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 Tensorflow, Shark, Caffe i PyTorch 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.

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 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.

Dalsza część pracy składa się z bloku omówienia i analizy wybranych bibliotek uczenia maszynowego pod kątem określonych kryteriów. Pierwszą z nich, opisaną w rozdziale czwartym, jest biblioteka Eigen. DOPISAĆ !!!!!!!

Rozdział 2

Uczenie maszynowe w ujęciu praktycznym

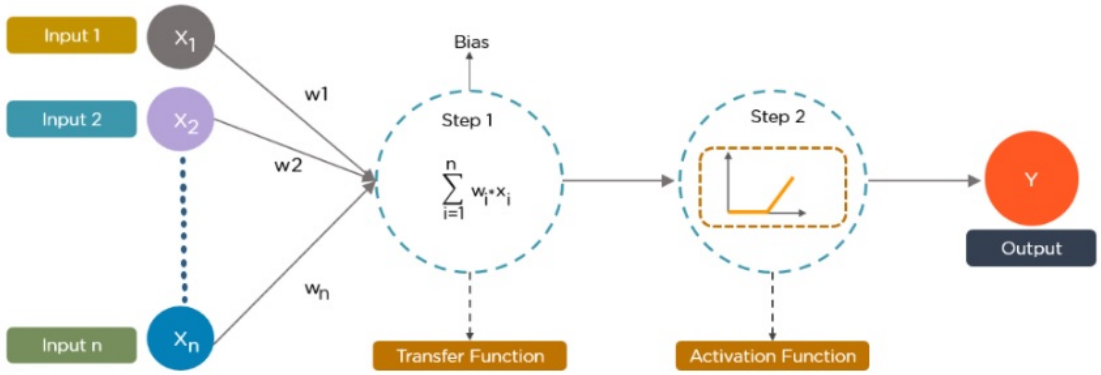
2.1. Problemy współczesnego uczenia maszynowego

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.

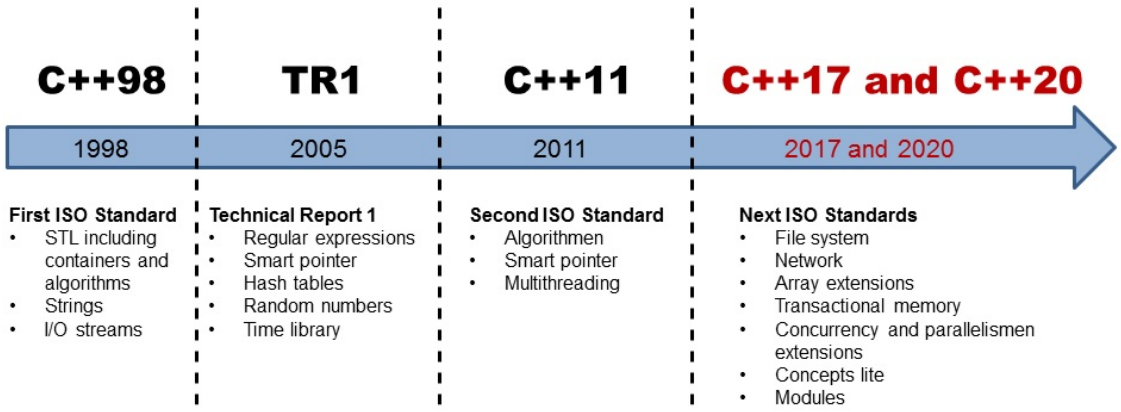
W trakcie rozwoju technologii uczenia maszynowego, postawiono stanowcze kroki w kierunku rozwiązywania powyższych problemów, aby sprostać narastającym wymaganiom związanym z coraz to nowymi i bardziej skomplikowanymi zastosowaniami sztucznej inteligencji. Dokonywano tego poprzez między innymi optymalizację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

3.2. Charakterystyka i przetwarzenie danych

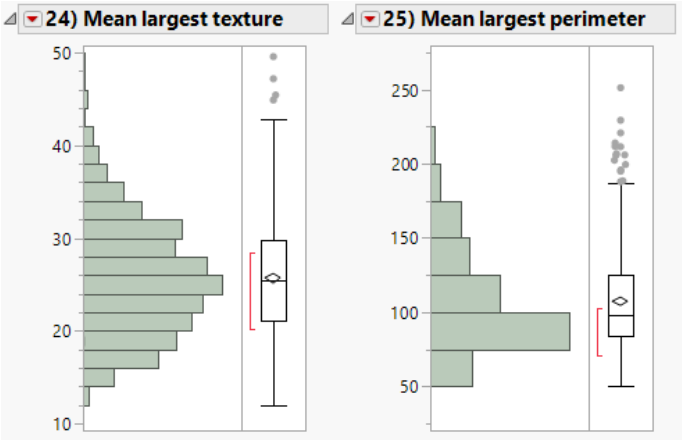
W celu przeprowadzenia procesu uczenia maszynowego, jednym z najistotniejszych kroków jakie należy podjąć jest wstępne zaznajomienie się z zestawem danych i jego analiza pod kątem rozkładu poszczególnych zmiennych oraz prawdopodobieństw. W tym celu wykorzystane zostało oprogramowanie JMP.

3.2.1. 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.

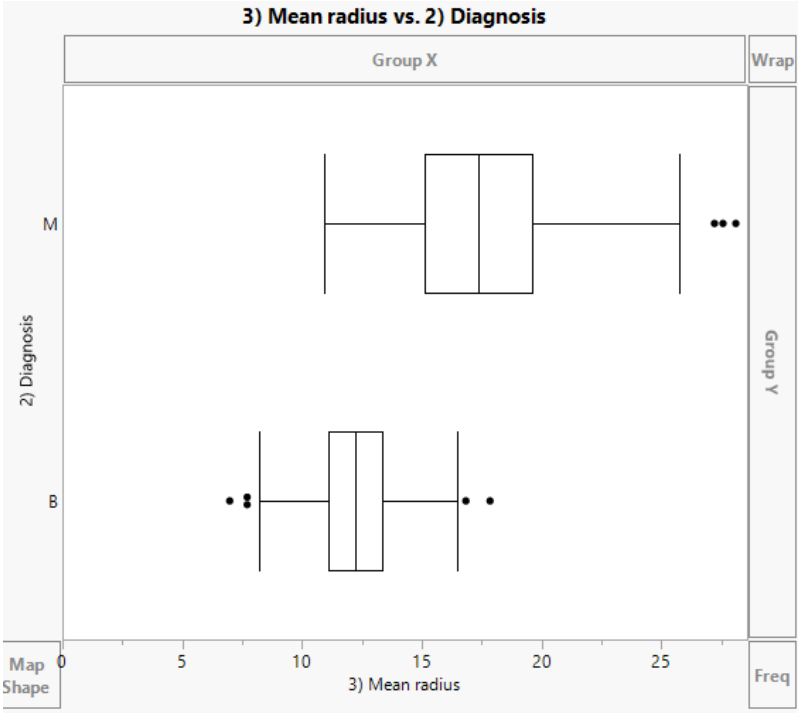
Rysunek 3.1. Histogram rozkładu zmiennej odpowiedzi



Rysunek 3.2. Przykłady histogramów zmiennych decyzyjnych

3.2.2. Czyszczenie i normalizacja rozkładu danych

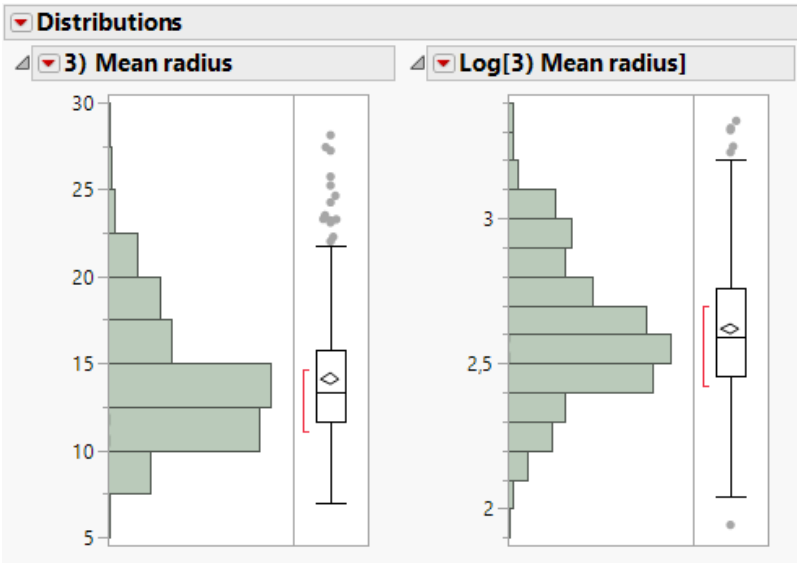
Na pełny zestaw danych składa się 569 obserwacji. Podczas wstępnej analizy stwierdzono istnienie 13 brakujących wartości dla regresora *Std err concave points*, dla których przyjęto wartość średnią z całej kolumny. Głównym problemem okazały się obserwacje odstające oraz skośności rozkładu. Do analizy 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 3.3. 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:

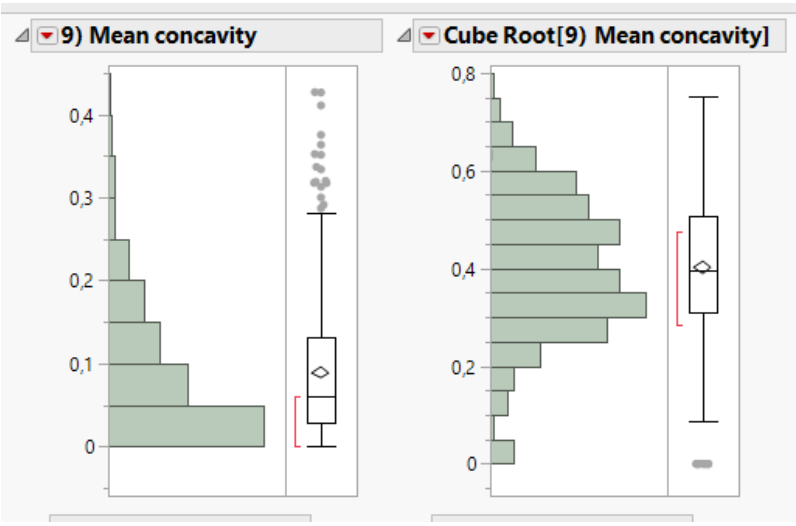
1. *Mean radius*;
2. *Mean texture*;
3. *Mean perimeter*,
4. *Mean area*;
5. *Mean smoothness*;
6. *Mean symmetry*;
7. *Std err texture*;
8. *Std err smoothness*;
9. *Std err compactness*;
10. *Std err concave points*;
11. *Mean largest texture*;
12. *Mean largest smoothness*;
13. *Mean largest compactness*.



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

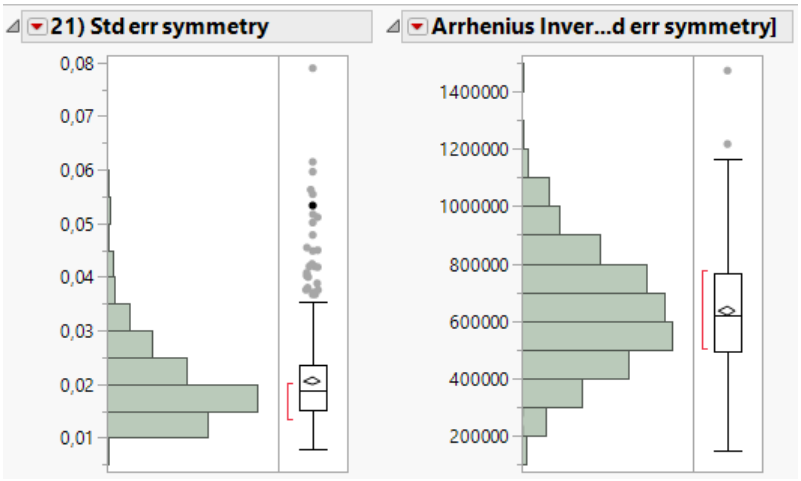
W drugim kroku podjęto próbę wykorzystania transformacji pierwiastkiem sześciennym dla pozostałych zmiennych decyzyjych, ze względu na jej skuteczność dla danych o rozkładzie prawoskośnym. Rysunek 3.5. przedstawia porównanie rozkładu zmiennej *Mean concavity* przed i po transfromacji pierwiastkiem sześciennym.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

3.3. Szablony docelowych modeli dla zadanych danych eksperymentalnych

Ze względu na dychotomiczny charakter zmiennej odpowiedzi, wybrany został przedstawiony poniżej zestaw metod dla których wykonano i przedstawiono 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.

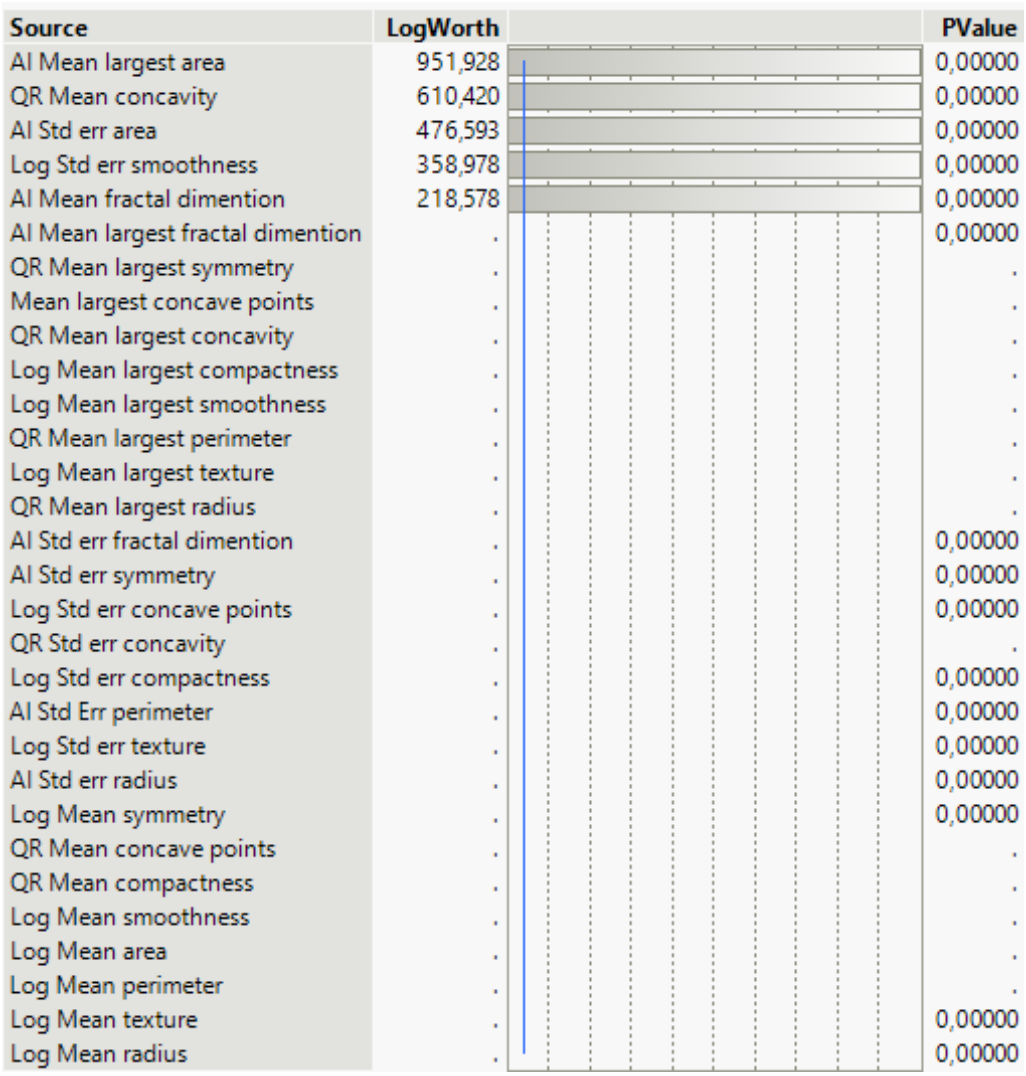
3.3.1. Regresja logistyczna

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 3.7 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 3.8 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



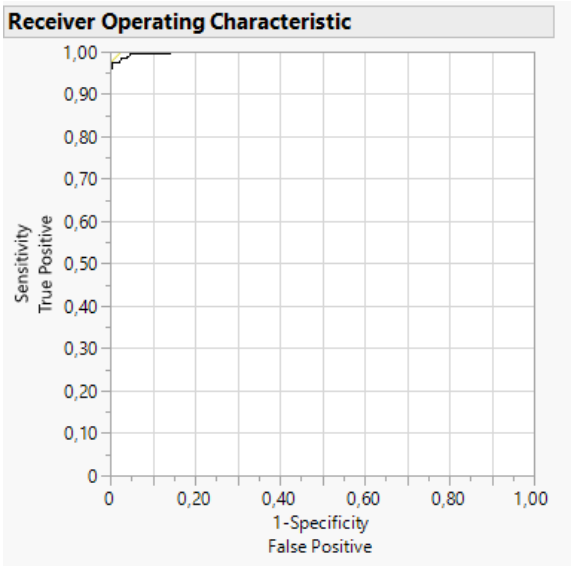
Rysunek 3.7. Wykres p-wartości dla całego zestawu zmiennych decyzyjnych.

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 3.9 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 3.8. Wykres i p-wartości istotnych zmiennych decyzyjnych



Rysunek 3.9. Krzywa charakterystyczna odbiornika (ROC) dla modelu regresji logistycznej

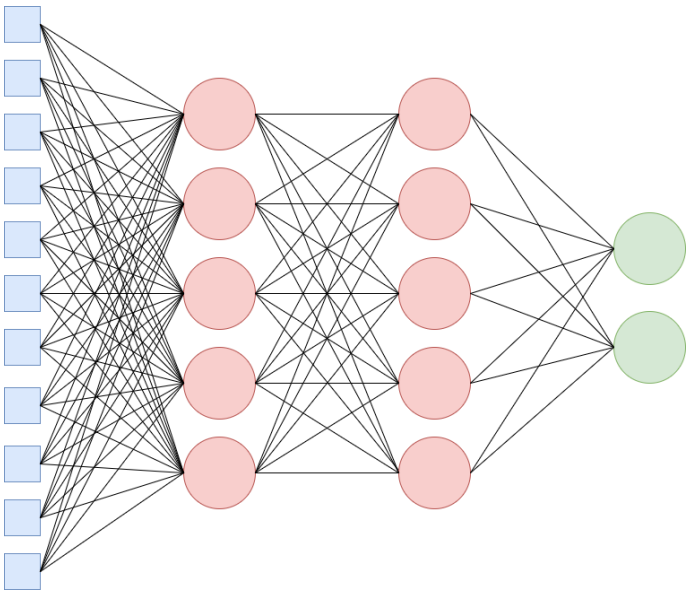
3.3.2. Głęboka sieć neuronowa

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 3.10.

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

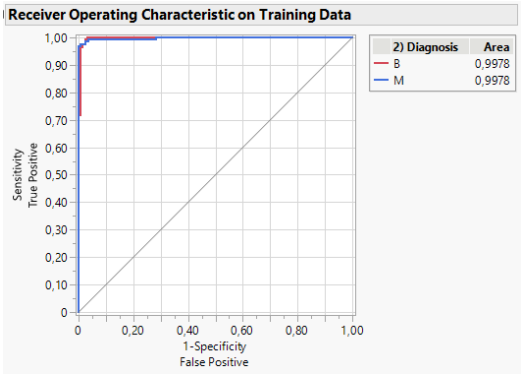
Typ warstwy	ilość neuronów	aktywacja
ukryta	5	tangens hiperboliczny
ukryta	5	tangens hiperboliczny
wyjściowa	2	—

Tabela 3.2. Struktura modelu sieci neuronowej

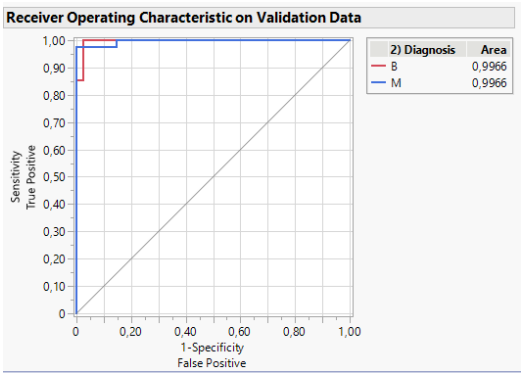


Rysunek 3.10. Schemat struktury sieci

testowego wyniosła 100%, natomiast dla danych uczących napotkano 5 przypadków *false-negative* (prawdopodobieństwo 3%) oraz 1 przypadek *false-positive* (prawdopodobieństwo 0,4%). Rysunki 3.11 oraz 3.12 przedstawiają krzywe charakterystyczne odbiornika dla zestawu testowego i walidacyjnego.



Rysunek 3.11. Krzywa charakterystyczna odbiornika dla zestawu testowego



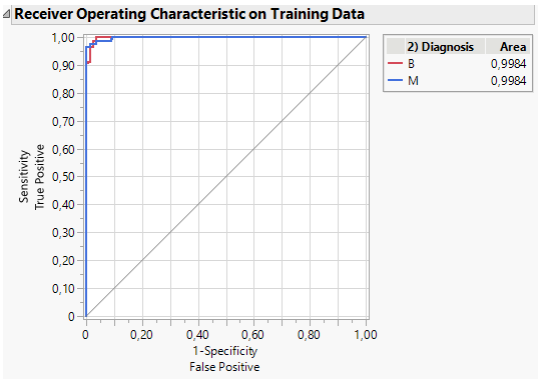
Rysunek 3.12. Krzywa charakterystyczna odbiornika dla danych walidacyjnych

3.3.3. Maszyna wektorów nośnych

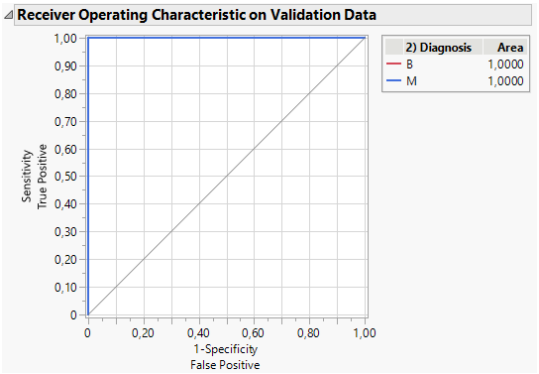
Do predykcji diagnozy wykorzystano ten sam zestaw regresorów, zawartych w tabeli 3.1. Ponownie w celu walidacji użyto metody wybrania losowego zestawu walidacyjnego spośród dostarczonych danych, w proporcji 80% obserwacji uczących i 20% testowych, z użyciem wartości 1234 dla ziarna generatora liczb pseudolosowych. 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 3.13. Krzywa charakterystyczna odbiornika dla danych uczących modelu SVM



Rysunek 3.14. 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 3.13 oraz 3.14 przedstawiają krzywe charakterystyczne odbiornika dla uzyskanego modelu.

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.

Listing 4.1. Przykładowy program do odczytu i przygotowania danych z pliku CSV dla biblioteki Shogun [5]

```
#include <shogun/base/init.h>
#include <shogun/base/some.h>
#include <shogun/io/File.h>

using namespace shogun;
using Matrix = shogun::SGMatrix<float64_t>

// [...]
// odczyt danych z pliku .csv
auto csv_file = shogun::some<shogun::CCSVFile>("sample_file.csv");
Matrix data;
data.load(vsv_file);

// transpozycja i rozdzielenie danych
Matrix::transpose_matrix(data.matrix, data.num_rows, data.num_cols);
Matrix inputs = data.submatrix(0, data.num_cols - 1); // utworzenie widoku
inputs = inputs.clone(); // przekopiowanie danych

// utworzenie widoku
Matrix outputs = data.submatrix(data.num_cols - 1, data.num_cols);
outputs = outputs.clone(); // przekopiowanie danych

// powrotna transpozycja macierzy danych treningowych dla algorytmów
// biblioteki
Matrix::transpose_matrix(inputs.matrix, inputs.num_rows, inputs.num_cols);

// przygotowanie klas wrapujących do wykorzystania przy uczeniu
auto features = shogun::some<shogun::CDenseFeatures<float64_t>>(inputs);
auto labels =
    shogun::wrap(new shogun::CMulticlassLabels(outputs.get_column(0)));
```

4.3. Metody przetwarzania i eksploracji danych

Biblioteka dostarcza możliwość normalizacji typu min-max, zapewniając że dane mieścić się będą w przedziale jednostkowym, za pomocą klasy *shogun::CRescaleFeatures*. Zastosowanie innych transformacji na danych, jak np. transformacji logarytmicznej, wymaga przygotowania własnego kodu. 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.

4.4. Modele uczenia maszynowego

Regresja liniowa realizowana dekompozycją macierzy Choleskiego przy użyciu klasy CLinearRidgeRegression.

4.5. Metody analizy modeli

Log-loss (CLogLoss),

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

4.7.1. Regresja liniowa

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. 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]

```
// [...]

static const std::train_data_url =
"https://raw.githubuserscontent.com/pandas-dev/pandas/master/pandas/
tests/data/iris.csv";
const std::string data_path{"iris.csv"};

// sprawdź istnienie pliku za pomocą std::filesystem
if (!fs::exists(data_path))
{
    if (!utils::DownloadFile(train_data_url, data_path))
    {
        std::cerr << "Unable␣to␣download␣the␣file␣" <<
            train_data_url << std::endl;
        return 1;
    }
}

// odczytaj dane z pliku do ciągu znakowego
std::ifstream data_file(data_path);
std::string train_data_str((std::istreambuf_iterator<char>(data_file)),
                           std::istreambuf_iterator<char>());

// usuń wiersz zawierający etykiety
train_data_str.erase(0, train_data_str.find_first_of("\n") + 1);

// zamień nazwy klas na identyfikatory
train_data_str =
    std::regex_replace(train_data_str, std::regex("Iris-setosa"), "0");
train_data_str =
    std::regex_replace(train_data_str, std::regex("Iris-versicolor"), "1");
train_data_str =
    std::regex_replace(train_data_str, std::regex("Iris-virginica"), "2");

// odczytanie i parsowanie danych
shark::ClassificationDataset train_data;
shark::csvStringToData(train_data, train_data_str, shark::LAST_COLUMN);
```

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

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. Istnieją ponadto klasy trenerów które w przeciwieństwie do powyższych, nie używają metody *train()*, czego przykładem jest klasa PCA, wykorzystywana do redukcji wymiarowości zestawu danych. Listingi 5.3 oraz 5.4 ukazują sposób wykorzystania powyżej opisanych mechanizmów.

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);
```

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

```
// [...]

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

// konfiguracja enkodera do redukcji wymiarów
pca.encoder(enc, 2);
shark::Data<shark::RealVector> encoded_data = enc(data.inputs());
```

Dodatkowymi funkcjami jest możliwość przemieszania danych, i wydzielenia fragmentu jako dane testowe za pomocą metody *shuffle()* klasy *ClassificationDataset* oraz funkcji *splitAtElement()*.

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 5.5 ukazane zostało wykorzystanie podejścia iteracyjnego, natomiast listing 5.6 przedstawia metodę analityczną.

Listing 5.5. Przykład regresji liniowej z wykorzystaniem optymalizatora spadku gradientowego [8]

```
1  #include <shark/Data/Csv.h>
2  #include <shark/Algorithms/GradientDescent/CG.h>
3  #include <shark/ObjectiveFunctions/ErrorFunction.h>
4  #include <shark/ObjectiveFunctions/Loss/SquaredLoss.h>
5  #include <shark/Models/LinearModel.h>
6
7  using namespace shark;
8
9  RegressionDataset loadData(const std::string& dataFile,
10                             const std::string& labelsFile)
11 {
12     Data<RealVector> inputs;
13     Data<RealVector> labels;
14
15     try
16     {
17         importCSV(inputs, regressorsFile, ' ');
18         importCSV(labels, targetFile, ' ');
19     }
20     catch(const std::exception& e)
21     {
22         std::cerr << e.what() << '\n';
23         exit(EXIT_FAILURE);
24     }
25     RegressionDataset data(inputs, labels);
26     return data;
27 }
28 // [...]
```

```
29
30 // odczytanie danych z plików oraz podział na dane uczące i testowe
31 RegressionDataset data = loadData("data/regressionInputs.csv",
32                                    "data/regressionLabels.csv");
33 RegressionDataset test = splitAtElement(data, static_cast<std::size_t>(
34                                                 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 5.6. 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. Liniowa analiza dyskryminacyjna

Model liniowej 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()*. Predykcje uzyskiwane są za pomocą wywołania obiektu modelu jak funkcji (użycie operatora ()) przekazując mu dane uzyskane z ClassificationDataset za pomocą metody *inputs()*. Szczegóły implementacyjne zamieszczone zostały na listingu 5.7.

Listing 5.7. Przykład klasyfikacji z wykorzystaniem modelu LDA [9]

```
#include <shark/Algorithms/Trainers/LDA.h>
// [...]

using namespace shark;

int main(int argc, char **argv)
{
    // import danych
    ClassificationDataset data;
    try
    {
        importCSV(data, argv[1], LAST_COLUMN, ' ');
    }
    catch(...)
    {
        std::cerr << "Unable to read data from file " << argv[1] << std::endl;
        exit(EXIT_FAILURE);
    }

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

    // wyodrębnienie danych testowych
    auto test = splitAtElement(data, .5 * data.numberOfElements());

    // utworzenie i wytrenowanie modelu
    LDA ldaTrainer;
    LinearClassifier<> lda;
    ldaTrainer.train(lda, data);

    // analiza predykcji i dokładności modelu
    Data<unsigned int> prediction;
    ZeroOneLoss<unsigned int> loss;
    prediction = lda(data.inputs());
    std::cout << "LDA on training set accuracy: "
        << 1. - loss(data.labels(), prediction) << std::endl;
    prediction = lda(test.inputs());
    std::cout << "LDA on test set accuracy: "
        << 1. - loss(test.labels(), prediction) << std::endl;
}
```

5.4.3. 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 *One-*

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

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

```
1  using namespace shark;
2
3  // [...]
4
5  void LRClassification(const ClassificationDataset& train,
6                        const ClassificationDataset& test,
7                        unsigned int num_classes)
8  {
9      // utworzenie obiektu docelowego klasyfikatora oraz tablicy
10     // klasyfikatorów składowych
11     OneVersusOneClassifier<RealVector> ovo;
12     auto pairs = num_classes * (num_classes - 1) / 2;
13     std::vector<LinearClassifier<RealVector>> lr(pairs);
14
15     // iteracyjne konfigurowanie klasyfikatorów składowych
16     for (std::size_t n = 0, cls1=1; cls1 < num_classes; ++cls1)
17     {
18         using BinaryClassifierType =
19             OneVersusOneClassifier<RealVector>::binary_classifier_type;
20         std::vector<BinaryClassifierType*> ovo_classifiers;
21         for (std::size_t cls2 = 0; cls2 < cls1; ++cls2, ++n)
22         {
23             // pobranie binarnego podproblemu
24             ClassificationDataset binary_cls_data =
25                 binarySubProblem(train, cls2, cls1);
26
27             // trening modelu składowego
28             LogisticRegression<RealVector> trainer;
29             trainer.train(lr[n], binary_cls_data);
30
31             // załadowanie modelu składowego do serii
32             ovo_classifiers.push_back(&lr[n]);
33         }
34         // podłączenie serii do głównego klasyfikatora
35         ovo.addClass(ovo_classifiers);
36     }
37
38     // użycie modelu
39     auto predictions = ovo(test.inputs());
40     // [...]
41 }
```

Listing 5.9. Przykład prostej binarnej regresji logistycznej

```
1  using namespace shark;
2
3  // [...]
4  void SimpleLR(const ClassificationDataset& train,
5                const ClassificationDataset& test)
6  {
7      // utworzenie modelu oraz trenera
8      LinearClassifier<RealVector> model;
9      LogisticRegression<RealVector> trainer;
10
11     // trenowanie modelu
12     trainer.train(model, train);
13
14     // wykorzystanie modelu
15     auto predictions = model(test.inputs());
16     // [...]
17 }
```

5.4.4. Maszyna wektorów nośnych

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

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

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

- przepustowość modelu - podawana w konstruktorze *GaussianRbfKernel* jako liczba z przedziału $\langle 0; 1\rangle$;
- regularyzacja - podawana jako liczba rzeczywista w konstruktorze *CSvmTrainer*, domyślnie maszyna wektorów nośnych używa kary typu *1-norm penalty* za przekroczenie docelowej granicy;

- bias - flaga binarna (bool) określająca czy model ma używać biasu, podawana w konstruktorze *CSvmTrainer*;
- *sparsify* - parametr określający czy model ma zachować wektory które nie są nośne, dostępny przez metodę *sparsify()* trenera;
- minimalna dokładność zakończenia nauczania - pozwala wyspecyfikować precyzję modelu, jest dostępna jako pole struktury zwracane przez metodę *stoppingCondition()* klasy trenera;
- wielkość cache - ustawiana za pomocą funkcji *setCacheSize()* trenera;

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

Listing 5.10. Przykład maszyny wektorów nośnych dla problemu binarnego [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);


```

```
36 // trenowanie modelu
37 trainer.train(kc, training);
38
39 // wyświetlenie informacji diagnostycznych o uczeniu
40 std::cout << "Needed␣" << trainer.solutionProperties().seconds
41           << "␣seconds␣to␣reach␣a␣dual␣of␣"
42           << trainer.solutionProperties().value << std::endl;
43
44 // użycie modelu
45 auto predictions = kc(test.inputs());
```

Listing 5.11. 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 }
```

5.4.5. Sieć neuronowa

5.4.6. Neuronowa sieć splotowa

Biblioteka Shark-ML oprócz omówionych wyżej modeli, wspiera także modele do przetwarzania grafiki w postaci neuronowych sieci splotowych. Leżą jednak one poza zakresem niniejszej pracy, w związku z czym nie zostaną one dokładnie omówione.

5.5. Metody analizy modeli

https://www.shark-ml.org/doxygen_pages/html/group__lossfunctions.html
SquaredLoss ZeroOneLoss

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

5.7.1. Regresja liniowa

5.7.2. Maszyna wektorów nośnych

5.7.3. Sieć neuronowa

Rozdział 6

Biblioteka Dlib

6.1. Wprowadzenie

Jest to biblioteka do uczenia maszynowego napisana w nowoczesnym C++, o zastosowaniu przemysłowym oraz naukowym. Podobnie jak poprzednio omawiane biblioteki, posiada ona otwarte źródło na licencji Boost Software Licence [11]. 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

Jednym z ważniejszych aspektów pracy z danymi w bibliotece Dlib jest przystosowanie ich do procesu uczenia, np. przez transformacje normalizujące rozkład. Mimo że biblioteka sama w sobie nie udostępnia funkcji realizujące operacje transformacji, dostarcza ona szereg operacji na macierzach i wektorach, dzięki czemu użytkownik może pracować na danych już w docelowej strukturze, aplikując własne przekształcenia. Do wspieranych działań należą zarówno działania na macierzach, jak i na ich elementach jak np. mnożenie według elementów (ang. *elementwise multiplication*). Biblioteka udostępnia także 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.

6.4. Modele uczenia maszynowego

Biblioteka Dlib udostępnia szereg metod uczenia maszynowego, zarówno dla danych liczbowych, jak i obrazów, jednak ze względu na dokonywanie analizy porównawczej w kontekście zastosowań dla biostatystyki, ta sekcja uwzględnia jedynie metody przeznaczone do pracy z danymi numerycznymi.

6.5. Metody analizy modeli

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. Regresja liniowa

6.7.2. Maszyna wektorów nośnych

6.7.3. 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. *General Optimization Tasks*. 2018. URL: http://image.diku.dk/shark/sphinx_pages/build/html/rest_sources/tutorials/first_steps/general_optimization_tasks.html.

[9] The Shark developer team. *Linear Discriminant Analysis*. 2018. URL: http://image.diku.dk/shark/sphinx_pages/build/html/rest_sources/tutorials/algorithms/lda.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] Dlib team. *Dlib License*. 2003. URL: http://dlib.net/license.html.