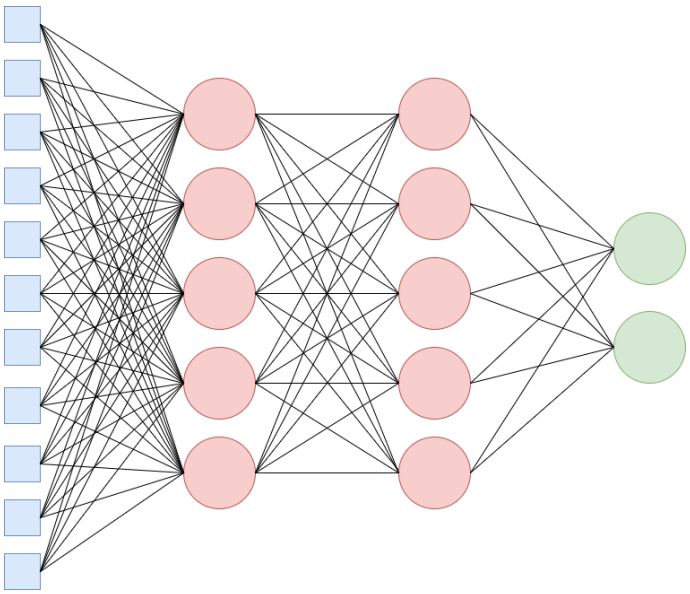
3.3.2. Głęboka sieć neuronowa

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

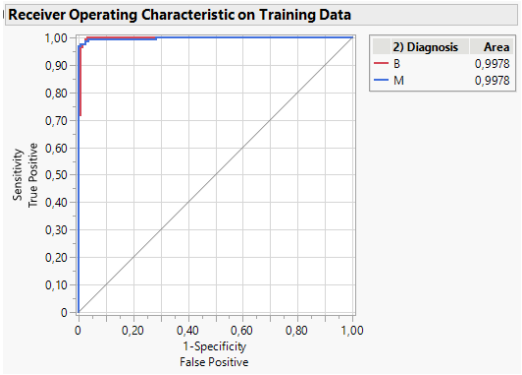
Typ warstwy	ilość neuronów	aktywacja
ukryta	5	tangens hiperboliczny
ukryta	5	tangens hiperboliczny
wyjściowa	2	—

Tabela 3.2. Struktura modelu sieci neuronowej

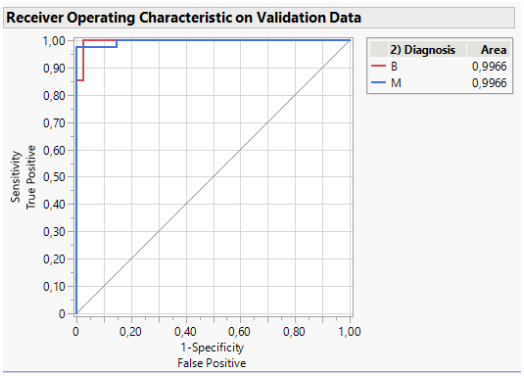


Rysunek 3.16. Schemat struktury sieci

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



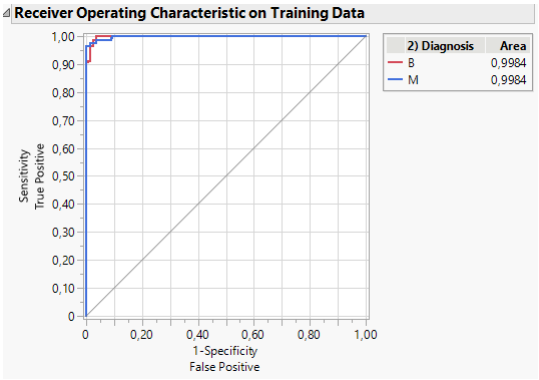
Rysunek 3.18. Krzywa charakterystyczna odbiornika dla danych walidacyjnych

3.3.3. Maszyna wektorów nośnych

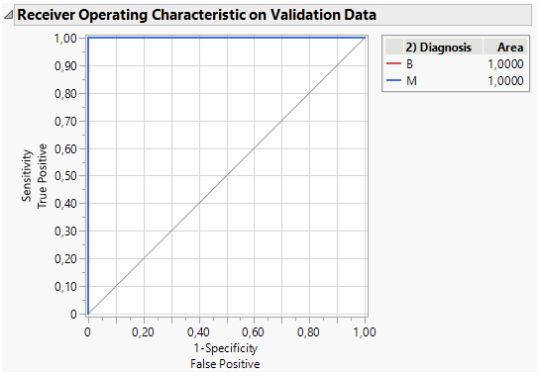
Do predykcji diagnozy wykorzystano ten sam zestaw regresorów, zawartych w tabeli 3.1. Ponownie w celu walidacji użyto metody wybrania losowego zestawu walidacyjnego spośród dostarczonych danych, w proporcji 80% obserwacji uczących i 20% testowych, z użyciem wartości 1234 dla ziarna generatora liczb pseudolosowych. 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

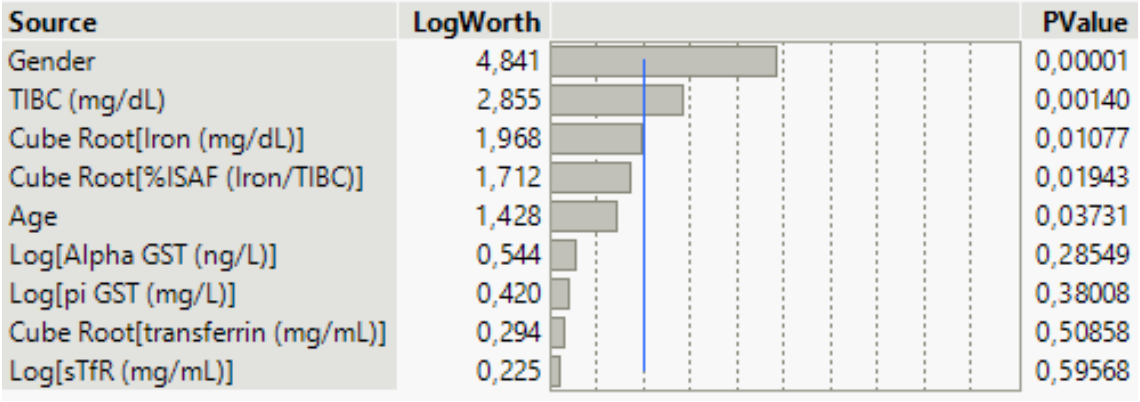


Rysunek 3.20. Krzywa charakterystyczna odbiornika dla danych walidacyjnych modelu SVM

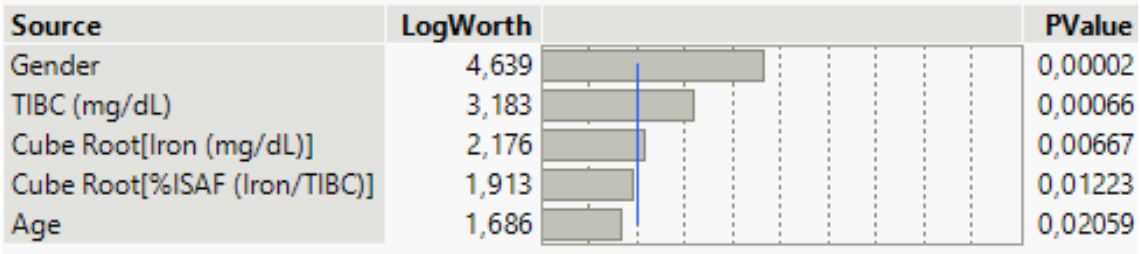
Utworzony w ten sposób model posiada generalizowaną statystykę R^2 na poziomie 0.97161 dla zestawu walidacyjnego, i uzyskał wskaźnik błędnej klasyfikacji wynoszący 0% dla danych testowych, oraz 1,3% dla danych uczących. Tabela 3.3 przedstawia wartości X dla poszczególnych regresorów. Rysunki 3.19 oraz 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 = shogun::SGMatrix<float64_t>;
25
26     Dataset ret;
27
28     // odczytanie surowej zawartości pliku csv i sparsowanie jej do macierzy
29     auto csvFile = some<CCSVFile>(filename);
30     Matrix data;
31     data.load(csvFile);
32     // transpozycja do postaci docelowej dla człowieka (działanie na kolumnach)
33     Matrix::transpose_matrix(data.matrix, data.num_rows, data.num_cols);
34     // podział macierzy na część regresorów i zmiennej odpowiedzi
35     switch(labelPos)
36     {
37         case FIRST:
```

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

Listing 4.2. Funkcja przepakowujące dane do kontenerów docelowych

```
1   #pragma once
2
3   #include <inc/shogun/csv.hpp>
4
5   inline void shogunModels()
6   {
7       using namespace shogun;
8
9       // odczytanie danych we własnym pośrednim typie danych
10      auto classificationDatasetTemp =
11          readShogunCsvData("wdbc_data_with_labels.csv", LabelPos::FIRST);
12      auto regressionDatasetTemp =
13          readShogunCsvData("IronGlutathione.csv", LabelPos::LAST);
14      // rozdzielenie danych na regresory i zmienne odpowiedzi
15      auto classificationFeatures =
16          some<CDenseFeatures<float64_t>>(
17              classificationDatasetTemp.inputs);
18      auto classificationLabels =
19          wrap(new CMulticlassLabels(
20              classificationDatasetTemp.outputs.get_column(0)));
21      auto regressionFeatures =
22          some<CDesneFeatures<float64_t>>(
23              regressionDatasetTemp.inputs);
24      auto regressionLabels =
25          some<CRegressionLabels>(
26              regressionDatasetTemp.outputs);
27
28  }
```

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]

```
#pragma once

inline void KernelPCA(shogun::Some<CDenseFeatures<float64_t>> inputs,
                      const int target_dim)
{
    using namespace shogun;

    // utworzenie jądra
    auto gaussKernel = some<CGaussianKernel>(inputs, inputs, 0.5);
    // utworzenie obiektu reduktora wymiarowości
    auto pca = some<CKernelPCA>();
    // konfiguracja reduktora
    pca->set_kernel(gaussKernel.get());
    pca->set_target_dim(target_dim);
    // nauczenie reduktora
    pca->fit(inputs);
    // zastosowanie redukcji
    auto featureMatrix = inputs->get_feature_matrix();
    for (index_t i = 0; i < inputs->get_num_vectors(); ++i)
    {
        auto vector = featureMatrix.get_column(i);
        auto newVector = pca->apply_to_feature_vector(vector);
    }
}
```

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 Shogun [5]

```
using namespace shogun;
// [...]
// utworzenie zestawu danych
auto x = some<CDenseFeatures<float64_t>>(x_values);
auto y = some<CRegressionLabels>(y_values);
// utworzenie modelu
float64_t tau_regularization = 0.0001;
auto lr = some<ClinearRidgeRegression>(tau_regularization, nullptr, nullptr);
// konfiguracja i trening modelu
lr->set_labels(y);
lr->train(x);
// wykonanie predykcji dla nowych danych
auto new_x = some<CDenseFeatures<float64_t>>(new_x_values);
auto y_predict = lr->apply_regression(new_x);
// odczytanie wag
auto weights = lr->get_w();
// wyliczenie wartości funkcji straty
y_predict = lr->apply_regression(x);
auto eval = some<CMeanSquaredError>();
auto mse = eval->evaluate(y_predict, y);
```

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 opisany w książce „Hands On Machine Learning" [5], wykorzystując dodatkowo mechanizm sprawdzianu krzyżowego do wyboru hiperparametrów, który opisany jest w dalszej części pracy.

Listing 4.6. Przykład regresji logistycznej z użyciem sprawdzianu krzyżowego [5]

```
using namespace shogun;
// [...]
// dane uczące i walidacyjne
Some<CDenseFeatures<DataType>> features;
Some<CMulticlassLabels> labels;
Some<CDenseFeatures<DataType>> test_features;
Some<CMulticlassLabels> test_labels;

// 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 dla drzewa decyzyjnego
index_t k = 3;
CStatifiedCrossValidationSplitting* splitting =
    new CStatifiedCrossValidationSplitting(labels, k);

// utworzenie kryterium ewaluacji dla drzewa decyzyjnego parametrów
auto eval_criterium = some<CMulticlassAccuracy>();
```

```
23 // utworzenie modelu regresji logistycznej
24 auto log_reg = some<CMulticlassLogisticRegression>();
25 // utworzenie obiektu sprawdzianu krzyżowego
26 auto cross = some<CCrossValidation>(log_reg, features, labels,
27                                     splitting, eval_criterium);
28 cross->set_num_runs(1);
29
30 auto model_selection = some<CGridSearchModelSelection>(cross, root);
31 // wybranie parametrów dla modelu
32 CParameterCombination* best_params =
33     wrap(model_selection->select_model(false));
34 // zaaplikowanie parametrów dla modelu
35 best_params->apply_to_machine(log_reg);
36 // wyświetlenie drzewa decyzyjnego
37 best_params->print_tree();
38
39 // trenowanie
40 log_reg->set_labels(labels);
41 log_reg->train(features);
42
43 // ewaluacja modelu dla danych testowych
44 auto new_labels = wrap(log_reg->apply_multiclass(test_features));
45
46 // ocena dokładności
47 auto accuracy = eval_criterium->evaluate(new_labels, test_labels);
48
49 // przetworzenie wyników
50 auto feature_matrix = test_features->get_feature_matrix();
51 for (index_t i = 0; i < new_labels->get_num_labels(); ++i)
52 {
53     auto label_idx_pred = new_labels->get_label(i);
54     auto vector = feature_matrix.get_column(i);
55     // [...]
56 }
```

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 wraz z doborem parametrów [5].

Listing 4.7. Przykład użycia maszyny wektorów nośnych [5]

```
1  using namespace shogun;
2  // [...]
3  // dane uczące i walidacyjne
4  Some<CDenseFeatures<DataType>> features;
5  Some<CMulticlassLabels> labels;
6  Some<CDenseFeatures<DataType>> test_features;
7  Some<CMulticlassLabels> test_labels;
8
9  // utworzenie jądra
10 auto kernel = some<CGaussianKernel>(features, features, 5);
11 // utworzenie wieloklasowej maszyny wektorów nośnych opartej
```

```
12 // o klasyfikację one-versus-one
13 auto svm = some<CMulticlassLibSVM>();
14 svm->set_kernel(kernel);
15
16 // utworzenie drzewa decyzyjnego dla parametrów
17 auto root = some<CModelSelectionParameters>();
18 // parametr określający stopień unikania missklasyfikacji
19 CModelSelectionParameters* c = new CModelSelectionParameters("C");
20 root->append_child(c);
21 c->build_values(1.0, 1000.0, R_LINEAR, 100.);
22 // dodanie jądra wyboru parametrów
23 auto params_kernel = some<CModelSelectionParameters>("kernel", kernel);
24 root->append_child(params_kernel);
25 // konfiuracja jądra
26 auto params_kernel_width =
27     some<CmodelSelectionParameters>("combined_kernel_weight");
28 params_kernel_width->build_values(0.1, 10.0, R_LINEAR, 0.5);
29 params_kernel->append_child(params_kernel_width);
30
31 // przygotowanie strategii podziału dla drzewa decyzyjnego
32 index_t k = 3;
33 CStatifiedCrossValidationSplitting* splitting =
34     new CStatifiedCrossValidationSplitting(labels, k);
35
36 // przygotowanie kryterium ewaluacji modelu
37 auto eval_criterium = some<CMulticlassAccuracy>();
38
39 // przygotowanie obiektu sprawdzianu krzyżowego
40 auto cross =
41     some<CCrossValidation>(svm, features, labels, splitting, eval_criterium);
42 cross->set_num_runs(1);
43
44 // wybór i zaaplikowanie parametrów dla modelu
45 auto model_selection = some<CGridSearchModelSelection>(cross, root);
46 CParameterCombination* best_params =
47     wrap(model_selection->select_model(false));
48 best_params->apply_to_machine(svm);
49 best_params->print_tree();
50
51 // trening
52 svm->set_labels(labels);
53 svm->train(features);
54
55 // ewaluacja modelu
56 auto new_labels = wrap(svm->apply_multiclass(test_features));
57 // obliczenie dokładności
58 auto accuracy = eval_criterium->evaluate(new_labels, test_labels);
59 std::cout << "svm␣" << name << "␣accuracy␣=␣" << accuracy << std::endl;
60
61 // przetworzenie wyników
62 auto feature_matrix = test_features->get_feature_matrix();
63 for (index_t i = 0; i < new_labels->get_num_labels(); ++i)
64 {
65     auto label_idx_pred = new_labels->get_label(i);
66     auto vector = feature_matrix.get_column(i);
67     // [...]
68 }
```

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 [5]

```
using namespace shogun;
// [...]
void KNNClassification(Some<CDenseFeatures<DataType>> features,
                       Some<CmulticlassLabels> labels,
                       Some<CDenseFeatures<DataType>> test_features,
                       Some<CmulticlassLabels> test_labels)
{
    // przygotowanie modelu
    std::int32_t k = 3;
    auto distance = some<CEuclideanDistance>(features, features);
    auto knn = some<CKNN>(k, distance, labels);

    // wygenerowanie predykcji
    auto new_labels = wrap(knn->apply_multiclass(test_features));

    // obliczenie dokładności
    auto eval_criterium = some<CMulticlassAccuracy>();
    auto accuracy = eval_criterium->evaluate(new_labels, test_labels);

    // przetwarzanie wyników
    auto feature_matrix = test_features->get_feature_matrix();
    for (index_t i = 0; i < new_labels->get_num_labels(); ++i)
    {
        auto label_idx_pred = new_labels->get_label(i);
        // [...]
    }
}
```

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 [5]

```
using namespace shogun;
// [...]
void GBMClassification(Some<CDenseFeatures<DataType>> features,
                       Some<CRegressionLabels> labels,
                       Some<CDesneFeatures<DataType>> test_features,
                       Some<CRegressionLabels> test_labels)
{
    // oznaczenie regresorów jako ciągłe
    SGVector<bool> feature_type(1);
    feature_type.set_const(false);

    // utworzenie bazowego drzewa decyzyjnego
    auto tree = some<CCARTree>(feature_type, PT_REGRESSION);
    tree->set_max_depth(3);
    // utworzenie funkcji straty
    auto loss = some<CSquaredLoss>();
    constexpr int iterations = 100;
    constexpr int learning_rate = 0.1;
    constexpr int sub_set_fraction = 1.0;
    auto model = some<CStochasticGBMachine>(tree,
                                            loss,
                                            iterations,
                                            learning_rate,
                                            sub_set_fraction);
    // konfiguracja modelu
    model->set_labels(labels);
    model->train(features);

    // ewaluacja modelu
    auto new_labels = wrap(model->apply_regression(test_features));
    auto eval_criterium = some<CMeanSquaredError>();
    auto accuracy = eval_criterium->evaluate(new_labels, test_labels);
}
```

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 [5]

```
using namespace shogun;
// [...]
void RFClassification(Some<CDenseFeatures<DataType>> features,
                      Some<CRegressionLabels> labels,
                      Some<CDenseFeatures<DataType>> test_features,
                      Some<CRegressionLabels> test_labels)
{
    std::int32_t num_rand_feats = 1;
    std::int32_t num_bags = 10;

    // utworzenie i konfiguracja modelu
    auto rand_forest =
        some<CRandomForest>(num_rand_feats, num_bags);
    auto vote =some<CMajorityVote>();
    rand_forest->set_combination_rule(vote);
    // oznaczenie danych jako ciągłe
    SGVector<bool> feature_type(1);
    feature_type.set_const(false);
    rand_forest->set_feature_types(feature_type);

    // trening
    rand_forest->set_labels(labels);
    rand_forest->set_machine_problem_type(PT_REGRESSION);
    rand_forest->train(features);

    // ewaluacja modelu
    auto new_labels = wrap(rand_forest->apply_regression(test_features));
    auto eval_criterium = some<CMeanSquaredError>();
    auto accuracy = eval_criterium->evaluate(new_labels, test_labels);
}
```

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.

Listing 4.11. Przykład użycia sieci neurnowej [5]

```
1  using namespace shogun;
2  // [...]
3  // przygotowanie danych wejściowych
4  std::size_t n = 10000;
5  SGMatrix<float64_t> x_values(1, static_cast<index_t>(n));
6  SGVector<float64_t> y_values(static_cast<index_t>(n));
7  // [...]
8  auto x = some<CDenseFeatures<float64_t>>(x_values);
9  auto y = some<CRegressionLabels>(y_values);
10
11 // konstrukcja architektury sieci
12 auto dimensions = x->get_num_features();
13 auto layers = some<CNeuralLayers>();
14 layers = wrap(layers->input(dimentions));
15 layers = wrap(layers->rectified_linear(32));
16 layers = wrap(layers->rectified_linear(16));
17 layers = wrap(layers->rectified_linear(8));
18 layers = wrap(layers->linear(1));
19 auto all_layers = layers->done();
20
21 // utworzenie sieci
22 auto network = some<CNeuralNetwork>(all_layers);
23 network->quick_connect();
24 network->initialize_neural_network();
25
26 // konfiguracja sieci
27 network->set_optimization_method(NNOM_GRADIENT_DESCENT);
```

```
network->set_gd_mini_batch_size(64);
network->set_l2_coefficient(0.0001);
network->set_max_num_epochs(500);
network->set_epsilon(0.0); // kryterium zbieżności
network->set_gd_learning_rate(0.01);
network->set_gd_momentum(0.5);

// dodatkowe ustawienie bardziej szczegółowych
// logów z procesu uczenia
shogun::sg_io->set_log_level(shogun::MSG_DEBUG);

// trening
network->set_labels(y);
network->train(x);
```

4.4.7. Brzegowa regresja jądra ?

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

4.5.3. Logarytmiczna funkcja straty

4.5.4. Metryka R^2

4.5.5. Metryka adjusted R^2

4.5.6. Dokładność

4.5.7. Precyzja i pamięć (recall)

4.5.8. Metryka F-score

4.5.9. Metryki AUC i ROC

4.5.10. Sprawdzian krzyżowy K-krotny

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 znalezienie wielu 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. Ponadto, 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.

4.7. Przykłady testowe

OPISAĆ PREPROCESSING CSV !!!!

4.7.1. Regresja logistyczna

4.7.2. Maszyna wektorów nośnych

4.7.3. Sieć neuronowa

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* 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, co zostało przedstawione na listingu 5.1. W aktualnej wersji biblioteki znalazły się także wbudowane funkcje pobierania danych współpracujące z protokołem HTTP.

Listing 5.1. Pobranie danych do uczenia z serwera HTTP [7]

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

    shark::ClassificationDataset trainData;
    shark::csvStringToData(trainData, trainDataString, shark::FIRST_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 5.1 pokazuje przykład wstępnego przetwarzania danych z wykorzystaniem normalizacji.

Listing 5.3. Wstępne przetwarzanie danych do uczenia [7]

```
// [...]

// przemieszanie danych i wyznaczenie danych testowych
train_data.shuffle();
auto test_data = shark::splitAtElement(train_data, 120);

// utworzenie normalizera
using Trainer = shark::NormalizeComponentsUnitVariance<shark::RealVector>;
bool remove_mean = true;
shark::Normalizer<shark::RealVector> normalizer;
Trainer normalizing_trainer(remove_mean);

// nauczenie normalizera średniej i wariancji danych treningowych
normalizing_trainer.train(normalizer, train_data.inputs());

// transformacja danych uczących
train_data = shark::transformInputs(train_data, 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 5.6 przedstawia sposób użycia LDA do zadania klasyfikacji.

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

```
1  using namespace shark;
2
3  // [...]
4
5  void LDAReduction(const UnlabeledData<RealVector>& data,
6                    const UnlabeledData<RealVector>& labels,
7                    std::size_t target_dim)
8  {
9      // utworzenie obiektów LDA
10     LinearClassifier<> encoder;
11     LDA lda;
12
13     // utworzenie zestawu danych
14     LabeledData<RealVector, unsigned int> dataset(
15         labels.numberOfElements(), InputLabelPair<RealVector, unsigned int>(
16             RealVector(data.element(0).size()), 0));
17
18     // wypełnienie zbioru danymi
19     for (std::size_t i = 0; i < labels.numberOfElements(); ++i)
20     {
21         // zmiana indeksów klas aby zaczynały się od 0
22         dataset.element(i).label =
23             static_cast<unsigned int>(labels.element(i)[0]) - 1;
24         dataset.element[i].input = data.element(i)
25     }
26
27     // trening enkodera
28     lda.train(encoder, dataset);
29
30     // utworzenie zredukowanego zestawu danych
31     auto new_labels = encoder(data);
32     auto new_data = encoder.decisionFunction()(data);
33 }
```

Listing 5.6. Przykład klasyfikacji z wykorzystaniem modelu LDA [8]

```
1  #include <shark/Algorithms/Trainers/LDA.h>
2  // [...]
3
4  using namespace shark;
5
6  int main(int argc, char **argv)
```

```
7  {
8      // import danych
9      ClassificationDataset data;
10     try
11     {
12         importCSV(data, argv[1], LAST_COLUMN, ' ');
13     }
14     catch(...)
15     {
16         std::cerr << "Unable to read data from file " << argv[1] << std::endl;
17         exit(EXIT_FAILURE);
18     }
19
20     // wyświetlenie informacji o danych
21     std::cout << "overall number of data points: " << data.numberOfElements()
22               << " number of classes: " << numberOfClasses(data)
23               << " input dimension: " << inputDimenstion(Data) << std::endl;
24
25     // wyodrębnienie danych testowych
26     auto test = splitAtElement(data, .5 * data.numberOfElements());
27     // utworzenie i wytrenowanie modelu
28     LDA ldaTrainer;
29     LinearClassifier<> lda;
30     ldaTrainer.train(lda, data);
31
32     // analiza predykcji i dokładności modelu
33     Data<unsigned int> prediction;
34     ZeroOneLoss<unsigned int> loss;
35     prediction = lda(data.inputs());
36     std::cout << "LDA on training set accuracy: "
37         << 1. - loss(data.labels(), prediction) << std::endl;
38     prediction = lda(test.inputs());
39     std::cout << "LDA on test set accuracy: "
40         << 1. - loss(test.labels(), prediction) << std::endl;
41
42     return 0;
43 }
```

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 5.7 ukazane zostało wykorzystanie podejścia iteracyjnego, natomiast listing 5.8 przedstawia metodę analityczną.

Listing 5.7. Przykład regresji liniowej z wykorzystaniem optymalizatora spadku gradientowego [9]

```
#include <shark/Data/Csv.h>
#include <shark/Algorithms/GradientDescent/CG.h>
#include <shark/ObjectiveFunctions/ErrorFunction.h>
#include <shark/ObjectiveFunctions/Loss/SquaredLoss.h>
#include <shark/Models/LinearModel.h>

using namespace shark;

RegressionDataset loadData(const std::string& dataFile,
                           const std::string& labelsFile)
{
    Data<RealVector> inputs;
    Data<RealVector> labels;

    try
    {
        importCSV(inputs, regressorsFile, ' ');
        importCSV(labels, targetFile, ' ');
    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << '\n';
        exit(EXIT_FAILURE);
    }
    RegressionDataset data(inputs, labels);
    return data;
}
// [...]

// odczytanie danych z plików oraz podział na dane uczące i testowe
RegressionDataset data = loadData("data/regressionInputs.csv",
                                  "data/regressionLabels.csv");
RegressionDataset test = splitAtElement(data, static_cast<std::size_t>(
                                            0.8*data.numberOfElements()));
```

```
35 // przygotowanie modelu
36 LinearModel<> model(inputDimension(data), labelDimension(data));
37 SquaredLoss<> loss;
38 ErrorFunction errorFunction(data, &model, &loss);
39
40 // przygotowanie i wyszkolenie optymizatora
41 CG optimizer;
42 errorFunction.init();
43 optimizer.init(errorFunction);
44 for (int i = 0; i < 100; ++i)
45 {
46     optimizer.step(errorFunction);
47 }
48
49 // przeliczenie predykcji modelu dla danych testowych
50 model.setParameterVector(optimizer.solution().point);
51 Data<RealVector> predictions = model(test.inputs());
52 double testError = loss.eval(test.labels(), predictions);
```

Listing 5.8. Przykład regresji liniowej z wykorzystaniem trenera analitycznego [5]

```
1  // [...]
2
3  using namespace shark;
4
5  // [...]
6
7  LinearRegression trainer;
8  LinearModel<> model;
9  trainer.train(model, data);
10
11 std::cout << "intercept:␣" << model.offset() << std::endl;
12 std::cout << "matrix:␣" << model.matrix() << std::endl;
13
14 auto prediction = model(test);
15 SquaredLoss<> loss;
16 auto se = loss(test.labels(), prediction);
```

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 5.9 prezentuje funkcję budującą model logistycznej regresji wieloklasowej, natomiast listing 5.10 prezentuje sposób utworzenia prostego modelu dla problemu binarnego.

Listing 5.9. Przykład funkcji tworzącej model wieloklasowej regresji logistycznej [5]

```
using namespace shark;

// [...]

void LRClassification(const ClassificationDataset& train,
                      const ClassificationDataset& test,
                      unsigned int num_classes)
{
    // utworzenie obiektu docelowego klasyfikatora oraz tablicy
    // klasyfikatorów składowych
    OneVersusOneClassifier<RealVector> ovo;
    auto pairs = num_classes * (num_classes - 1) / 2;
    std::vector<LinearClassifier<RealVector>> lr(pairs);

    // iteracyjne konfigurowanie klasyfikatorów składowych
    for (std::size_t n = 0, cls1=1; cls1 < num_classes; ++cls1)
    {
        using BinaryClassifierType =
            OneVersusOneClassifier<RealVector>::binary_classifier_type;
        std::vector<BinaryClassifierType*> ovo_classifiers;
        for (std::size_t cls2 = 0; cls2 < cls1; ++cls2, ++n)
        {
            // pobranie binarnego podproblemu
            ClassificationDataset binary_cls_data =
                binarySubProblem(train, cls2, cls1);

            // trening modelu składowego
            LogisticRegression<RealVector> trainer;
            trainer.train(lr[n], binary_cls_data);

            // załadowanie modelu składowego do serii
            ovo_classifiers.push_back(&lr[n]);
        }
        // podłączenie serii do głównego klasyfikatora
        ovo.addClass(ovo_classifiers);
    }

    // użycie modelu
    auto predictions = ovo(test.inputs());
    // [...]
}
```

Listing 5.10. Przykład prostej binarnej regresji logistycznej

```
using namespace shark;

// [...]
void SimpleLR(const ClassificationDataset& train,
              const ClassificationDataset& test)
```

```
{
    // utworzenie modelu oraz trenera
    LinearClassifier<RealVector> model;
    LogisticRegression<RealVector> trainer;

    // trenowanie modelu
    trainer.train(model, train);

    // wykorzystanie modelu
    auto predictions = model(test.inputs());
    // [...]
}
```

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 5.11 ukazuje przykład utworzenia i skonfigurowania modelu na podstawie wpisów dostępnych w dokumentacji biblioteki, natomiast listing 5.11 przedstawia sposób utworzenia maszyny wektorów nośnych dla problemów wieloklasowych wewnątrz funkcji przyjmującej zestawy danych uczących i testowych.

Listing 5.11. Przykład maszyny wektorów nośnych dla problemu binarnego [10]

```
#include <shark/Algorithms/Trainers/CSvmTrainer.h>
#include <shark/Models/Kernels/GaussianRbfKernel.h>

// umożliwia wygenerowanie przykładowego zestawu danych
#include <shark/Data/DataDistribution.h>

using namespace shark;

// ...

// wygenerowanie przykładowego zestawu danych
unsigned int trainingDataPoints = 500;
unsigned int testDataPoints = 10000;
Chessboard problem;
ClassificationDataset training = problem.generateDataset(trainingDataPoints);
ClassificationDataset test = problem.generateDataset(testDataPoints);

// przygotowanie kernelu
double gamma = 0.5;     // przepustowość kernelu
GaussianRbfKernel<> kernel(gamma);
KernelClassifier<RealVector> kc; // liniowa funkcja dla przestrzeni kernelu

// przygotowanie klasy trenera
double regularization = 1000.0;
bool bias = true;
// drugi parametr szablonu określa wykorzystanie typu double dla pamięci
// cache modelu zamiast float
CSvmTrainer<RealVector, double> trainer(&kernel, regularization, bias);

// konfiguracja modelu
trainer.sparsify() = false; // zachowanie wektorów nie-nośnych
trainer.stopCondition().minAccuracy = 1e-6;
trainer.setCacheSize(0x1000000);


// trenowanie modelu
trainer.train(kc, training);

// wyświetlenie informacji diagnostycznych o uczeniu
std::cout << "Needed␣" << trainer.solutionProperties().seconds
          << "␣seconds␣to␣reach␣a␣dual␣of␣"
          << trainer.solutionProperties().value << std::endl;
```

```
43
44 // użycie modelu
45 auto predictions = kc(test.inputs());
```

Listing 5.12. Przykład maszyny wektorów nośnych dla problemu wieloklasowego [5]

```
1  using namespace shark;
2
3  // ...
4
5  void SVMClassification(const ClassificationDataset& train,
6                         const ClassificationDataset& test,
7                         unsigned int num_classes)
8  {
9      double gamma = 0.5;
10     GaussianRbfKernel<> kernel(gamma);
11
12     // utworzenie obiektu modelu docelowego
13     OneVersusOneClassifier<RealVector> ovo;
14
15     // utworzenie kontenera na poszczególne podproblemy
16     unsigned int pairs = num_classes * (num_classes - 1) / 2;
17     std::vector<KernelClassifier<RealVector>> svm(pairs);
18
19     for (std::size_t n = 0, cls1 = 1; cls1 < num_classes; cls1++)
20     {
21         // utworzenie zestawu klasyfikatorów podproblemów dla danej klasy
22         using BinaryClassifierType =
23             OneVersusOneClassifier<RealVector>::binary_classifier_type;
24         std::vector<BinaryClassifierType*> ovo_classifiers;
25
26         for (std::size_t cls2 = 0; cls2 < cls1; cls2++, n++)
27         {
28             // utworzenie podproblemu binarnego
29             ClassificationDataset binary_cls_data =
30                 binarySubProblem(train, cls2, cls1);
31
32             // trenowanie modelu składowego
33             double c = 10.0;
34             CSvmTrainer<RealVector> trainer(&kernel, c, false);
35             trainer.train(svm[n], binary_cls_data);
36             ovo_classifiers.push_back(&svm[n]);
37         }
38         // dołożenie zestawu klasyfikatorów do głównego modelu
39         ovo.addClass(ovo_classifiers);
40     }
41
42     // użycie modelu
43     auto predictions = ovo(test.inputs());
44 }
```

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 5.13 przedstawia sposób przygotowania klasyfikatora kNN.

Listing 5.13. Przykład utworzenia klasyfikatora kNN [11]

```
#include <shark/Data/Csv.h>
#include <shark/Models/NearestNeighborModel.h>
#include <shark/Algorithms/NearestNeighbors/TreeNearestNeighbors.h>
#include <shark/Models/Trees/KDTree.h>

using namespace shark;

int main()
{
    std::string filename = "sample_data_file.csv"

    // odczytanie danych z pliku
    ClassificationDataset data;
    try
    {
        importCSV(data, filename, LAST_COLUMN, ' ');
    }
    catch (...)
    {
        std::cerr << "unable to read data from file " <<  filename
                  << std::endl;
        exit(EXIT_FAILURE);
    }

    // wyświetlenie informacji o danych
    std::cout << "number of data points: " << data.numberOfElements()
              << " number of classes: " << numberOfClasses(data)
              << " input dimension: " << inputDimension(data)
              << std::endl;

    // wydzielenie zestawu danych testowych
    ClassificationDataset dataTest = splitAtElement(
        data,
        static_cast<std::size_t>(
            .5 * data.numberOfElements())
        );
    // wyświetlenie informacji
    std::cout << "training data points: " << data.numberOfElements()
              << std::endl;
    std::cout << "test data points: " << dataTest.numberOfElements()
              << std::endl;

    // utworzenie i konfiguracja drzewa oraz algorytmu
    KDTree<RealVector> tree(data.inputs());
    TreeNearestNeighbors<RealVector,unsigned int> algorithm(data, &tree);

```

```
47      // konfiguracja modelu
48      const unsigned int K = 1; // ilość sąsiadów dla algorytmu kNN
49      NearestNeighborModel<RealVector, unsigned int> KNN(&algorithm, K);
50
51      // przykład użycia modelu
52      ZeroOneLoss<unsigned int> loss;
53      auto prediction = KNN(data.inputs());
54      std::cout << K << "-KNN␣on␣training␣set␣accuracy:␣"
55                << 1. - loss.eval(data.labels(), prediction) << std::endl;
56      prediction = KNN(dataTest.inputs());
57      std::cout << K << "-KNN␣on␣test␣set␣accuracy:␣"
58                << 1. - loss.eval(dataTest.labels(), prediction)
59                << std::endl;
60
61      return 0;
62  }
```

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 5.14 przedstawia sposób utworzenia i użycia modelu losowego lasu.

Listing 5.14. Utworzenie modelu algorytmu złożonego losowego lasu [5]

```
1  using namespace std;
2
3  // [...]
4
5  void RFClassification(const ClassificationDataset& train,
6                        const ClassificationDataset& test)
7  {
8      RFTrainer<unsigned int> trainer;
9      trainer.setNTrees(100);
10     trainer.setMinSplit(10);
11     trainer.setMaxDepth(10);
12
13
14     // DOKOŃCZYĆ !!!!
15 }
```

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 [12]. 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 5.15 przedstawia kod realizujący cały proces, stanowiący przykład z książki „Hands On Machine Learning with C++”.

Listing 5.15. Przykład trójwarstwowej sieci neuronowej [5]

```
1  using namespace shark;
2
3  // [...]
4
5  // utworzenie zestawu danych
6  size_t n = 10000;
7  std::vector<RealVector> x_data[n];
8  std::vector<RealVector> y_data[n];
9  Data<RealVector> x = createDataFromRange(x_data);
10 Data<RealVector> y = createDataFromRange(y_data);
11 RegressionDataset train_data(x, y);
12
13 // zdefiniowanie warstw sieci
14 using DenseLayer = LinearModel<RealVector, TanhNeuron>;
15 DenseLayer layer1(1, 32, true);
16 DenseLayer layer2(32, 16, true);
17 DenseLayer layer3(16, 8, true);
18 LinearModel<RealVector> output(8, 1, true);
19 // połączenie warstw
20 auto network = layer1 >> layer2 >> layer3 >> output;
21
22 // utworzenie i konfiguracja funkcji straty
23 SquaredLoss<> loss;
24 ErrorFunction<> error(train_data, &network, &loss, true);
25 TwoNormRegularizer<> regularizer(error.numberOfVariables());
26 double weight_decay = 0.0001;
```

```
27 error.setRegularizer(weight_decay, &regularizer);
28 error.init();
29
30 // inicjalizacja wag sieci
31 initRandomNormal(network, 0.001);
32
33 // utworzenie i konfiguracja optymalizatora
34 SteepestDescent<> optimizer;
35 optimizer.setMomentum(0.5);
36 optimizer.setLearningRate(0.01);
37 optimizer.init(error);
38
39 // przeprowadzenie procesu uczenia
40 std::size_t epochs = 1000;
41 std::size_t iterations = train_data.numberOfBatches();
42 // pętla przechodząca przez kolejne epoki
43 for (std::size_t epoch = 0; epoch != epochs; ++epoch)
44 {
45     double avg_loss = 0.0;
46     // pętla operująca na pojedynczych batch'ach
47     for (std::size_t i = 0; i != iterations; ++i)
48     {
49         // wykonanie kroku optymalizatora
50         optimizer.step(error);
51         // zapisanie częściowej wartości średniej funkcji straty
52         if (i % 100 == 0)
53         {
54             avg_loss += optimizer.solution().value;
55         }
56     }
57     // obliczenie średniej wartości funkcji straty
58     avg_loss /= iterations;
59     std::cout << "Epoch " << epoch << " | Avg. Loss " << avg_loss << std::endl;
60 }
61 // konfiguracja modelu do docelowego użycia
62 network.setParameterVector(optimizer.solution().point);
```

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 [13]:

- **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 5.16 przedstawia omówiony sposób użycia na przykładzie błędu średniokwadratowego.

Listing 5.16. Użycie funkcji straty na przykładzie błędu średniokwadratowego

```
1  using namespace shark;
2
3  // [...]
4
5  SquaredLoss<> mse_loss;
6  auto mse = mse_loss(train_data.labels(), predictions);
7  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 5.17 przedstawia sposób ich wyliczenia, posiadając wartość błędu średniokwadratowego.

Listing 5.17. Implementacja metryk R^2 oraz adjusted R^2

```
1  using namespace shark;
2
3  // [...]
4
5  // błąd średniokwadratowy
6  SquaredLoss<> mse_loss;
7  auto mse = mse_loss(train_data.labels(), predictions);
8
9  // metryka R^2
10 auto var = variance(train_data.labels());
11 auto r_squared = 1 - mse / var(0);
12
13 // metryka adjusted R^2
14 auto adj_r_squared = 1 - (1 - r_squared)((num_regressors - 1)/
15                       (num_regressors - data_size - 1));
```

5.5.3. Metryka AUC-ROC

AUC-ROC 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 5.18 przedstawia sposób obliczenia wartości wymienionej metryki.

Listing 5.18. Przykład użycia klasy *NegativeAUC* do obliczenia pola pod funkcją ROC

```
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 5.19 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 5.19. 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;
```

```
12 double polynomial_degree = 8;
13 int num_epochs = 300;
14
15 // konfiguracja docelowego modelu
16 PolynomialModel<> model;
17 PolynomialRegression trainer(regularization_factor, polynomial_degree,
18                              num_epochs);
19
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.

5.7. Przykłady testowe

Niniejszy rozdział przedstawia sposób praktycznego wykorzystania omówionych wcześniej metod wraz z porównaniem uzyskiwanych przez nie wyników na przykładzie problemu omówionego w rozdziale 3. Odczyt danych realizowany jest z pliku .csv za

pomocą funkcji dostępnej w bibliotece Shark-ML, natomiast sam plik uległ takiemu samemu przetwarzaniu wstępnemu jak opisano w niniejszej sekcji w rozdziale 4. Podobnie jak poprzednio każdy z modeli zawarty został w osobnej funkcji, pozwalając na wywołanie wszystkich poniższych przykładów z poziomu pojedynczego programu o funkcji *main()* przedstawionej na listingu **??**.

PONIŻSZY KOD PRZETESTOWAĆ I SKOMPILOWAĆ

Listing 5.20. Napisany dotychczasowo kod

```
1  #include <fstream>
2  #include <filesystem>
3  #include <regex>
4  #include <string>
5  #include <string_view>
6
7  #include <sharkModels.hpp>
8
9  int main()
10 {
11     sharkModels();
12 }
```

5.7.1. Regresja logistyczna

KOD DO PRZETESTOWANIA

5.7.2. Maszyna wektorów nośnych

KOD DO PRZETESTOWANIA

5.7.3. Sieć neuronowa

KOD DO PRZETESTOWANIA

Rozdział 6

Biblioteka Dlib

6.1. Wprowadzenie

Jest to biblioteka do uczenia maszynowego napisana w nowoczesnym C++, o zastosowaniu przemysłowym oraz naukowym [14]. Podobnie jak poprzednio omawiane biblioteki, posiada ona otwarte źródło na licencji Boost Software Licence [15]. Do dziedzin wykorzystujących wyżej wspomnianą bibliotekę należą robotyka, systemy wbudowane, telefony komórkowe 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.

6.3.2. Redukcja wymiarowości

6.3.2.1. PCA

6.3.2.2. Liniowa analiza dyskryminacyjna

6.3.2.3. Mapowanie Sammona

6.4. Modele uczenia maszynowego

6.4.1. Regresja liniowa

6.4.2. Maszyna wektorów nośnych

6.4.3. Sieci neuronowe

6.4.4. Brzegowa regresja jądra

6.5. Metody analizy modeli

6.5.1. Sprawdzian krzyżowy K-krotny

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 przykłady kodów źródłowych dla programów machine learningu zgrupowane są razem z innymi mechanizmami.

6.7. Przykłady testowe

6.7.1. Maszyna wektorów nośnych

6.7.2. Sieć neuronowa

Rozdział 7

Zestawienie zbiorcze i podsumowanie

7.1. Oferowane funkcjonalności

7.2. Wymagany nakład pracy

7.3. Jakość i ilość dostępnych źródeł referencyjnych

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. *Linear Discriminant Analysis*. 2018. URL: http://image.diku.dk/shark/sphinx_pages/build/html/rest_sources/tutorials/algorithms/lda.html.

[9] The Shark developer team. *General Optimization Tasks*. 2018. URL: http://image.diku.dk/shark/sphinx_pages/build/html/rest_sources/tutorials/first_steps/general_optimization_tasks.html.

[10] The Shark developer team. *Support Vector Machines: First Steps*. 2018. URL: http://image.diku.dk/shark/sphinx_pages/build/html/rest_sources/tutorials/algorithms/svm.html.

[11] The Shark developer team. *Nearest Neighbor Classification*. 2018. URL: http://image.diku.dk/shark/sphinx_pages/build/html/rest_sources/tutorials/algorithms/nearestNeighbor.html.

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

[13] 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.

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

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