Uniwersity of Zielona Góra

Faculty of Computer, Electrical and Control Engineering

Master's Thesis

Kierunek: Computer Science

Comparative analysis of machine learning libraries in C++ for applications in biostatistics

Kacper Wojciechowski, B.CS

Promotor:

Pracę akceptuję:

.....
(data i podpis promotora)

Zielona Góra, June 2023

Streszczenie

This dissertation aims to analyze and compare machine learning libraries available for C++ for use in biostatistics. The following chapters describe:

- general problems encountered in the process of implementing machine learning solutions;
- characteristics of biostatistics datasets used for testing chosen libraries;
- control results of the methods selected for libraries comparison purpose;
- Shogun, Shark-ML and Dlib libraries with implementations of selected machine learning methods;
- summary of featurs offered by each library in respect to each other.

Keywords: machine learning, C++, library, neural network, deep machine learning, shallow machine learning.

Contents

Chapter 1

Introduction

1.1. About modern challenges

In current times people encounter more and more devices with intelligent functions, such as predicting phenomenons based on various datasets, image recognition, speech analysis, or natural language processing. They make its way into more daily life aspects, such as eg. medicine, work and private homes. Depending on the needs, few of the applicable uses of machine learning in medicine are recognition of cancer cells using MRI scans, diagnosis based on patient's symptoms, or setting norms for compounds naturally occuring in human body, depending on the situation (such as stress) and organic samples tests.

One of the major parts of medical field is biostatistics, based on statistical analysis and reasoning based on data such as blood tests results, eating habits and lifestyle of the patient. Especially important type of systems present in this field are expert systems, utilizing shallow and deep machine learning to help doctors diagnose any potential issues. [1]

At the foundation of aforementioned subjects lies the implementation of machine learning solutions and all related problems. Due to this fact, over the decades specialists created various tools, such as libraries and frameworks, aiming to help programmers in quick and accurate application of artificial intelligence products on different platforms and in different languages, starting from C++, through Python, reaching environments such as Matlab [2].

A major step in preparing AI software is proper selection of needed tools, performed at the design stage, so that they offer functionalities adequate to the requirements. This dissertation performs comparative analysis of machine learning libraries for C++ for use in biostatistics, and aims to help the reader choose the right tools for a research project. It is worth to mention, that only a selected subset of functionalities was presented, suitable for the context of test datasets. As such, mentioned libraries can offer broader range of more precise methods, or new method added after creation of this paper. The realization of this dissertation, the implementation parts were inspired by the examples in this handbook [3], however each listing is the creation of the author, fine-tuned to the used methods and datasets, and equiped with evaluation mechanisms.

1.2. Aim and scope

Aim of this dissertation is to perform a comparative analysis and breakdown of machine learning libraries for C++, presenting examples based on biostatistics datasets.

Scope of the thesis:

- overview of available libraries;
- data engineering;
- shallow and deep supervised learning;
- efficiency aspects of selecting models;
- practical experiment based on medical and biological data.

1.3. Structure

First chapter shows the general aspect touched in this paper, starting from the field of the problem and its uses, ending at the dissertations main topic, additionally describing aim and scope of the thesis, and its structure.

Following chapter guides user through the machine learning topic and usually encountered problems regarding computational complexity and resource usage. Those issues are the base argument of suggesting C++ language as a desired technology for solving them via various libraries.

Data engineering is the main topic of the third chapter. It describes the choice and preparation of biostatistics datasets for the learning process, and selected benchmark methods. The reader is presented with the way of data normalization, selection of regressors, and the control results of selected methods.

Three of the following chapters contain the analysis of selected main functionalities of Shogun, Shark-ML and Dlib libraries respectively. They discuss how does the way of work with each of the products look like, starting from accepted data formats, through data manipulation, ending on available models and their analysis.

Seventh chapter sums up the similarities and differences between mentioned libraries based on results of benchmarking machine learning methods, and available functionalities. Additionally, it describes author's subjective opinion regarding his personal experiences with creating implementations, and working with knowledge sources for each of the product.

Chapter 2

A practical take at machine learning

2.1. Modern problems

Machine learning is based on advanced algorithms and complex data structures, performing calculations on training and validation datasets provided by the user, aswell as data received during their regular operation.

Some of the most basic models are created via techniques such as linear and non-linear regression, logistics regression or linear discriminant analysis. They result in shallow machine learning models, which have relatively small computational complexity to evaluate results of a dataset in comparison to other models [4].

Few of the more soffisticated methods are for example decision trees, based on algorythmical tree logic [5]. Each of the tree layer corresponds to the best predictor available at a current state, causing branching to specific values or ranges of values. The result calculation is achieved via traversing the tree from the root to one of its ending leafs.

The most advanced, yet at the same time most demanding in complexity and memory types of deep learning models such as deep neural networks, convolutional neural networks and recursive neural networks. They utilize structure consisting of one input layer, one or more hidden layers containing mathematical neurons, and one output layer. Each subsequent layer is connected to the previous one, either fully or partially, feeding the results of processing forward. Basic neural networks make neurons present in the same layer independent from one another. Each connection has its own weight, which is multiplied by the input from that connection, and sumed up with the results from other connections, to be passed forward to the activation function. This function decide whether a neuron should activate, and (in case of continuous range $\langle 0; 1 \rangle$) to what extent [6]. An example of a network with a singular neuron was presented on figure 2.1.

Extensive neural networks, such as CNN, require additional steps to preprocess the input data, such that it is acceptable by the network, like for example pooling. By analizing structures incorporated by each of the mentioned methods, following implementation challenges can be identified [7]:

- Efficiency - it is tightly connected to the computational complexity of the

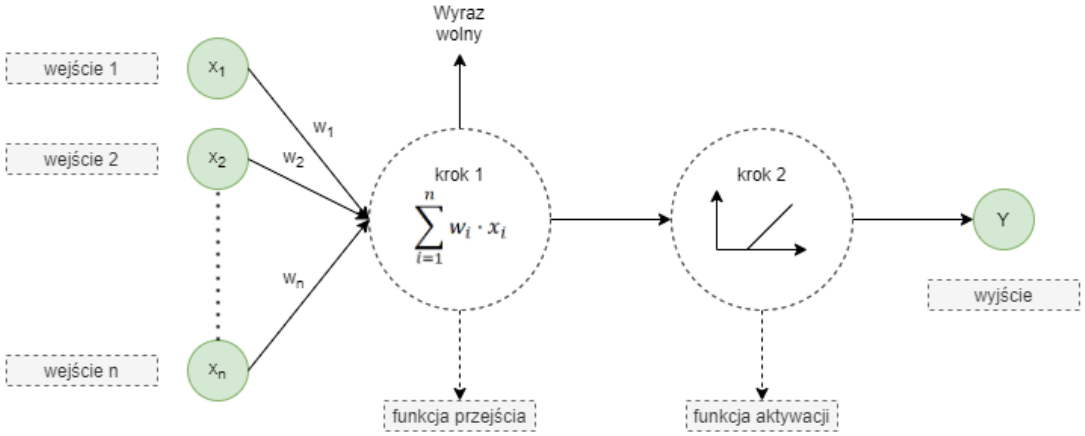

Figure 2.1. Neuron blueprint – Simplelearn.

used methods, aswell as the characteristics of used programming language and hardware platform. The desired effect would be to minimize the learning time of the model and propagation time from inputing the data to receiving a result. In most cases, minimalization of propagation time takes priority.

- Memory usage - this is a concern mainly when using platform with limited resources, such as microcontrolers and microcomputers, where current RAM and flash memory sizes (especially in embedded systems) can be very limited in comparison to regular computers or mobile platforms.

During development of machine learning technology, significant steps were taken into solving above mentioned problems, and fulfill everincreasing requirements of modern machine learning applications. Some of the ways this was achieved are algorithm optimization, selecting high frequency hardware platforms, utilizing paralel computing and using high performance compiled programming languages, especially those that support low level operations.

2.2. C++ as a solution to machine learning problems

Among various languages and environments supporting machine learning, such as Python, C++, Java and Matlab, one of the special few is C++. It is an imperative language with strong typing, connecting low level functionalities for specific hardware architectures with high level programming. As such, it offers vast control over memory usage and optimization possibilities, like adjusting used types for the specific processing needs, contol over variable placing in memory (it is up to the developer to choose whether the data will be placed on stack or heap) and function call optimizations by inlining and tail recursion optimization. In contrary to scripting languages whose code is interpreted during execution, such as Python and Matlab language, C++ is a compiled language, which means that its code is converted into a binary executable adjusted to specific CPU architecture. This completely removes

the overhead of interpreting the source code, as the conversion to CPU language is done only once, during the binary generation process. Additionally, this allows the compiler to perform various low level optimizations [8].

Parts of C++ mechanisms which find their root back in C language allow to use Assembly insets, further increasing the efficiency, at the cost of portability. Some platforms also offer API for hardware acceleration modules, such as eg. Neural Networks API - NNA - of Android, allowing for faster processing via specially designed hardware [9].

Figure 2.2. Concurrency evolution in modern C++ - Modernes C++.

One of the major factors increasing efficiency of machine learning models is paralel processing. Availability of multithreading (added in C++11 standard and further developed, as presented in figure 2.2) and compatibility with CUDA language and API [10] allows C++ to perform many calculations simultaneously utilizing multiple cores of the CPU or delegating processing to one or many graphics cards (where the number of GPU processors vastly outnumbers CPU cores). Additional, although fairly obvious benefit of using this language is easy integration of the models with programs already made in the same language (which is often the case in embedded systems, IoT and computationally intensive software).

2.3. The aim of creating libraries

Due to the complexity of mechanisms that are part of machine learning, various experience of developers and the need for intense optimization, implementation of the machine learning methods is lengthy and expensive. Here programmers can seek for help in libraries created by corporations such as Google and big open-source communities. The libraries provide ready for use mechanisms (often created according to object oriented paradigm, via a set of classes), which are constantly updated and optimized by developers who use it in their daily work or passion projects. They offer a way to quickly create your own models, and often also allow to use already created pre-trained models on stock datasets. Important benefit of using existing libraries that are still supported is better stability, as parts of the library are implemented and tested by experienced developers, like in the case of TensorFlow library provided by Google.

Majority of libraries for machine learning, even in languages such as Python,

are in fact made in C++, providing API for different languages. Sadly, not all libraries made in C++ offer suitable API to use in that same language, as a result, in most cases its use is convoluted and unnecessarily difficult. As a result, some of the libraries work with models created (even by this very same library) in different language, as is the case with TensorFlow lite. Common example is using Python to create a model graph or exporting model to ONNX format (Open Neural Network Exchange) [11]. In this analysis, presented libraries offer the ability to create models in C++, without the need for other language.

Chapter 3

Used datasets

3.1. Overview of experimental data

In order to compare the selected libraries and to show examples, it was necessary to select experimental data that can be used as a reference point. For this purpose, one binary classification and one regression dataset was selected.

3.1.1. Classification data

The classification task was carried out using complete "*Wisconsin Diagnostic Breast Cancer*" from November 1995, containing data about cancer tumors. The dataset was created by William H. Wolberg PhDm W. Nick Street and Olvi L. Mangasarian from the University of Wisconsin [12]. It is available for download from the repository of the University of California [13], and has the following structure:

1) ID - identification number of a pacient;
2) Diagnosis [*Malignant - M / Benign - B*] - tumor type, **response variable**;
3) Classification data:
    a) *Radius*;
    b) *Texture*;
    c) *Perimeter*;
    d) *Area*;
    e) *Smoothness* - described as local differences in tumor radius;
    f) *Compactness* - used to determine the stadium of the tumor;
    g) *Concavity*;
    h) *Concave points*;
    i) *Symmetry* - helpful in evaluation of the spread characteristics;
    j) *Fractal dimention („coastline approximation" - 1)* - describes the complexity of nerve cells, helping to determine whether cells exhibit cancerous mutations.

For each of the variables above, the data contain an average, standard deviation, and average of three highest observations, and are organized sequentially, starting from set of averages. All variables have continuous character. There is also a reduced version of the data containing only the averages set, available as an attachment to this handbook [14].

3.1.2. Dane regresyjne

For the demonstation purposes of regression task, the "IronGlutathione" dataset attached to this handbook [14] was selected. It describes the connection between iron content and glutathione transferase enzyme in human body. It was created in 2012, and contains 90 observations of following 10 variables:

1. *Age*;
2. *Gender*;
3. *Alpha GST (ng/L)* - glutatione transferase type α content;
4. *pi GST (mg/L)* - glutatione transferase type π content;
5. *transferrin (mg/mL)*;
6. *sTfR (mg/mL)* - solvable transferin receptor content;
7. *Iron (mg/dL)*;
8. *TIBC (mg/dL)* - total capacity of iron bonding;
9. *%ISAF (Iron / TIBC)* - transferin saturation coefficient;
10. *Ferritin (ng/dL)*;

Ferritin (ng/dL) was selected as a response variable.

3.2. Data characteristics

During the process of learning by the model, one of the most crucial steps is to familiarize with the dataset and analyze the distribution of the regressors. JMP Pro [15] was selected as the software for this purpose.

3.2.1. Classification data

3.2.1.1. Predictor and response distribution analysis

The analysis was started by characterising the *Diagnosis* variable. Figure 3.1 shows the histogram and table describing the number of observations in each class and the probability of an observation to belong to each of them. The data contains 357 observations of benign tumor with probability of $\approx 62.7\%$, and 212 malignant tumors with probability of $\approx 37.3\%$.

Based on produced histograms, it was determined that majority of the predictors have right-skewed distribution and contain outliers, as shown on the attached box plots from fig. 3.2. *Mean Largest Concave Points* was an exception for despite having lighly skewed distribution, it contains no outliers. Based on this information, in order to properly teach the model, the data needed to be cleaned and normalized.

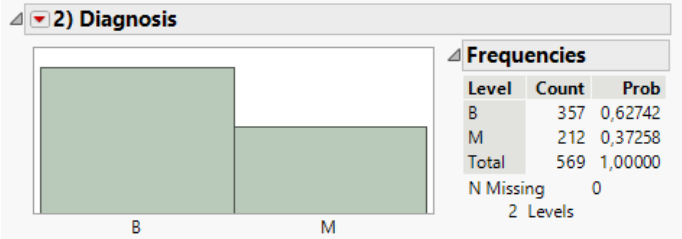

Figure 3.1. Response distribution.

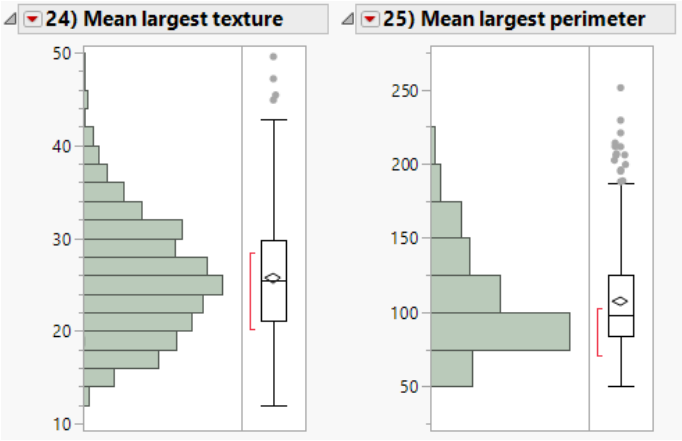

Figure 3.2. Predictors distribution sample.

3.2.1.2. Cleaning and normalizing the data

The full dataset consists of 569 observations, with 13 missing values for *Std err concave points*, which were filled with the average of other values from this column. Outliers and distribution skewedness proved to be the main problem. Box plots were used to analyze and identify observations with outlying values, where Y axis described the response variable, and X axis the cleaned predictor. Sample graph was shown on fig. 3.3. Due to relatively small size of the dataset, in order to reduce or eliminate the need for excluding observations, the data was first normalized.

Logarithmic transformation was used as a first approach to the normalization task for every predictor. In comparison to the original state, *Mean largest concave points* created a distribution very similar to Gaussian distribution, excluding it from attempts to transform it further. Sample results were shown on fig. 3.4. The logarithm proved effective for the following variables:

1. *Mean radius*;
2. *Mean texture*;

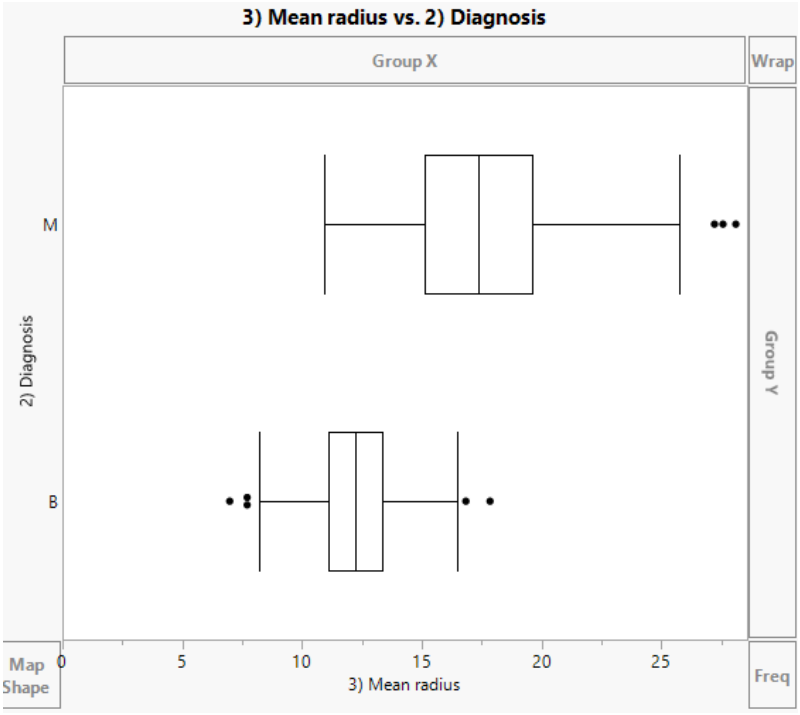

Figure 3.3. Analysis of outliers.

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*.

Next step was to use cube root transformation on each of the remaining variables, due to its effectiveness on right-skewed data. Figure 3.5 shows the achieved results for *Mean concavity*. The transformation was applied to following predictors:

1. *Mean compactness*;
2. *Mean concavity*;
3. *Mean concave points*;

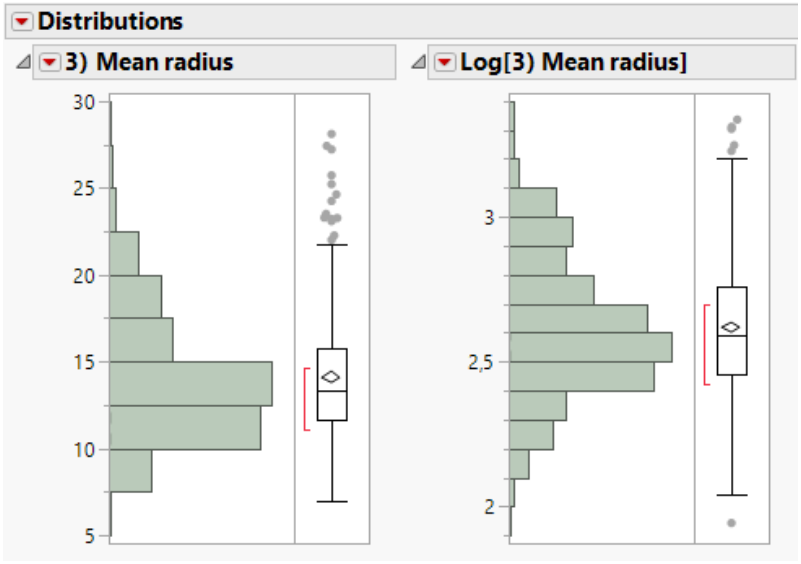

Figure 3.4. Comparison of distributions before and after the logarithmic transformation.

4. *Std err concavity*;
5. *Mean largest radius*;
6. *Mean largest perimeter*;
7. *Mean largest concavity*;
8. *Mean largest symmetry.*

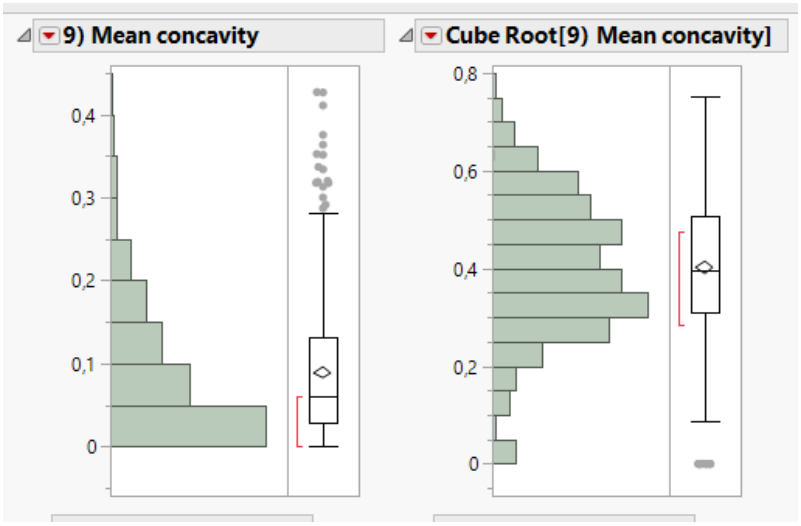

Figure 3.5. Comparison of distributions before and after the cube root transformation.

Inverse Arrhenius transformation was used as the last resort. It takes root in the Arrhenius formula describing energy of chemical reaction activation as 3.1 [16]:

$$\ln k = \ln A - \frac{E_a}{RT} \tag{3.1}$$

where:

- k - constant chemical reactivity coefficient;
- A - frequency coefficient;
- E_a - reaction activation energy;
- R - perfect gas constant;
- T - reaction temperature in Kelvins;

Sadly, part of the modified variables maintained partial right-skewed distribution, however other available transformations, such as square root, square power, $\log(x+1)$, decimal logarithm, power function, and exponential function provided similar or worse effect. Example result is shown on fig. 3.6. The Inverse Arrhenius transformation was applied to following predictors:

1. *Mean fractal dimension*;
2. *Std err radius*;
3. *Std err perimeter*;
4. *Std err area*;
5. *Std err symmetry*;
6. *Std err fractal dimension*;
7. *Mean largest area*;
8. *Mean largest fractal dimension.*

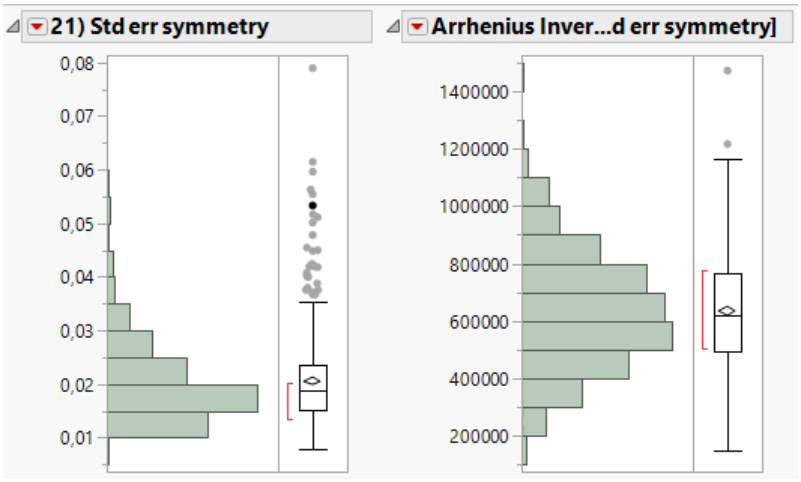

Figure 3.6. Comparison of the distribution before and after inverse Arrhenius transformation.

Due to very small amount of observations, outliers were kept to prevent data loss and further changes in distributions. In order to make the data compatibile with used libraries, the result variable was recoded from using characters to numbers: 1 for malignant and 0 for benign tumor;

Further analysis aimed to exclude highly correlated predictors. This was accomplished via scatterplot matrix, resulting in the exclusion of following variables from the learning process:

1. *Log mean perimeter*;
2. *Log mean area*;
3. *Log mean symmetry*;
4. *Arrhenius inverse std err perimeter*;
5. *Arrhenius inverse std err area*;
6. *Cube root std err concavity*;
7. *Cube root mean largest perimeter*;
8. *Log mean largest compactness*;
9. *Cube root mean largest concavity*;
10. *Mean largest concave points*;
11. *Cube root mean largest radius*;
12. *Cube root mean concave points*;
13. *Cube root mean concavity*;
14. *Log std err compactness*;
15. *Log mean largest texture*;
16. *Log mean largest smoothness*;
17. *Cube root mean largest symmetry.*

Figure 3.7 shows the resulting matrix. After removing correlated variables, the following remained:

1. *Log mean radius*;
2. *Log mean texture*;
3. *Log mean smoothness*;
4. *Arrhenius inverse mean dimension*;
5. *Arrhenius inverse std err radius*;
6. *Log std err texture*;
7. *Log std err smoothness*;
8. *Arrhenius inverse std err symmetry*;
9. *Arrhenius inverse std err dimension.*

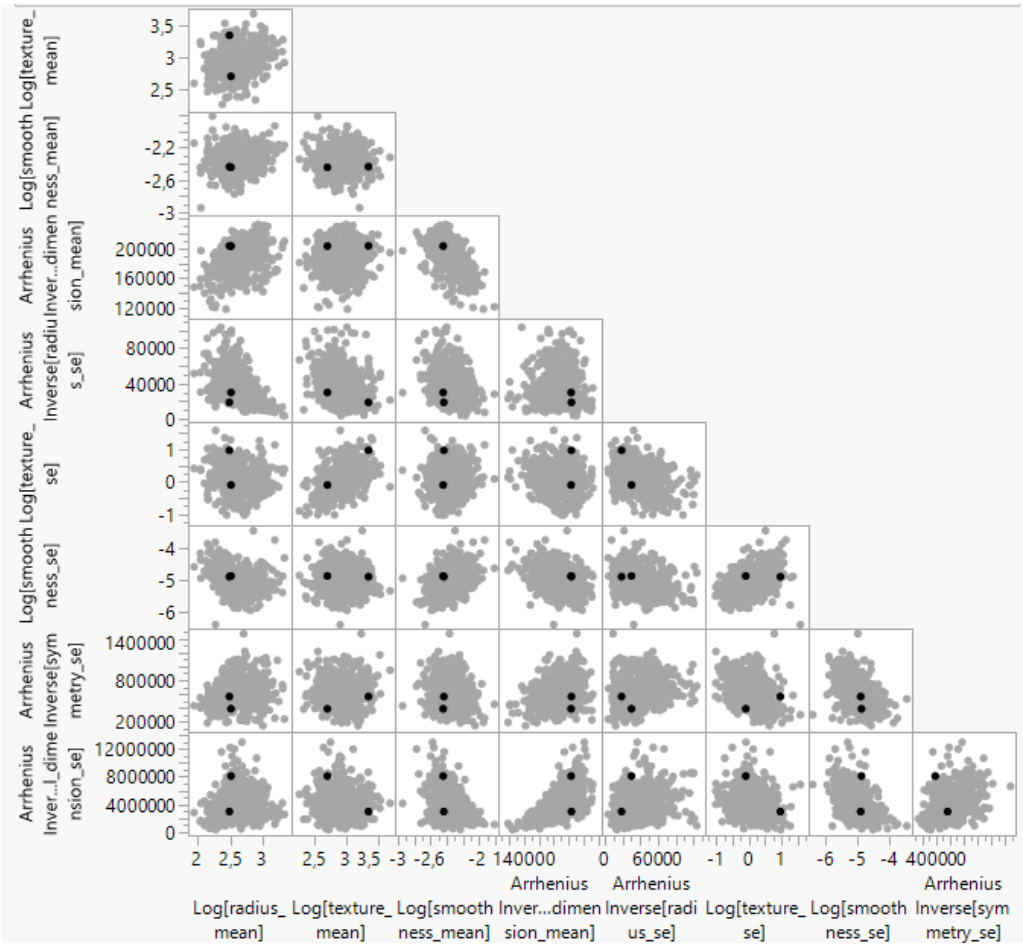

Figure 3.7. Scatterplot matrix after removing correlated predictors

3.2.2. Regression data

3.2.2.1. Distribution analysis

Simiarly to the classification data, the analysis was started from the response variable. It was determined to have highly right-skewed distribution, as shown on fig. 3.8.

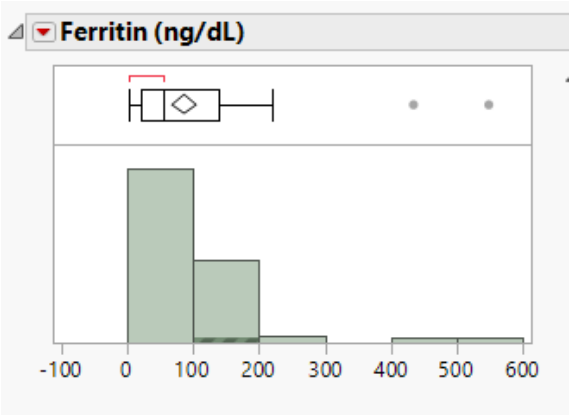

Figure 3.8. Response variable distribution

Attached box plot shows two outliers. The same problem was spotted for *%ISAF*,

Iron, *sTfR*, *Transferrin* and especially *Alpha GST*. Other predictors proved to have distribution close to the Gaussian curve, and not containing any outliers. Figure 3.9 shows sample histograms of the predictors.

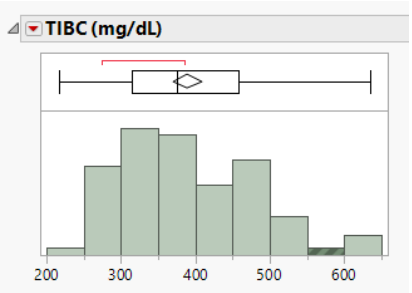

Figure 3.9. Distribution of TIBC.

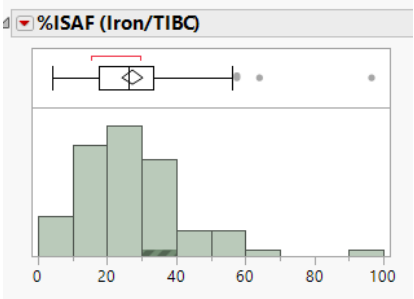

Figure 3.10. Distribution of ISAF.

One variable - *Gender* - has dichotomous character, although it is fairly balanced, with 46.7% of observations assigned to female (F) class and 53.3% to male (M) class. Its distribution was showed on fig. 3.11/

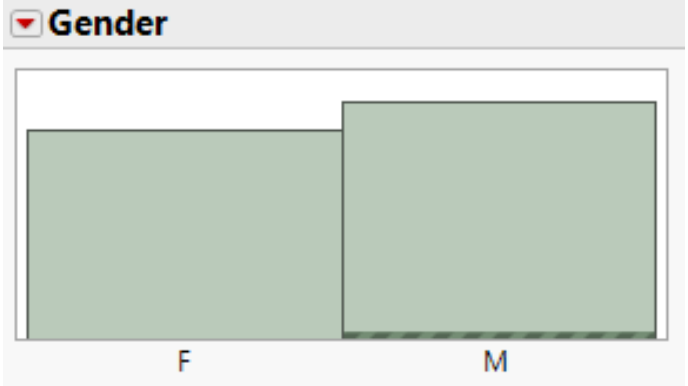

Figure 3.11. Distribution of *Gender*.

3.2.2.2. Cleaning and normalizing the data

It was noticed that one observation had missing value for *Ferritin*. Due to only single missing value for 85 observations, the observation without said value was removed. Other things needed to address were normalizing the distribution of some of the predictors and decision in respect to outliers. As previously, normalization came first. Following variables were deemed good as-is:

1. *Age*;
2. *Gender*;
3. *TIBC*.

For other variables the same three transformations mentioned in section 3.2.1.1 were applied, first of which was using a logarithm. Satisfying result was achieved for predictors:

1. *alpha GST*;
2. *pi GST*;
3. *sTfR*;
4. *Ferritin* (response variable).

Figure 3.12 shows sample change in distribution.

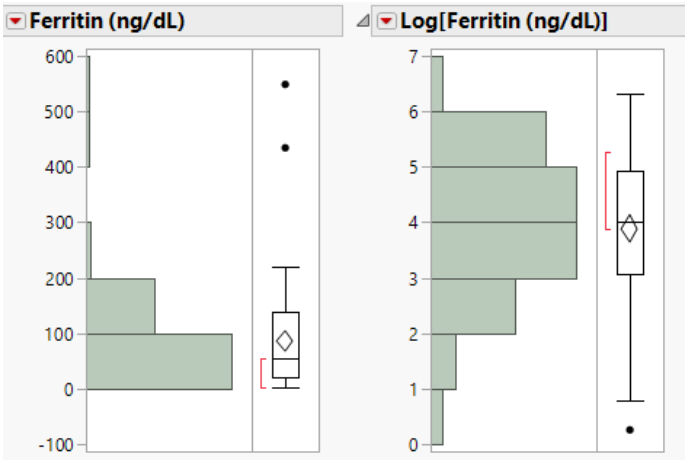

Figure 3.12. Effect of logarithmic transformation on distribution

Second modification via cube root was successfully used, as shown on fig. 3.13, for these variables:

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

The inverse Arrhenius transformation proved ineffective for every variable, producing worse effects from the previous methods. Due to low total amount of observations, and relatively few outliers, it was decided that they should be kept for

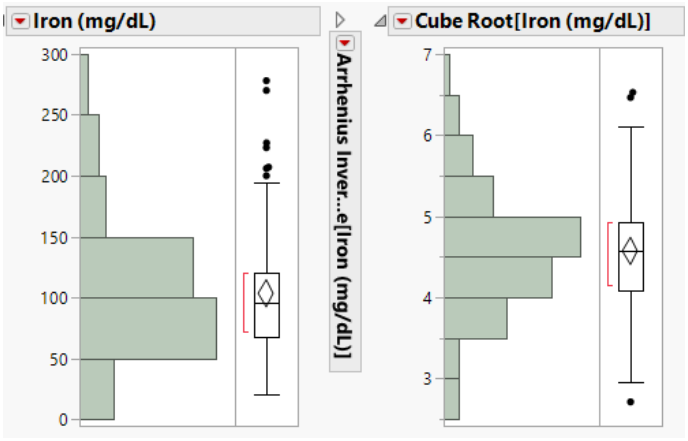

Figure 3.13. Cube root distribution normalization

the learning process. To make the data compatibile with used libraries, the *Gender* variable was recoded from characters to numbers: 1 for female, 0 for male.

3.3. Models built in JMP

After the analysis of the datasets, a few models were selected to serve as a reference for practical tests. Sections below present used templates for the models, such as predictors choice chosen based on practical experimentation with JMP machine learning software.

3.3.1. Support Vector Machine (SVM)

This model accepted all variables remaining after the exploration process for learning. A randomized set of observations was used for the validation purpose, in proportions of 80% learning data and 20% testing data, using value 1234 as seed for the pseudorandom number generator. *Radial Basis Function* was chosen as the SVM kernel, which is a default choice for this model in JMP environment.

Predictor	Learned coefficient
Log mean radius	2,6191
Log mean texture	2,9353
Log mean smoothness	-2,3502
Arrhenius inverse mean dimension	186640
Arrhenius inverse std err radius	38170
Log std err texture	0.1049
Log std err smoothness	-5,0286
Arrhenius inverse std err symmetry	635100
Arrhenius inverse std err dimension	4053000

Table 3.1. Predictors coefficients after learning process.

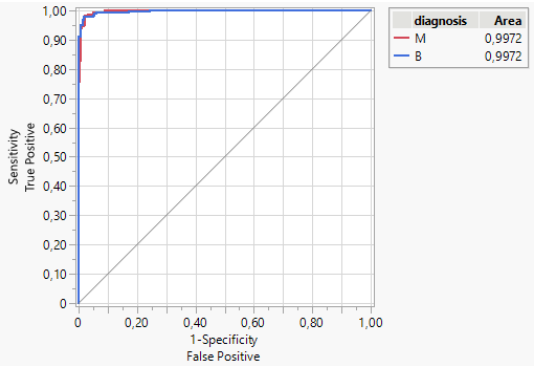

Figure 3.14. ROC curve for the learning data.

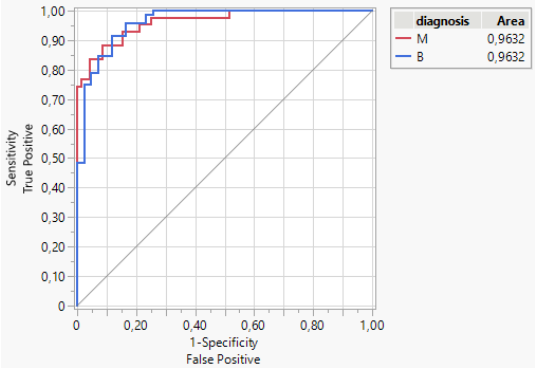

Figure 3.15. ROC curve for the testing data.

The model achieved missclassification rate of 2.64% for learning data and 9.57% for testing data. Table 3.1 shows coefficients determined during learning process for each of the predictors. Figures 3.14 and 3.15 show receiver operation characteristic curves for learning and validation data respectively, presenting classification accuracy of 0.9972 for the former and 0.9632 for the latter.

3.3.2. Linear regression

To determine which predictors have the biggest impact on the response, the p-value was calculated for each of the variables. It was decided that the acceptance threshold should be a p-value equal or below 0.05. Figure 3.16 shows a plot for all predictors, and fig. 3.17 only for those selected for the learning process.

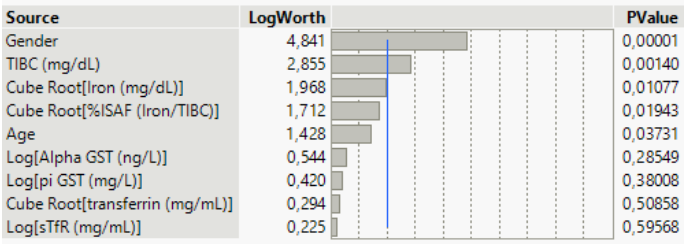

Figure 3.16. P-value plot for all predictors.

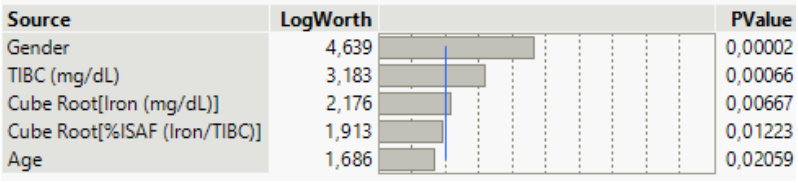

Figure 3.17. P-value plot for selected predictors.

Through the process of elimination, the variables mentioned in the table 3.2 were chosen for the learning process:

Predictor	p-value
Gender	0.00002
TIBC	0.00066
Cube Root Iron	0.00667
Cube Root %ISAF	0.01223
Age	0.02059

Table 3.2. Selected predictors and their p-value

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

Table 3.3. Weights assigned to the predictors

As a result of the learning, a model was created with the linear determination coefficient R^2 of 0.4654525, which suggests that the data are moderately well linearly approximable. The table 3.3 shows the learned weights for each of the predictors.

Chapter 4

Shogun

4.1. Introductions

Shogun is an open-source free machine learning library made in C++, accessible based on *BSD 3-clause* license [17]. It provides interfaces for various languages, such as Python, Ruby or C#, but also allows users to use it in its native language. The library focuses on classification and regression problems.

4.2. Data formats

Base class allowing for loading data into Shogun is *std::vector* present in the Standard Template Library (STL) of C++ language. As such, a user can use any mechanism of loading and parsing data, eg. from a file, network or another device, that at the end returns a vector containing the observations, as long as he provides it himself. One popular choice for storing training data is CSV file, for which Shogun provides support [3]. Still in that case data is a subject to several requirements:

- **The file can only contain numeric data** - if any text data are present, preprocessing is needed to recode them into some numeric values (eg. one-hot encoding or regular subsequent numbers in terms of response variable classes). Sadly this means that data explanations cannot be stored alongside it.
- **Comma separator** - this becomes an issue if data is previously processed via other software, like Microsoft Excel or JMP, as it commonly uses semicolon as a separator. Despite the CSV format allows for several different separators, Shogun accepts only comma as a valid one.
- **Real values should use dot character as a decimal point** - this comes from the characteristics of C++ (and also other languages), and is required for the language to be able to automatically parse a value from text representation to numeric representation.

In order to read and parse data from CSV file, user needs to use *shogun::CCSVFile* class, which result is then loaded into an object of *shogun::SGMatrix*. Because of the fact that this class saved the stored data column-wise, to use them in the training

process, user needs to transpose it, and then separate this matrix into two parts, one of which contains predictors, and the other labels. Sample code snippet of this procedure is shown on listing 4.1. After correct data separation, the container needs to be transposed again so it will be accepted by the learning algorithm, and it needs to be wrapped into specifically designed classes called *CDenseFeatures*, *CMulticlassLabels* or *CRegressionLabels*. This part was shown on listing 4.2.

Listing 4.1. Sample read and preprocessing of data from a CSV file

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

```
49      {
50          case LabelPos::FIRST:
51              ret.trainInputs = data.submatrix(1, data.num_cols)
52                      .clone();
53              ret.trainOutputs = data.submatrix(0, 1).clone();
54              break;
55          case LabelPos::LAST:
56              ret.trainInputs = data.submatrix(0, data.num_cols - 1)
57                      .clone();
58              ret.trainOutputs =
59                  data.submatrix(data.num_cols - 1, data.num_cols)
60                  .clone();
61              break;
62      };
63      // ponowna transpozycja do positaci docelowej dla algorytmów
64      // uczących (operowanie na wierszach)
65      Matrix::transpose_matrix(ret.trainInputs.matrix,
66                               ret.trainInputs.num_rows,
67                               ret.trainInputs.num_cols);
68      Matrix::transpose_matrix(ret.trainOutputs.matrix,
69                               ret.trainOutputs.num_rows,
70                               ret.trainOutputs.num_cols);
71      // podział danych na część treningową i testową
72      auto temp = ret.testInputs = ret.trainInputs.submatrix(
73          static_cast<long>(0.8 * ret.trainInputs.num_cols),
74          ret.trainInputs.num_cols).clone();
75      ret.testInputs = std::move(temp);
76      auto temp2 = ret.trainInputs.submatrix(
77          0, static_cast<long>(0.8 * ret.trainInputs.num_cols))
78              .clone();
79      ret.trainInputs = std::move(temp2);
80      auto temp3 = ret.trainOutputs.submatrix(
81          static_cast<long>(0.8 * ret.trainOutputs.num_cols),
82          ret.trainOutputs.num_cols).clone();
83      ret.testOutputs = std::move(temp3);
84      auto temp4 = ret.trainOutputs.submatrix(
85          0, static_cast<long>(0.8 * ret.trainOutputs.num_cols))
86              .clone();
87      ret.trainOutputs = std::move(temp4);
88      return ret;
89  }
```

Listing 4.2. Repackaging data into desired containers

```
1   #pragma once
2
3   #include <inc/shogun/csv.hpp>
4   #include <inc/shogun/linear.hpp>
5   #include <inc/shogun/logistic.hpp>
6   #include <inc/shogun/svm.hpp>
7   #include <inc/shogun/neural.hpp>
8
9   #include <shogun/base/init.h>
10  #include <iostream>
11
12  inline void shogunModels()
13  {
14      using namespace shogun;
```

```
15
16      init_shogun_with_defaults();
17      // odczytanie danych we własnym pośrednim typie danych
18      auto classificationDatasetTemp =
19          readShogunCsvData("wdbc_data_with_labels_tn.csv", LabelPos::LAST);
20      auto regressionDatasetTemp =
21          readShogunCsvData("IronGlutathione_tn.csv", LabelPos::LAST);
22      // rozdzielenie danych na regresory i zmienne odpowiedzi
23      auto classificationTrainFeatures =
24          some<CDenseFeatures<float64_t>>(
25              classificationDatasetTemp.trainInputs);
26      auto classificationTestFeatures =
27          some<CDenseFeatures<float64_t>>(
28              classificationDatasetTemp.testInputs);
29      auto classificationTrainLabels =
30          some<CMulticlassLabels>(
31              classificationDatasetTemp.trainOutputs);
32      auto classificationTestLabels =
33          some<CMulticlassLabels>(
34              classificationDatasetTemp.testOutputs);
35      auto regressionTrainFeatures =
36          some<CDenseFeatures<float64_t>>(
37              regressionDatasetTemp.trainInputs);
38      auto regressionTestFeatures =
39          some<CDenseFeatures<float64_t>>(
40              regressionDatasetTemp.testInputs);
41      auto regressionTrainLabels =
42          some<CRegressionLabels>(
43              regressionDatasetTemp.trainOutputs);
44      auto regressionTestLabels =
45          some<CRegressionLabels>(
46              regressionDatasetTemp.testOutputs);
47
48      // wywołanie modeli
49      shogunLinear(
50          regressionTrainFeatures,
51          regressionTestFeatures,
52          regressionTrainLabels,
53          regressionTestLabels);
54      shogunSVM(
55          classificationTrainFeatures,
56          classificationTestFeatures,
57          classificationTrainLabels,
58          classificationTestLabels);
59
60      exit_shogun();
61  }
```

4.3. Data processing and exploration methods

4.3.1. Normalizing

The library provides a normalizing min-max mechanism that guarantees the data will be rescaled into a $\langle\ 0;\ 1\ \rangle$ range, with a class *shogun::CRescaleFeatures*. An object of this class can be then used again for the new data corresponding to the

previously learned predictors. Its two main methods are:

- *fit()* - teaches the normalizer object statistical characteristics of the provided data;
- *transform()* - performs normalizing.

In case of some algorithms provided by Shogun, normalizing is one of the first steps executed automatically, so it is not always necessary to do this in preprocessing. Such information should be included in the documentation of a given method. Listing 4.3 shows how to use the aforementioned class. The class, aswell as the function shown on the listing perform the normalizing in-place, hence no need to override the object storing the data.

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. Dimensionality reduction

Shogun gives to the user several algorithms of dimensionality reduction as following classes [3]:

- **Principle Component Analysis** - class *CPCA*;
- **Kernel Principle Component Analysis** - class *CKernelPCA*;
- **Multidimensional Scaling** - class *MultidimensionalScaling*;
- **IsoMap** - class *CIsoMap*;
- **ICA** - class *CFastICA*;
- **Factor Analysis** - class *CFactorAnalysis*;
- **t-SNE** - class *CTDistributedStochasticNeighborEmbedding*.

Each of the classes mentioned above operates by first learning the parameters of the training data by calling the *fit()* function, and establishing the desired dimensions count (except for ICA). Trained reductor object can be used to reduce the dimensionality by *apply_to_feature_vector()* method, that returns transformed data, or in case of ICA, Factor Analysis and t-SNE, by *transform()* method, which result needs to be casted into a pointer to *CDenseFeatures*. Unfortunately, using any of the reductor objects requires creating a new copy of the data object during transformation, instead of performing it in-place. Listing 4.4 shows an example based on class *CKernelPCA*.

Listing 4.4. Dimensionality reduction using Kernel PCA [3].

```
#pragma once

inline void KernelPCA(
    shogun::Some<shogun::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. L1 and L2 Regularisation

In Shogun, regularisation is an integral part of a machine learning model, and as such, it is always performed by given model. Each model defines itself which kind of regularization it performs, and this cannot be changed.

4.4. Machine Learning Models

4.4.1. Linear Regression

One of the fundamental machine learning algorithms provided by the Shogun library is linear regression, performed by the *CLinearRidgeRegression* class. As the name suggests, it utilizes Ridge Regression, configured at the model creation stage. Listing 4.5 shows how to adjust the model.

Listing 4.5. Linear regression example in Shogun

```
#pragma once

#include <iostream>
#include <numeric>
#include <inc/shogun/verify.hpp>
#include <shogun/base/some.h>
#include <shogun/features/DenseFeatures.h>
#include <shogun/labels/RegressionLabels.h>
#include <shogun/regression/LinearRidgeRegression.h>
#include <vector>

inline void shogunLinear(
    shogun::Some<shogun::CDenseFeatures<float64_t>>& trainInputs,
    shogun::Some<shogun::CDenseFeatures<float64_t>>& testInputs,
    shogun::Some<shogun::CRegressionLabels>& trainOutputs,
    shogun::Some<shogun::CRegressionLabels>& testOutputs)
{
    using namespace shogun;

    // utworzenie modelu
    float64_t tauRegularization = 0.0001;
    auto linear =
        some<CLinearRidgeRegression>(tauRegularization, nullptr, nullptr);
    // nauczanie
    linear->set_labels(trainOutputs);
    linear->train(trainInputs);
    // weryfikacja modelu
    std::cout << "-----␣Shogun␣Linear␣-----" << std::endl;
    std::cout << "Train␣data:␣" << std::endl;
    auto predictions = wrap(linear->apply_regression(trainInputs));
    shogunVerifyModel(predictions, trainOutputs);

    std::cout << "Test␣data:␣" << std::endl;
    auto predictions2 = wrap(linear->apply_regression(testInputs));
    shogunVerifyModel(predictions2, testOutputs);
}
```

4.4.2. Logistic Regression

Shogun library contains implementation of multiclass logistic regression via *CMulticlassLogisticRegression* class. It also provides configurable regularisation. Listing 4.6 shows how to use it.

Listing 4.6. Logistic regression example in Shogun

```
#pragma once

#include <inc/shogun/verify.hpp>

#include <iostream>
#include <shogun/base/some.h>
#include <shogun/features/DenseFeatures.h>
#include <shogun/labels/MulticlassLabels.h>
#include <shogun/multiclass/MulticlassLogisticRegression.h>

```

```
11
12 inline void shogunLogistic(
13     shogun::Some<shogun::CDenseFeatures<float64_t>>& trainInputs,
14     shogun::Some<shogun::CDenseFeatures<float64_t>>& testInputs,
15     shogun::Some<shogun::CMulticlassLabels>& trainOutputs,
16     shogun::Some<shogun::CMulticlassLabels>& testOutputs)
17 {
18     using namespace shogun;
19
20     // utworzenie modelu
21     auto logReg = some<CMulticlassLogisticRegression>();
22
23     // nauka modelu
24     logReg->set_labels(trainOutputs);
25     logReg->train(trainInputs);
26
27     // ewaluacja modelu
28     std::cout << "-----␣Shogun␣Logistic␣-----" << std::endl;
29     std::cout << "Train:" << std::endl;
30     auto prediction = wrap(logReg->apply_multiclass(trainInputs));
31     shogunVerifyModel(prediction, trainOutputs);
32
33     std::cout << "Test:" << std::endl;
34     auto prediction2 = wrap(logReg->apply_multiclass(testInputs));
35     shogunVerifyModel(prediction2, testOutputs);
36 }
```

4.4.3. Support Vector Machine

Similarly to logistic regression, Shogun provides the multiclass Support Vector Machine model implementation for classification tasks, as *CMulticlassLibSVM* class. It contains several configurable parameters, along with the choice of kernel itself. Listing 4.7 shows an example of use.

Listing 4.7. Support Vector Machine example in Shogun

```
1  #pragma once
2
3  #include <inc/shogun/verify.hpp>
4
5  #include <iostream>
6  #include <shogun/base/some.h>
7  #include <shogun/features/DenseFeatures.h>
8  #include <shogun/evaluation/CrossValidation.h>
9  #include <shogun/labels/MulticlassLabels.h>
10 #include <shogun/kernel/GaussianKernel.h>
11 #include <shogun/multiclass/MulticlassLibSVM.h>
12 #include <shogun/evaluation/MulticlassAccuracy.h>
13 #include <shogun/evaluation/StratifiedCrossValidationSplitting.h>
14 #include <shogun/modelselection/ModelSelection.h>
15 #include <shogun/modelselection/ModelSelectionParameters.h>
16 #include <shogun/modelselection/GridSearchModelSelection.h>
17 #include <shogun/modelselection/ParameterCombination.h>
18
19 inline void shogunSVM(
20                 shogun::Some<shogun::CDenseFeatures<float64_t>>& trainInputs,
```

```
21                 shogun::Some<shogun::CDenseFeatures<float64_t>>& testInputs,
22                 shogun::Some<shogun::CMulticlassLabels>& trainOutputs,
23                 shogun::Some<shogun::CMulticlassLabels>& testOutputs)
24 {
25     using namespace shogun;
26 
27     // utworzenie jądra
28     auto kernel = some<CGaussianKernel>();
29     kernel->init(trainInputs, trainInputs);
30     // utworzenie i konfiguracja modelu
31     auto svm = some<CMulticlassLibSVM>(LIBSVM_C_SVC);
32     svm->set_kernel(kernel);
33 
34     // poszukiwanie hiperparametrów
35     auto root = some<CModelSelectionParameters>();
36     // stopień unikania missklasyfikacji
37     CModelSelectionParameters* c = new CModelSelectionParameters("C");
38     root->append_child(c);
39     c->build_values(1.0, 1000.0, R_LINEAR, 100.);
40     // utworzenie wskaźnika na jądro
41     auto paramsKernel = some<CModelSelectionParameters>("kernel", kernel);
42     root->append_child(paramsKernel);
43     // utworzenie wskaźnika na wagi
44     auto paramsKernelWidth =
45             some<CModelSelectionParameters>("combined_kernel_weight");
46     paramsKernelWidth->build_values(0.1, 10.0, R_LINEAR, 0.5);
47     paramsKernel->append_child(paramsKernelWidth);
48     // utworzenie podziału do sprawdzianu krzyżowego
49     index_t k = 3;
50     CStratifiedCrossValidationSplitting* splitting =
51             new CStratifiedCrossValidationSplitting(trainOutputs, k);
52     // utworzenie kryterium trafności
53     auto evalCriterium = some<CMulticlassAccuracy>();
54     // utworzenie obiektu sprawdzianu krzyżowego
55     auto cross =
56             some<CCrossValidation>(
57                 svm, trainInputs, trainOutputs, splitting, evalCriterium);
58     cross->set_num_runs(1);
59     // utworzenie obiektu selekcji modelu
60     auto modelSelection = some<CGridSearchModelSelection>(cross, root);
61     // wybór i zaaplikowanie parametrów
62     CParameterCombination* bestParams =
63             wrap(modelSelection->select_model(false));
64     bestParams->apply_to_machine(svm);
65     bestParams->print_tree();
66 
67 
68     // trening
69     svm->set_labels(trainOutputs);
70     svm->train(trainInputs);
71 
72     // ewaluacja modelu
73     std::cout << "-----␣Shogun␣SVM␣-----" << std::endl;
74     std::cout << "Train:" << std::endl;
75     auto prediction = wrap(svm->apply_multiclass(trainInputs));
76     shogunVerifyModel(prediction, trainOutputs);
77 
78     std::cout << "Test:" << std::endl;
```

```
79      auto prediction2 = wrap(svm->apply_multiclass(testInputs));
80      shogunVerifyModel(prediction2, testOutputs);
81  }
```

4.4.4. K-Nearest Neighbours

K-Nearest Neighbours algorith is available as *CKNN* class. It allows user to select the method of calculating distances by passing a correct object, and the count of neighbours considered the nearest. The main available distance metrics are: Euklidean, Hamming, Manhattan and cosinus similarity. In comparison to other methods, it does not require the configuration of hyperparameters, allowing the use of it in crossvalidation without problems. Listing 4.8 presents an example configuration and use of KNN algorithm using Euclidean distance.

Listing 4.8. KNN algorithm example in Shogun

```
1   #pragma once
2
3   #include <inc/shogun/verify.hpp>
4
5   #include <iostream>
6   #include <shogun/base/some.h>
7   #include <shogun/features/DenseFeatures.h>
8   #include <shogun/labels/MulticlassLabels.h>
9   #include <shogun/multiclass/KNN.h>
10  #include <shogun/distance/EuclideanDistance.h>
11
12  inline void shogunKNN(
13                  shogun::Some<shogun::CDenseFeatures>& trainInputs,
14                  shogun::Some<shogun::CDenseFeatures>& testInputs,
15                  shogun::Some<shogun::CMulticlassLabels>& trainOutputs,
16                  shogun::Some<shogun::CMulticlassLabels>& testOutputs)
17  {
18      using namespace shogun;
19
20      // przygotowanie dystansu
21      auto distance = some<CEuclideanDistance>(trainInputs, trainInputs);
22      // przygotowanie modelu
23      std::int32_t k = 3;
24      auto knn = some<CKNN>(k, distance, trainOutputs);
25      // ewaluacja modelu
26      std::cout << "-----␣Shogun␣KNN␣-----" << std::endl;
27      std::cout << "Train:" << std::endl;
28      auto prediction = wrap(knn->apply_multiclass(trainInputs));
29      shogunVerifyModel(prediction, trainOutputs);
30
31      std::cout << "Test:" << std::endl;
32      auto prediction2 = wrap(knn->apply_multiclass(testInputs));
33      shogunVerifyModel(prediction2, testOutputs);
34  }
```

4.4.5. Ensemble algorithms

4.4.5.1. Gradient boosting

Implementation of ensemble algorithm using gradient boosting is adjusted only for regression models. The class *CStochasticGBMachine* is responsible for its execution. It allows the user to configure several parameters, some of which are:

- base algorithm;
- loss function;
- iteration count;
- learning coefficient;
- fraction of vectors to choose at each iteration.

Listing 4.9 shows a method of creating such model using binary decisive tree for regression and classification (implemented by *CCARTree* class) as a base algorithm.

Listing 4.9. Gradient boosting example in Shogun

```
#pragma once

#include <inc/shogun/verify.hpp>

#include <iostream>
#include <shogun/base/some.h>
#include <shogun/features/DenseFeatures.h>
#include <shogun/labels/MulticlassLabels.h>
#include <shogun/loss/SquaredLoss.h>
#include <shogun/lib/SGMatrix.h>
#include <shogun/lib/SGVector.h>
#include <shogun/machine/StochasticGBMachine.h>
#include <shogun/multiclass/tree/CARTree.h>

inline void shogunGradientBoost(
    shogun::Some<shogun::CDenseFeatures>& trainInputs,
    shogun::Some<shogun::CDenseFeatures>& testInputs,
    shogun::Some<shogun::CMulticlassLabels>& trainOutputs,
    shogun::Some<shogun::CMulticlassLabels>& testOutputs)
{
    using namespace shogun;

    // oznaczenie regresorów jako ciągłe
    SGVector<bool> featureType(1);
    featureType.set_const(false);
    // utworzenie binarnego drzewa decyzyjnego
    auto tree = some<CCARTree>(featureType, PT_REGRESSION);
    tree->set_max_depth(3);
    // utworzenie funkcji straty
    auto loss = some<CSquaredLoss>();
    // utworzenie i konfiguracja modelu
    constexpr int iterations = 100;
    constexpr int learningRate = 0.1;
    constexpr int subsetFraction = 1.0;
```

```
35      auto model = some<CStochasticGBMachine>(tree,
36                                              loss,
37                                              iterations,
38                                              learningRate,
39                                              subsetFraction);
40      // trening
41      model->set_labels(trainOutputs);
42      model->train(trainInputs);
43      // ewaluacja modelu
44      std::cout << "-----␣Shogun␣Gradient␣Boost␣-----" << std::endl;
45      std::cout << "Train:" << std::endl;
46      auto prediction = wrap(model->apply_multiclass(trainInputs));
47      shogunVerifyModel(prediction, trainOutputs);
48
49      std::cout << "Test:" << std::endl;
50      auto prediction2 = wrap(model->apply_multiclass(testInputs));
51      shogunVerifyModel(prediction2, testOutputs);
52  }
```

4.4.5.2. Random forest

The random forest method is available in the Shogun library via the *CRandomForest* class. On the contrary to gradient boosting, its implementation allows also for classification tasks. Some of the main configurable parameters are:

- number of trees;
- number of batches to which the data should be separated;
- algorithm of the result selection;
- type of the problem;
- continuity of the regressors.

Listing 4.10 shows how to create and configure random forest model for cosinus approximation.

Listing 4.10. Przykład użycia metody losowego lasu

```
1   #pragma once
2
3   #include "inc/shogun/verify.hpp"
4
5   #include <shogun/base/some.h>
6   #include <shogun/ensemble/MajorityVote.h>
7   #include <shogun/labels/RegressionLabels.h>
8   #include <shogun/lib/SGMatrix.h>
9   #include <shogun/lib/SGVector.h>
10  #include <shogun/machine/RandomForest.h>
11
12  inline void shogunRandomForest(
13      shogun::Some<shogun::CDenseFeatures<DataType>> trainInputs,
14      shogun::Some<shogun::CDenseFeatures<DataType>> testInputs,
15      shogun::Some<shogun::CRegressionLabels> trainOutputs,
```

```
16      shogun::Some<shogun::CRegressionLabels> testOutputs)
17  {
18      using namespace shogun;
19
20      // utworzenie i konfiguracja modelu
21      constexpr std::int32_t numRandFeats = 1;
22      constexpr std::int32_t numBags = 10;
23      auto randForest =
24          some<CRandomForest>(numRandFeats, numBags);
25      auto vote = some<CMajorityVote>();
26      randForest->set_combination_rule(vote);
27      // oznaczenie danych jako ciągłe
28      SGVector<bool> featureType(1);
29      featureType.set_const(false);
30      randForest->set_feature_type(featureType);
31      // trening
32      randForest->set_labels(trainOutputs);
33      randForest->set_machine_problem_type(PT_REGRESSION);
34      randForest->train(trainInputs);
35      // ewaluacja modelu
36      std::cout << "-----␣Shogun␣Random␣Forest␣-----" << std::endl;
37      std::cout << "Train␣data:" << std::endl;
38      auto predictions = wrap(randForest->apply_regression(trainInputs));
39      shogunVerifyModel(predictions, trainOutputs);
40
41      std::cout << "Test␣data:" << std::endl;
42      auto predictions2 = wrap(randForest->apply_regression(testInputs));
43      shogunVerifyModel(predictions2, testOutputs);
44  }
```

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 najszybszego spadku oraz Broydena-Fletchera-Goldfarba-Shannona. Sieć neuronowa posiada wbudowaną regularyzację L2, którą można skonfigurować, podobnie jak pozostałe parametry takie jak współczynnik uczenia, liczba epok, kryterium zbieżności dla funkcji straty, czy wielkość zestawów wsadowych. Niestety, niemożliwy jest wybór funkcji straty, gdyż jest on dokonywany automatycznie na podstawie typu zmiennej odpowiedzi. Listing 4.11 przedstawia pełny proces budowania, konfiguracji oraz uczenia sieci. Niestety ze względu na brak implementacji warstwy neuronu o aktywacji w postaci tangensa hiperbolicznego, na potrzeby prezentacji zastosowania zdecydowano się na wykorzystanie funkcji reLU, co ma wpływ na uzyskane wyniki.

Listing 4.11. Przykład użycia sieci neuronowej.

```
#pragma once

#include "inc/shogun/verify.hpp"

#include <shogun/features/DenseFeatures.h>
#include <shogun/labels/MulticlassLabels.h>
#include <shogun/neuralnets/NeuralLayers.h>
#include <shogun/neuralnets/NeuralNetwork.h>
#include <shogun/base/some.h>

inline void sharkNeural(
    shogun::Some<shogun::CDenseFeatures<float64_t>> trainInputs,
    shogun::Some<shogun::CDenseFeatures<float64_t>> testInputs,
    shogun::Some<shogun::CMulticlassLabels> trainOutputs,
    shogun::Some<shogun::CMulticlassLabels> testOutputs)
{
    using namespace shogun;

    // konstrukcja architektury sieci
    auto dimensions = trainInputs->get_num_features();
    auto layers = some<CNeuralLayers>();
    layers = wrap(layers->input(dimensions));
    layers = wrap(layers->rectified_linear(5));
    layers = wrap(layers->rectified_linear(5));
    layers = wrap(layers->logistic(1));
    auto allLayers = layers->done();
    // utworzenie sieci
    auto network = some<CNeuralNetwork>(allLayers);
    network->quick_connect();
    network->initialize_neural_network();
    // konfiguracja sieci
    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);
    network->set_gd_learning_rate(0.01);
    network->set_gd_momentum(0.5);
```

```
39      // trening
40      network->set_labels(trainOutputs);
41      network->train(trainInputs);
42      // walidacja
43      std::cout << "-----␣Shogun␣Neural␣Network␣-----" << std::endl;
44      std::cout << "Train␣data:" << std::endl;
45      auto predictions = wrap(network->apply_multiclass(trainInputs));
46      shogunVerifyModel(predictions, trainOutputs);
47
48      std::cout << "Test␣data:" << std::endl;
49      auto predictions2 = wrap(network->apply_multiclass(testInputs));
50      shogunVerifyModel(predictions2, testOutputs);
51  }
```

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 modelu oraz zaobserwowanych wartości odpowiedzi. Listing 4.12 ukazuje sposób użycia wspomnianego mechanizmu.

Listing 4.12. Przykład obliczenia wartości błędu średniokwadratowego [3]

```
1  using namespace shogun;
2
3  // [...]
4
5  auto mse_error = some<CMeanSquaredError>();
6  auto mse = mse_error->evaluate(predictions, train_labels);
```

4.5.2. Średni błąd bezwzględny

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

Listing 4.13. Przykład obliczenia wartości średniego błędu bezwzględnego [3].

```
1  using namespace shogun;
2  // [...]
3  auto mae_error = some<CMeanAbsoluteError>();
4  auto mae = mae_error->evaluate(predictions, train_labels);
```

4.5.3. Logarytmiczna funkcja straty

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

Listing 4.14. Przykład użycia klasy *CLogLoss*.

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

4.5.4. Metryka R^2

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

Listing 4.15. Przykład obliczenia metryki R^2.

```
#pragma once

#include <iostream>
#include <cmath>
#include <shogun/evaluation/MeanSquaredError.h>
#include <shogun/evaluation/MeanAbsoluteError.h>
#include <shogun/evaluation/ROCEvaluation.h>
#include <shogun/labels/RegressionLabels.h>
#include <shogun/labels/MulticlassLabels.h>

inline void shogunVerifyModel(
    const shogun::Some<shogun::CRegressionLabels>& predictions,
    const shogun::Some<shogun::CRegressionLabels>& targets)
{
    using namespace shogun;

    // błąd średniokwadratowy
    auto mseError = some<CMeanSquaredError>();
    auto mse = mseError->evaluate(predictions, targets);
    std::cout << "MSE␣=␣" << mse << std::endl;
    // metryka R^2
    float64_t avg = 0.0;
    float64_t sum = 0.0;
    // obliczenie średniej
    for (index_t i = 0; i < targets->get_num_labels(); i++)
    {
        avg += targets->get_label(i);
```

```
28      }
29      avg /= targets->get_num_labels();
30      // obliczenie wariancji
31      for (index_t i = 0; i < targets->get_num_labels(); i++)
32      {
33          sum += std::pow(targets->get_label(i), 2);
34      }
35      float64_t variance = (sum / targets->get_num_labels()) - std::pow(avg, 2);
36      // obliczenie metryki R^2
37      auto r_square = 1 - (mse / variance);
38      std::cout << "R^2 = " << r_square << std::endl << std::endl;
39  }
40
41  inline void shogunVerifyModel(
42      const shogun::Some<shogun::CMulticlassLabels>& predictions,
43      const shogun::Some<shogun::CMulticlassLabels>& targets)
44  {
45      using namespace shogun;
46
47      // obliczenie pola pod wykresem ROC
48      auto roc = some<CROCEvaluation>();
49      roc->evaluate(predictions->get_binary_for_class(1),
50                    targets->get_binary_for_class(1));
51      std::cout << "AUC ROC = " << roc->get_auROC() << std::endl;
52  }
```

4.5.5. Dokładność

Do obliczenia dokładności w przypadku zadań regresji, biblioteka udostępnia klasę *CMulticlassAccuracy*. Pozwala ona nie tylko na samą klasyfikację, lecz także oferuje metodę pobrania macierzy błędnych klasyfikacji. Nie znaleziono natomiast klasy *CAccuracyMeasure* wspomnianej w pracy [3], co sugerowałoby jej usunięcie z biblioteki. Listing 4.16 pokazuje w jaki sposób należy użyć klasy *CMulticlassAccuracy*.

Listing 4.16. Przykład obliczenia dokładności modelu.

```
1  using namespace shogun;
2  // [...]
3  auto acc_measure = some<CMulticlassAccuracy>();
4  auto acc = acc_measure->evaluate(predictions, train_labels);
5  auto confusionMatrix = acc_measure->get_confusion_matrix(
6      predictions, train_labels);
```

4.5.6. Precyzja i pełność (recall), oraz metryka F1

W podręczniku [3] wspomniane zostały klasy *CRecallMeasure* oraz *CF1Measure* mające pozwolić obliczyć odpowiednio pamięć modelu oraz metrykę F1, jednak w trakcie pracy z biblioteką nie znaleziono definicji tych klas, lub jakichkolwiek innych pełniących te funkcje, w związku z czym założono brak implementacji tych metod dla biblioteki Shogun. Zdecydowano się na wspomnienie o tym fakcie, w celu podkreślenia możliwoscy wystąpienia rozbieżności między źródłami wiedzy, a aktualnym stanem biblioteki.

4.5.7. Pole pod wykresem krzywej operacyjnej

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

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

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

4.5.8. K-krotny sprawdzian krzyżowy

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

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

```
#pragma once

inline void shogunCrossValidLogistic(
    shogun::Some<shogun::CDenseFeatures<float64_t>>& trainInputs,
    shogun::Some<shogun::CDenseFeatures<float64_t>>& testInputs
    shogun::Some<shogun::CMulticlassLabels> trainOutputs,
    shogun::Some<shogun::CMulticlassLabels> testOutputs)
{
    using namespace shogun;

    // utworzenie drzewa parametrów
    auto root = some<CModelSelectionParameters>();
    // współczynnik regularyzacji
    CModelSelectionParameters* z = new CModelSelectionParameters("m_z");
    root->append_child(z);
    z->build_values(0.2, 1.0, R_LINEAR, 0.1);
    // utworzenie strategii podziału drzewa decyzyjnego
    index_t k = 3;
    CStatifiedCrossValidationSplitting* splitting =
        new CStatifiedCrossValidationSplitting(labels, k);
    // utworzenie kryterium ewaluacji dla drzewa decyzyjnego
    auto evalCriterium = some<CMulticlassAccuracy>();
    // utworzenie modelu regresji logistycznej
    auto logReg = some<CMulticlassLogisticRegression>();
    // utworzenie obiektu sprawdzianu krzyżowego
```

```
26      auto cross = some<CCrossValidation>(logReg, trainInputs, trainOutputs,
27                                          splitting, evalCriterium);
28      cross->set_num_runs(1);
29      auto modelSelection = some<CGridSearchModelSelection>(cross, root);
30      // wybranie parametrów dla modelu
31      CParameterCombination* bestParams =
32          wrap(modelSelection->select_model(false));
33      // zaaplikowanie parametrów dla modelu
34      bestParams->apply_to_machine(logReg);
35      // wyświetlenie drzewa decyzyjnego
36      bestParams->print_tree();
37      // trenowanie docelowego modelu
38      logReg->set_labels(trainOutputs);
39      logReg->train(trainInputs);
40
41      // ewaluacja modelu
42      std::cout << "-----␣Shogun␣CV␣Logistic␣-----" << std::endl;
43      std::cout << "Train:" << std::endl;
44      auto prediction = wrap(logReg->apply_multiclass(trainInputs));
45      shogunVerifyModel(prediction, trainOutputs);
46
47      std::cout << "Test:" << std::endl;
48      auto prediction2 = wrap(logReg->apply_multiclass(testInputs));
49      shogunVerifyModel(prediction2, testOutputs);
50
51      delete splitting;
52  }
```

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 [17] możliwe jest wygenerowanie przykładów jej wykorzystania także w języku C++ w folderze *examples*. Przykłady te należy zbudować za pomocą odpowiedniego skryptu Pythona zawartego w repozytorium, powodując wygenerowanie listingów kodów w docelowym języku w plikach JSON. Niestety, okazują się one obrazować użycie biblioteki w nienaturalny, proceduralnie generowany sposób, sprawiając, że przy faktycznej próbie skorzystania z API projektu, stają się one bezużyteczne. Dodatkowo, jedyna forma dokumentacji projektu ogranicza się do komentarzy w kodzie, zmuszając użytkownika do błądzenia po repozytorum w poszukiwaniu potrzebnych informacji.

Shogun jest jedną z bibliotek opisaną w podręczniku [3], wprowadzającym czytelnika zarówno do podstawowych funkcjonalności Shogun, jak i podsumowującej podstawy teorii uczenia maszynowego w kontekście ich zastosowania. Większość z przykładów realizacji poszczególnych typów modeli w tej książce posiada przedstawione główne fragmenty listingów dla biblioteki Shogun. Warto jednak zaznaczyć, że różnią się one od przykładów generowanych przez skrypt budujący, obrazując wykorzystanie faktycznie udostępnianego API biblioteki. W toku pracy nad niniejszym rozdziałem, książka ta okazała się jedynym wartościowym źródłem wiedzy na jego temat.

Chapter 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 optymalizacji liniowej i nieliniowej (w związku z czym posiada ona część funkcjonalności biblioteki do algebry liniowej), maszyny jądrowe (np. maszyna wektorów nośnych) i sieci neuronowe. [18] Podmiotami udostępniającymi bibliotekę jest Uniwersytet w Kopenhagdze w Danii, oraz Instytut Neuroinformatyki z Ruhr-Universitat Bochum w Niemczech.

5.2. Formaty źródeł danych

Biblioteka operuje na własnych reprezentacjach macierzy i wektorów, które tworzone są poprzez opakowywanie surowych tablic za pomocą specjalnych adapterów, jak np. *remora::dense_matrix_adaptor<>()* lub za pomocą kontenerów biblioteki standardowej C++ i funkcji *createDataFromRange()*. Mechanizm ten jest identyczny jak w przypadku pozostałych z omawianych bibliotek, co daje użytkownikowi dużą dowolność co do sposobu przechowywania danych i mechanizmu ich odczytywania. Posiada ona także dedykowany parser dla plików w formacie CSV, lecz zakłada on obecność w pliku jedynie danych numerycznych. Do jego użycia należy użyć klasy kontenera *ClassificationDataset* lub *RegressionDataset* oraz metody *importCSV* która zapisuje odczytane dane do wcześniej wspomnianego obiektu poprzez mechanizm zwracania przez parametr. Jeden z argumentów funkcji określa która z kolumn zawiera zmienną decyzyjną, dzięki czemu biblioteka jest w stanie od razu oddzielić dane wejściowe od kolumny oczekiwanych wartości. Artykuł [19] dostępny na platformie GitHub pokazuje także, jak pobrać dane w postaci formatu CSV z internetu z pomocą API biblioteki *curl*, i przetworzyć je do formy akceptowanej przez Shark-ML. W aktualnej wersji biblioteki znalazły się także wbudowane funkcje pobierania danych współpracujące z protokołem HTTP. Listing 5.1 ukazuje jak odczytać dane z pliku .csv znajdującego się na dysku użytkownika.

Listing 5.1. Odczytanie danych z pliku CSV.

```
#pragma once

#include <inc/shark/printEvaluation.hpp>
#include <shark/Data/Dataset.h>
#include <shark/Data/DataDistribution.h>
#include <shark/Data/Csv.h>
#include <shark/Data/SparseData.h>

template<typename DatasetType>
inline DatasetType sharkReadCsvData(std::string filePath)
{
    using namespace shark;

    DatasetType trainData;
    importCSV(trainData, filePath.c_str(), LAST_COLUMN);
    return trainData;
}
```

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

Listing 5.2. Sposób opakowywania danych do przetwarzania przez Shark-ML [3].

```
// 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 normalizatora, 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 5.3 pokazuje przykład wstępnego przetwarzania danych z wykorzystaniem normalizacji.

Listing 5.3. Wstępne przetwarzanie danych do uczenia [19].

```
#pragma once

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

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

5.3.2. Redukcja wymiarowości

5.3.2.1. Analiza składowych głównych

Algorytm redukcji wymiarowości przez analizę składowych głównych 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 analiza dyskryminacyjna (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.

Listing 5.5. Przykład redukcji zestawu danych z wykorzystaniem modelu LDA [3].

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

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 wprowadzeniu 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 średniej statystycznej gradientu (ang. *Statistic Averagte Gradient, SAG*). W przypadku bardziej skompilowanych modeli 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 załadowania uzyskanych wag do modelu regresji. Parametry modelu możliwe są do odczytania z wykorzystaniem metod *offset()* i *matrix()* lub metody *parameterVector()*. Na listingu 5.6 ukazane zostało wykorzystanie podejścia iteracyjnego, natomiast listing 5.7 przedstawia metodę analityczną.

Listing 5.6. Przykład regresji liniowej z wykorzystaniem optymalizatora spadku gradientowego.

```
1  #pragma once
2
3  #include <iostream>
4  #include <inc/shark/printEvaluation.hpp>
5  #include <shark/Algorithms/Trainers/LinearRegression.h>
6  #include <shark/Models/LinearModel.h>
7  #include <shark/ObjectiveFunctions/Loss/SquaredLoss.h>
8
9  inline void sharkLinear(const shark::RegressionDataset& trainData,
10                          const shark::RegressionDataset& testData)
11 {
12     using namespace shark;
13
14     // przygotowanie modelu i trenera
15     LinearModel<> model;
16     LinearRegression trainer;
17     //trening
18     trainer.train(model, trainData);
19     // ewaluacja
20     std::cout << "-----␣Shark␣Linear␣-----" << std::endl;
21     std::cout << "Train␣data:" << std::endl;
22     auto predictions = model(trainData.inputs());
23     printSharkModelEvaluation(
24         trainData.labels(), predictions);
25
26     std::cout << "Test␣data:" << std::endl;
```

```
27      predictions = model(testData.inputs());
28      printSharkModelEvaluation(
29          testData.labels(), predictions);
30  }
```

Listing 5.7. Przykład regresji liniowej z wykorzystaniem trenera analitycznego [3].

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

5.4.2. Regresja logistyczna

Mechanizm regresji logistycznej dostępny w bibliotece Shark-ML z natury rozwiązuje problem klasyfikacji dla dwóch klas. Istnieje jednak możliwość przygotowania wielu klasyfikatorów, w liczbie wyrażonej wzorem:

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

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

wieloklasowej, natomiast listing 5.9 pokazuje sposób utworzenia prostego modelu dla problemu binarnego.

Listing 5.8. Przykład funkcji tworzącej model wieloklasowej regresji logistycznej [3].

```
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 5.9. Przykład prostej binarnej regresji logistycznej.

```
#pragma once

#include <iostream>

#define SHARK_CV_VERBOSE 1
#include <inc/shark/printEvaluation.hpp>
#include <shark/Algorithms/Trainers/LogisticRegression.h>
#include <shark/Data/Dataset.h>
#include <shark/Models/Classifier.h>
```

```
10 #include <iostream>
11
12 inline void sharkLogistic(const shark::ClassificationDataset& trainData,
13                           const shark::ClassificationDataset& testData)
14 {
15     using namespace shark;
16
17     // utworzenie modelu
18     std::cout << "Start␣logistic\n";
19     LinearClassifier<RealVector> logisticModel;
20     LogisticRegression<RealVector> trainer;
21     // trening
22     std::cout << "Training\n";
23     trainer.train(logisticModel, trainData);
24     // ewaluacja
25     std::cout << "-----Shark␣Logistic␣Regression-----" << std::endl;
26     std::cout << "Train␣data␣model␣evaluation:" << std::endl;
27     auto predictions = logisticModel(trainData.inputs());
28     printSharkModelEvaluation(
29         trainData.labels(), predictions);
30
31     std::cout << "Test␣data␣model␣evaluation:" << std::endl;
32     predictions = logisticModel(testData.inputs());
33     printSharkModelEvaluation(
34         testData.labels(), predictions);
35 }
```

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 jeden z typów modeli jądrowych (ang. *kernel model*). Opiera się ona na wykonaniu regresji liniowej w przestrzeni cech określonych przez wykorzystane jądro. Podobnie jak w przypadku regresji logistycznej, API biblioteki umożliwia wykonanie klasyfikacji dla przypadku binarnego, natomiast rozwiązanie przy jej użyciu problemu wieloklasowego wymaga kombinacji instancji dychotomicznych maszyn wektorów nośnych w model złożony, czego można dokonać przy pomocy klasy *OneVersusOneClassifier* oraz liczbie 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 $[0; 1]$;
- regularyzacja - podawana jako liczba rzeczywista w konstruktorze *CSvmTrainer*, domyślnie maszyna wektorów nośnych używa kary w postaci normy L1 za przekroczenie docelowej granicy;
- bias - flaga binarna (bool) określająca czy model ma używać wyrazu wolnego (ang. *bias*), podawana w konstruktorze *CSvmTrainer*;
- *sparsify* - parametr określający czy model ma zachować wektory, które nie są nośne, dostępny przez metodę *sparsify()* trenera;
- minimalna dokładność zakończenia nauczania - pozwala wyspecyfikować precyzję modelu, jest dostępna jako pole struktury zwracane przez metodę *stoppingCondition()* klasy trenera;
- wielkość cache - ustawiana za pomocą funkcji *setCacheSize()* trenera;

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

Listing 5.10. Przykład maszyny wektorów nośnych dla problemu binarnego.

```
1  #pragma once
2
3  #define SHARK_CV_VERBOSE 1
4  #include <inc/shark/printEvaluation.hpp>
5  #include <shark/Algorithms/KMeans.h>
6  #include <shark/Algorithms/Trainers/CSvmTrainer.h>
7  #include <shark/Data/Dataset.h>
8  #include <shark/Models/Classifier.h>
9  #include <shark/Models/Kernels/GaussianRbfKernel.h>
10 #include <shark/ObjectiveFunctions/Regularizer.h>
11
12 inline void sharkSVM(const shark::ClassificationDataset& trainData,
13                      const shark::ClassificationDataset& testData)
14 {
15     using namespace shark;
16
17     // utworzenie jądra
18     double gamma = 0.16;
19     GaussianRbfKernel<> kernel(gamma);
20     KernelClassifier<RealVector> svm;
21     double regularization = 1;
22     bool bias = true;
23     // utworzenie i konfiguracja modelu
24     CSvmTrainer<RealVector> trainer(
25         &kernel, regularization, bias);
26     trainer.sparsify() = false;
27     // trening
```

```
28      trainer.train(svm, trainData);
29      // ewaluacja
30      std::cout << "-----Shark␣SVM-----" << std::endl;
31      std::cout << "Train␣data:" << std::endl;
32      auto predictions = svm(trainData.inputs());
33      printSharkModelEvaluation(
34          trainData.labels(), predictions);
35
36      std::cout << "Test␣data:" << std::endl;
37      predictions = svm(testData.inputs());
38      printSharkModelEvaluation(
39          testData.labels(), predictions);
40  }
```

Listing 5.11. Przykład maszyny wektorów nośnych dla problemu wieloklasowego [3].

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

5.4.4. Algorytm K najbliższych sąsiadów

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

Listing 5.12. Przykład utworzenia klasyfikatora kNN.

```
1  #pragma once
2
3  #define SHARK_CV_VERBOSE 1
4  #include <inc/shark/printEvaluation.hpp>
5  #include <shark/Algorithms/KMeans.h>
6  #include <shark/Algorithms/NearestNeighbors/TreeNearestNeighbors.h>
7  #include <shark/Data/Dataset.h>
8  #include <shark/Models/Classifier.h>
9  #include <shark/Models/NearestNeighborModel.h>
10 #include <shark/Models/Trees/KDTree.h>
11 #include <iostream>
12
13 inline void sharkKNN(const shark::ClassificationDataset& trainData,
14                      const shark::ClassificationDataset& testData)
15 {
16     using namespace shark;
17
18     // utworzenie i konfiguracja drzewa oraz algorytmu
19     KDTree<RealVector> tree(trainData.inputs());
20     TreeNearestNeighbors<RealVector, unsigned int> algorithm(
21         trainData, &tree);
22
23     // konfiguracja modelu
24     const unsigned int K = 2; // ilość sąsiadów dla algorytmu kNN
25     NearestNeighborModel<RealVector, unsigned int> KNN(&algorithm, K);
26
27     // ewaluacja modelu
28     std::cout << "-----Shark␣KNN-----" << std::endl;
29     std::cout << "Train␣data␣:" << std::endl;
30     auto predictions = KNN(trainData.inputs());
31     printSharkModelEvaluation(
32         trainData.labels(), predictions);
33
34     std::cout << "Test␣data:" << std::endl;
35     predictions = KNN(testData.inputs());
36     printSharkModelEvaluation(
37         testData.labels(), predictions);
38 }
```

5.4.5. Algorytm zbiorowy

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

Listing 5.13. Utworzenie modelu algorytmu złożonego losowego lasu [3].

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

5.4.6. Sieć neuronowa

Skonstruowanie sieci neuronowej w bibliotece Shark-ML wykorzystuje pewne mechanizmy oferowane przez klasę *LinearModel<>*. Pozwala ona na określenie typu i liczby wejść, wyjść, oraz zastosowania wyrazu wolnego. Każda warstwa składa się z pojedynczego obiektu modelu liniowego, gdzie liczba 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 [20]. 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 wagami losowymi 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 wsadowe porcje danych, wykonując na nich krok optymalizatora. Po zakończeniu uczenia, należy skonfigurować obiekt modelu przekazując mu wagi ustalone przez optymalizator, uzyskując w ten sposób gotową instancję nauczonej sieci neuronowej. Listing 5.14 przedstawia kod realizujący cały proces, stanowiący przykład z podręcznika [3].

Listing 5.14. Przykład sieci neuronowej o dwóch warstwach ukrytych do zadania klasyfikacji [3].

```
1  #pragma once
2
3  #include <iostream>
4  #include <inc/shark/printEvaluation.hpp>
5  #include <shark/Algorithms/GradientDescent/SteepestDescent.h>
6  #include <shark/Models/ConcatenatedModel.h>
7  #include <shark/Models/LinearModel.h>
8  #include <shark/Data/Dataset.h>
9  #include <shark/ObjectiveFunctions/ErrorFunction.h>
10 #include <shark/ObjectiveFunctions/Regularizer.h>
11 #include <shark/ObjectiveFunctions/Loss/CrossEntropy.h>
12
13 inline void sharkNN(const shark::ClassificationDataset& trainData,
14                     const shark::ClassificationDataset& testData)
15 {
16     using namespace shark;
17
18     // zdefiniowanie warstw sieci
19     using DenseTanhLayer = LinearModel<RealVector, TanhNeuron>;
20     using DenseLogisticLayer = LinearModel<RealVector, LogisticNeuron>;
21     DenseTanhLayer layer1(inputDimension(trainData), 5, true);
22     DenseTanhLayer layer2(5, 5, true);
23     DenseLogisticLayer output(5, numberOfClasses(trainData), true);
24     // połączenie warstw
25     auto network = layer1 >> layer2 >> output;
26     // utworzenie i konfiguracja funkcji straty
27     CrossEntropy<unsigned int, RealVector> loss;
28     ErrorFunction<> error(trainData, &network, &loss, true);
29     //TwoNormRegularizer<> regularizer(error.numberOfVariables());
30     double weightDecay = 0.01;
31     //error.setRegularizer(weightDecay, &regularizer);
32     error.init();
33     // inicjalizacja wag sieci
34     initRandomNormal(network, 0.001);
```

```
35      // utworzenie i konfiguracja optymalizatora
36      SteepestDescent<> optimizer;
37      optimizer.setMomentum(0.5);
38      optimizer.setLearningRate(0.1);
39      optimizer.init(error);
40      // przeprowadzenie procesu uczenia
41      std::size_t epochs = 1000;
42      std::size_t iterations = trainData.numberOfBatches();
43      // pętla przechodząca poszczególne epoki
44      for (std::size_t epoch = 0; epoch != epochs; ++epoch)
45      {
46          double avgLoss = 0.0;
47          // pętla operująca na pojedynczych batch'ach
48          for (std::size_t i = 0; i != iterations; ++i)
49          {
50              // wykonanie kroku optymalizatora
51              optimizer.step(error);
52              // zapisanie częściowej wartości funkcji straty
53              //if (i % 100 == 0)
54              //{
55                  avgLoss += optimizer.solution().value;
56              //}
57          }
58          // wyliczenie średniej wartości funkcji straty
59          avgLoss /= iterations;
60          std::cout << "Epoch " << epoch << " | Avg. loss " << avgLoss
61                    << std::endl;
62      }
63      // konfiguracja modelu docelowego
64      network.setParameterVector(optimizer.solution().point);
65      std::cout << "In Shape=" << network.inputShape() << std::endl;
66      std::cout << "Out Shape=" << network.outputShape() << std::endl;
67      //Classifier<ConcatenatedModel<RealVector>> model(network);
68
69      // walidacja
70      std::cout << "-----Shark Neural -----" << std::endl;
71      std::cout << "Train data:" << std::endl;
72      std::cout << "trainData.numberOfBatches() = " <<
73              trainData.numberOfBatches() << std::endl;
74      auto predictions = network(trainData.inputs());
75      std::cout << "predictions.numberOfBatches() = " <<
76              predictions.numberOfBatches();
77      printSharkModelEvaluation(
78          trainData.labels(), predictions);
79
80      std::cout << "Test data:" << std::endl;
81      predictions = network(testData.inputs());
82      printSharkModelEvaluation(
83          testData.labels(), predictions);
84  }
```

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 [21]:

- **średni błąd bezwzględny** - realizowany za pomocą klasy *AbsoluteLoss*;
- **błąd średniokwadratowy** - realizowany za pomocą klasy *SquaredLoss*;
- **błąd typu zero-jeden** - realizowany za pomocą klasy *ZeroOneLoss*;
- **błąd dyskretny** - realizowany za pomocą klasy *DiscreteLoss*;
- **entropia krzyżowa** - realizowana za pomocą klasy *CrossEntropy*;
- **kawałkami liniowa funkcja straty** - realizowany za pomocą klasy *HingeLoss*;
- **średniokwadratowy błąd kawałkami liniowej funkcji straty** - realizowany za pomocą klasy *SquaredHingeLoss*;
- **kawałkami liniowa funkcja straty typu epsilon** - realizowany za pomocą klasy *EpsilonHingeLoss*;
- **średniokwadratowy błąd kawałkami liniowej funkcji straty typu epsilon** - realizowany za pomocą klasy *SquaredEpsilonHingeLoss*;
- **funkcja straty Hubera** - realizowana za pomocą klasy *HuberLoss*;
- **funkcja straty Tukeya** - realizowana za pomocą klasy *TukeyBiweightLoss*.

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

Listing 5.15. Użycie funkcji straty na przykładzie błędu średniokwadratowego

```
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 skorygowane R^2

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

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

```
using namespace shark;

// [...]

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

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

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

5.5.3. Pole pod wykresem krzywej charakterystycznej odbiornika

Pole pod wykresem krzywej charakterystycznej odbiornika stanowi jedną z często wykorzystywanych metryk poprawności predykcji modelu, w związku z czym nie mogło jej zabraknąć w bibliotece Shark-ML. Jest ona dostępna za pośrednictwem klasy *NegativeAUC*, wykorzystywanej w taki sam sposób jak pozostałe omówione wcześniej funkcje straty. W przeciwieństwie do standardowego podejścia, wspomniana klasa oblicza dopełnienie pola pod wykresem funkcji ROC, aby umożliwić wykorzystanie jej jako minimalizowanego celu w procesie uczenia. Listing 5.17 przedstawia sposób obliczenia wartości wymienionej metryki.

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

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <shark/LinAlg/Base.h>

std::vector<shark::RealVector> repackToRealVectorRange(
                const auto& dataContainer)
{
    using namespace shark;

    // przepakowanie danych z typu unsigned int na typ RealVector
    std::vector<RealVector> v;
    std::for_each(dataContainer.elements().begin(),
                  dataContainer.elements().end(),
                  [&](const auto e){
```

```
16                          RealVector rv(1);
17                          rv(0) = static_cast<double>(e);
18                          v.emplace_back(rv); });
19      return v;
20  }
21
22  inline void printSharkModelEvaluation(
23      const shark::Data<unsigned int>& labels,
24      const auto& predictions)
25  {
26      using namespace shark;
27
28      // przygotowanie solvera pola pod wykresem ROC
29      constexpr bool invert = false;
30      NegativeAUC<unsigned int, RealVector> auc(invert);
31
32      // przepakowanie danych
33      auto predVec = repackToRealVectorRange(predictions);
34      auto predData = createDataFromRange(predVec);
35      // obliczenie AUC ROC
36      auto roc = auc(labels, predData);
37      std::cout << "ROC:␣" << (-1 * roc) << std::endl << std::endl;
38  }
```

5.5.4. K-krotny sprawdzian krzyżowy

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 K-krotnego sprawdzianu krzyżowego w wymaga wykorzystania trzech klas. Pierwszą z nich stanowi *CVFolds*, której zadaniem jest przechowanie zestawu danych podzielonego na odpowiednią liczbę 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 wyznaczenie wartości hiperparametrów dla modelu. Wynikiem procesu jest uzyskanie najlepszego zestawu hiperparametrów do procedury uczenia - użytkownik musi zawołać metodę *step()* klasy *GridSearch* tylko jeden raz. Listing 5.18 przedstawia przykład zawarty w podręczniku [3], w którym autor przedstawia proces wykorzystania powyższych klas na własnoręcznie zaimplementowanym modelu regresji wielomianowej.

Listing 5.18. Przykład realizacji sprawdzianu krzyżowego K-fold w Shark-ML [3].

```
1  using namespace shark;
2
3  // [...]
4
5  // przetworzenie danych uczących
6  const unsigned int num_folds = 5;
7  CVFolds<RegressionDataset> folds =
8      createCVSameSize<RealVector, RealVector>(train_data, num_folds);
9
10 // przygotowanie parametrów dla docelowego modelu
```

```
11 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 językowi C++, w związku z czym dużo łatwiej dostępne są wątki społecznościowe i artykuły omawiające realizację różnorodnych typów modeli z jej użyciem, oraz oferując przykładowy kod źródłowy. Strona główna projektu [22] posiada rozbudowaną, przejrzystą i opatrzoną przykładami dokumentację, znacznie ułatwiając korzystanie z dostępnego API.

Chapter 6

Biblioteka Dlib

6.1. Wprowadzenie

Jest to biblioteka do uczenia maszynowego napisana w nowoczesnym C++, o zastosowaniu przemysłowym oraz naukowym [23]. Podobnie jak poprzednio omawiane biblioteki, posiada ona otwarte źródło na licencji Boost Software Licence [24]. Do dziedzin wykorzystujących wyżej wspomnianą bibliotekę należą robotyka, systemy wbudowane, telefonia komórkowa oraz oprogramowanie 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 [3].

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

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

6.3. Metody przetwarzania i eksploracji danych

6.3.1. Normalizacja

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

Listing 6.2. Funkcja normalizująca.

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

6.3.2. Redukcja wymiarowości

6.3.2.1. Analiza składowych głównych

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

Listing 6.3. Przykład redukcji wymiarowości z użyciem metody PCA.

```
#pragma once

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

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

6.3.2.2. Liniowa analiza dyskryminacyjna

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

Listing 6.4. Przykład redukcji wymiarowości z użyciem algorytmu LDA.

```
#pragma once

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

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

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

6.3.2.3. Mapowanie Sammona

Jednym z algorytmów wyróżniających bibliotekę DLib na tle pozostałych, jest implementacja metody redukcji wymiarowości poprzez skalowanie wielowymiarowe z użyciem nieliniowego algorytmu tzw. mapowania Sammona [25]. Całość wspomnianego algorytmu implementowana jest za pomocą klasy *dlib::sammon_projection*, i ogranicza się do utworzenia jej instancji. Wykorzystując metodę należy przekazać do utworzonego obiektu za pomocą operatora wywołania funkcji wektor danych, oraz oczekiwaną liczbę wymiarów, otrzymując przekształcone dane. W związku z powyższym, funkcja realizująca redukcję z użyciem wyżej wymienionej metody sprowadza się do wykorzystania dwóch linii. Dokładny sposób jej użycia pokazano na listingu 6.5

Listing 6.5. Przykład redykcji wymiarowości z użyciem mapowania Sammona

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

6.3.3. Regularyzacja L2

Biblioteka Dlib posiada funkcję trenera pozwalającą na użycie brzegowej regresji, realizującą regularyzację L2, o nazwie *rr_trainer* dla regresji liniowej oraz *krr_trainer*

dla regresji nieliniowej. Przykłady zastosowania poszczególnych wersji tego algorytmu zostały przedstawione w sekcji omawiającej model regresji liniowej, oraz grzbietowej regresji jądrowej.

6.4. Modele uczenia maszynowego

6.4.1. Regresja liniowa

Biblioteka Dlib posiada pośrednią realizację modelu regresji liniowej. Wykorzystuje ona technikę grzbietowej regresji jądrowej, przekazując jądro liniowe. Następnie przeprowadzane jest uczenie, zapisując docelowy model w postaci funkcji decyzyjnej. Listing 6.6 przedstawia szczegóły powyższego mechanizmu.

Listing 6.6. Przykład regresji liniowej w Dlib.

```
#pragma once

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

#include <inc/dlib/eval.hpp>

inline void dlibLinear(
    std::vector<dlib::matrix<double, 5, 1>> trainData,
    std::vector<dlib::matrix<double, 5, 1>> testData,
    std::vector<double> trainLabels,
    std::vector<double> testLabels)
{
    using namespace dlib;
    // utworzenie oraz konfiguracja trenera i jądra
    using linearKernel = linear_kernel<matrix<double, 5, 1>>;
    krr_trainer<linearKernel> trainer;
    trainer.set_kernel(linearKernel());
    // trening
    decision_function<linearKernel> model = trainer.train(
        trainData, trainLabels);
    // ewaluacja
    std::cout << "-----␣Dlib␣Linear␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = std::vector<double>(trainData.size());
    for (auto& sample : trainData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, trainLabels, Task::REGRESSION);
    predictions.clear();
    std::cout << "Test␣data:" << std::endl;
    for (auto& sample : testData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, testLabels, Task::REGRESSION);
}
```

6.4.2. Maszyna wektorów nośnych

W celu realizacji wieloklasowej klasyfikacji z użyciem maszyny wektorów nośnych, biblioteka Dlib oferuje klasę funkcji decyzyjnej *dlib::one_vs_one_decision_function*. Przechowuje ona wynikowy model uczenia algorytmem maszyny wektorów nośnych, zawarty w klasie *one_versus_one*, do której przesłany zostaje trener SVM. Dokładny sposób użycia został przedstawiony na listingu 6.7.

Listing 6.7. Przykład użycia maszyny wektorów nośnych w Dlib.

```
#pragma once

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

inline void dlibSVM(
    std::vector<dlib::matrix<double, 11, 1>> trainData,
    std::vector<dlib::matrix<double, 11, 1>> testData,
    std::vector<double> trainLabels,
    std::vector<double> testLabels)
{
    using namespace dlib;
    using OVOTrainer = one_vs_one_trainer<any_trainer<matrix<double, 11, 1>>>;
    using Kernel = radial_basis_kernel<matrix<double, 11, 1>>;
    // utworzenie trenera maszyny wektorów nośnych
    svm_nu_trainer<Kernel> svmTrainer;
    svmTrainer.set_kernel(Kernel(0.16));
    // utworzenie trenera klasyfikatora wieloklasowego
    OVOTrainer trainer;
    trainer.set_trainer(svmTrainer);
    // utworzenie modelu
    one_vs_one_decision_function<OVOTrainer> model =
        trainer.train(trainData, trainLabels);
    // ewaluacja
    std::cout << "-----␣Dlib␣SVM␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = std::vector<double>(trainData.size());
    for (auto& sample : trainData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, trainLabels, Task::CLASSIFICATION);
    predictions.clear();
    std::cout << "Test␣data:" << std::endl;
    for (auto& sample : testData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, testLabels, Task::CLASSIFICATION);
}
```

6.4.3. Sieci neuronowe

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

- współczynnik uczenia;
- współczynnik zmiany szybkości uczenia;
- rozmiar porcji wsadowych;
- maksymalna liczba epok.

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

Listing 6.8. Przykład sieci neuronowej w Dlib.

```
1  #pragma once
2
3  #include <dlib/dnn.h>
4  #include <dlib/matrix.h>
5
6  #include <inc/dlib/eval.hpp>
7
8  #include <vector>
9
10 inline void dlibNeural(
11     std::vector<dlib::matrix<double>> trainData,
12     std::vector<dlib::matrix<double>> testData,
13     std::vector<double> trainLabels,
14     std::vector<double> testLabels)
15 {
16     using namespace dlib;
17     // zdefiniowanie architektury sieci
18     using Architecture = loss_mean_squared<fc <1,
19                              htan<fc<5,
20                              htan<fc<5,
21                              input<matrix<double>>>>>>>>;
22     // utworzenie sieci
23     Architecture model;
24     // utworzenie i konfiguracja algorytmu optymalizacji
```

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

6.4.4. Brzegowa regresja jądrowa

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

Listing 6.9. Przykład użycia brzegowej regresji jądra w Dlib.

```
1   #pragma once
2
3   #include <dlib/matrix.h>
4   #include <dlib/svm_threaded.h>
5
6   #include <inc/dlib/eval.hpp>
7
8   #include <vector>
9
10  inline void dlibKRR(
11      std::vector<dlib::matrix<double>> trainData,
12      std::vector<dlib::matrix<double>> testData,
13      std::vector<double> trainLabels,
14      std::vector<double> testLabels)
15  {
16      using namespace dlib;
17
18      using OVOTrainer = one_vs_one_trainer<any_trainer<double>>;
19      using Kernel = radial_basis_kernel<double>;
20      // utworzenie trenera maszyny wektorów nośnych
```

```
21      krr_trainer<Kernel> krrTrainer;
22      krrTrainer.set_kernel(Kernel(0.1));
23      // utworzenie trenera klasyfikatora wieloklasowego
24      OVOTrainer trainer;
25      trainer.set_trainer(krrTrainer);
26      // utworzenie modelu
27      one_vs_one_decision_function<OVOTrainer> model =
28          trainer.train(trainData, trainLabels);
29      // ewaluacja
30      std::cout << "-----␣Dlib␣KRR␣-----" << std::endl;
31      std::cout << "Train␣data:" << std::endl;
32      auto predictions = std::vector<double>(trainData.size());
33      for (auto& sample : trainData)
34      {
35          predictions.emplace_back(model(sample));
36      }
37      dlibEval(predictions, trainLabels, Task::CLASSIFICATION);
38      predictions.clear();
39      std::cout << "Test␣data:" << std::endl;
40      for (auto& sample : testData)
41      {
42          predictions.emplace_back(model(sample));
43      }
44      dlibEval(predictions, testLabels, Task::CLASSIFICATION);
45  }
```

6.5. Metody analizy modeli

6.5.1. Pole pod wykresem krzywej charakterystycznej odbiornika

Biblioteka Dlib posiada implementację funkcji wyznaczającej krzywą ROC, jednak wymaga ona pewnego przetwarzania danych przed i po jej użyciu. W celu jej zastosowania należy podzielić dane sklasyfikowane prawidłowo i nieprawidłowo. Wynikiem funkcji jest wektor zawierający współrzędne poszczególnych punktów krzywej charakterystycznej odbiornika, które pozwalają na narysowanie wykresu na płaszczyźnie. Obliczenie wartości pola pod wykresem należy dokonać ręcznie, wykorzystując np. jedną z metod całkowania numerycznego. Listing 6.10 przedstawia funkcję wyznaczającą wartości predykcji modelu, w tym obliczenie wartości pola pod wykresem krzywej charakterystycznej odbiornika dla zadania klasyfikacji, wykonując całkowanie numeryczne z użyciem metody trapezów.

Listing 6.10. Obliczenie pola pod wykresem funkcji ROC dla Dlib.

```
1   #pragma once
2
3   #include <cmath>
4   #include <dlib/statistics.h>
5
6   enum class Task
7   {
8       CLASSIFICATION,
9       REGRESSION
10  };
```

```
11
12 inline void dlibEval(std::vector<double> predictions,
13                       std::vector<double> labels,
14                       Task task)
15 {
16     using namespace dlib;
17
18     if (task == Task::CLASSIFICATION)
19     {
20         // przygotowanie kontenerów na podzielone dane
21         std::vector<double> correct;
22         std::vector<double> incorrect;
23         // przygotowanie wartości detektora
24         constexpr double positiveDetectionScore = 0.75;
25         constexpr double negativeDetectionScore = 0.25;
26         // podział danych
27         for (int i = 0; i < predictions.size() && i < labels.size(); ++i)
28         {
29             if (predictions[i] == labels[i])
30                 correct.emplace_back(positiveDetectionScore);
31             else
32                 incorrect.emplace_back(negativeDetectionScore);
33         }
34         // obliczenie krzywej roc
35         auto roc = compute_roc_curve(correct, incorrect);
36         // obliczenie pola pod wykresem roc
37         double aucRoc = 0.0;
38         for (int i = 0; i < roc.size() - 1; i++)
39         {
40             if (roc[i+1].false_positive_rate != 0.0)
41             {
42                 aucRoc += (roc[i].true_positive_rate + roc[i+1].true_positive_rate)
43                         * (roc[i+1].false_positive_rate - roc[i].false_positive_rate) / 2;
44             }
45         }
46         std::cout << "AUC␣ROC:␣" << aucRoc << std::endl;
47     }
48     else
49     {
50         // obliczenie sumy kwadratów różnic
51         auto sum = 0.0;
52         for (int i = 0; i < predictions.size() && i < labels.size(); ++i)
53         {
54             sum += pow(labels[i] - predictions[i], 2);
55         }
56         // obliczenie błędu średniokwadratowego
57         auto mse = sqrt(sum / predictions.size());
58         std::cout << "MSE:␣" << mse << std::endl;
59     }
60 }
```

6.5.2. K-krotny sprawdzian krzyżowy

Przeprowadzenie sprawdzianu krzyżowego z wykorzystaniem biblioteki Dlib stanowi złożony proces. Jest on realizowany wielofazowo, rozpoczynając od zdefiniowania własnej funkcji obliczającej interesujący użytkownika wynik metryki sprawdzianu, w

oparciu o funkcję *dlib::cross_validate_regression_trainer()*. Zewnętrzna część ustalania docelowych wartości hiperparametrów dokonywana jest przez funkcję *dlib::find_min_global()* przyjmującą adres stworzonej funkcji optymalizacyjnej, kontenery przechowujące informacje o minimalnych i maksymalnych dopuszczalnych wartościach poszczególnych hiperparametrów, oraz liczbę dozwolonych wywołań funkcji optymalizacyjnej. Aby odczytać wynikowe wartości, należy sięgnąć do kolejnych pól składowej x zwróconej przez funkcję *find_min_global()* struktury. Szczegóły zostały zaprezentowane w oparciu o przykład z podręcznika [3] na listingu 6.11.

Listing 6.11. Przykład realizacji sprawdzianu krzyżowego.

```
#pragma once
#include <dlib/global_optimization.h>
#include <dlib/matrix.h>
#include <dlib/svm.h>
#include <cmath>

inline void dlibCrossValidate(std::vector<dlib::matrix<double>> data,
                              std::vector<double> labels)
{
    using namespace dlib;
    // podział danych
    auto dataSplit = data.begin() + data.size() * 0.8;
    auto trainData = std::vector<matrix<double>>(
        data.begin(), dataSplit);
    auto testData = std::vector<matrix<double>>(
        dataSplit, data.end());
    auto labelSplit = labels.begin() + labels.size() * 0.8;
    auto trainLabels = std::vector<matrix<double>>(
        labels.begin(), labelSplit);
    auto testLabels = std::vector<matrix<double>>(
        labelSplit, labels.end());
    // utworzenie funkcji sprawdzianu krzyżowego
    auto crossValidationScore = [&](const double gamma, const double c,
                                    const double degreeIn)
    {
        using namespace dlib;

        auto degree = std::floor(degreeIn);
        // zdefiniowanie jądra
        using Kernel = polynomial_kernel<double>;
        // przygotowanie i konfiguracja trenera
        svr_trainer<Kernel> trainer;
        trainer.set_kernel(Kernel(gamma, c, degree));
        // obliczenie metryk sprawdzianu krzyżowego
        matrix<double> result = cross_validate_regression_trainer(
            trainer, trainData, trainLabels, 10);
        // zwrócenie metryki błędu średniokwadratowego
        return result(0, 0);
    }
    // przeprowadzenie sprawdzianu
    auto result = find_min_global(
        crossValidationScore,
        {0.01, 1e-8, 5}, // wartości minimalne
        {0.1, 1, 15},    // wartości maksymalne
        max_function_calls(50));
    // odczytanie ustalonych wartości hiperparametrów
```

```
47      auto gamma = result.x(0);
48      auto c = result.x(1);
49      auto degree = result.x(2);
50      // utworzenie modelu w oparciu o ustalone hiperparametry
51      using Kernel = polynomial_kernel<double>;
52      svr_trainer<Kernel> trainer;
53      trainer.set_kernel(Kernel(gamma, c, degree));
54      auto model = trainer.train(trainData, trainLabels);
55      // ewaluacja
56      std::cout << "-----␣Dlib␣CrossValidated␣SVM␣-----" << std::endl;
57      std::cout << "Train␣data:" << std::endl;
58      auto predictions = std::vector<double>(trainData.size());
59      for (auto& sample : trainData)
60      {
61          predictions.emplace_back(model(sample));
62      }
63      dlibEval(predictions, trainLabels, Task::REGRESSION);
64      predictions.clear();
65      std::cout << "Test␣data:" << std::endl;
66      for (auto& sample : testData)
67      {
68          predictions.emplace_back(model(sample));
69      }
70      dlibEval(predictions, testLabels, Task::REGRESSION);
71  }
```

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 pracy [3]. Niestety, większość forów społecznościowych skupia się na pracy z Dlib z poziomu interfejsu języka Python, co może utrudnić szukanie rozwiązań dla specyficznych przypadków. Warto wspomnieć, że oprócz funkcjonalności uczenia maszynowego, Dlib realizuje także inne zadania, jak np. networking, co sprawia, że nawigacja po stronie projektu jest lekko utrudniona przez obecność potencjalnie nieinteresujących użytkownika elementów.

Chapter 7

Zestawienie zbiorcze i podsumowanie

7.1. Oferowane funkcjonalności

Omówione w ramach niniejszej pracy biblioteki różnią się typem i liczbą oferowanych funkcjonalności. Tabela 7.1 zbiorczo podsumowuje poszczególne omówione w poprzednich rozdziałach aspekty.

Funkcjonalność	Biblioteka		
	Shogun	Shark-ML	Dlib
Odczyt danych	std::vector, wsparcie dla formatu CSV	surowe tablice, wsparcie dla formatu CSV, wsparcie dla HTTP	std::vector, wsparcie dla formatu CSV
Normalizacja	min-max	przedział jednostkowy, jednostkowa wariancja, zerowa średnia i wybrana wariancja	standaryzacja
Redukcja wymiarowości	PCA, Kernel PCA, MDS, IsoMap, ICA, Factor analysis, t-SNE	PCA, Liniowa analiza dyskryminacyjna	PCA, Liniowa analiza dyskryminacyjna, Mapowanie Sammona
Regularyzacja	L1 i L2 automatyczna	L1 i L2	L2
Regresja liniowa	Tak	Tak	Tak

Regresja logistyczna	Tak	Tak	Nie
Maszyna wektorów nośnych	Tak	Tak	Tak
Algorytm K-najbliższych sąsiadów	Tak	Tak	Nie
Algorytm zbiorowy	Wzmacnianie gradientu, losowy las	Losowy las	Nie
Sieć neuronowa	Tak	Tak	Tak
Jądrowa regresja grzbietowa	Nie	Nie	Tak
Błąd średniokwadratowy	Tak	Tak	Nie
Średni błąd bezwzględny	Tak	Tak	Nie
Błąd typu zero-jeden	Nie	Tak	Nie
Błąd dyskretny	Nie	Tak	Nie
Entropia krzyżowa	Nie	Tak	Nie
Kawałkami liniowa funkcja straty	Nie	Tak	Nie
Średnio-kwadratowy błąd typu kawałkami liniowej funkcji straty	Nie	Tak	Nie
Kawałkami liniowa funkcja straty typu epsilon	Nie	Tak	Nie
Średnio-kwadratowy błąd kawałkami liniowej funkcji straty typu epsilon	Nie	Tak	Nie

Funkcja straty Hubera	Nie	Tak	Nie
Funkcja straty Tukeya	Nie	Tak	Nie
Logarytmiczna funkcja straty	Tak	Nie	Nie
Metryka R^2	Tak	Tak	Nie
Dokładność	Tak	Nie	Tak
Pole pod wykresem ROC	Tak	Tak	Tak*
Sprawdzian krzyżowy k-krotny	Tak	Tak	Tak

Table 7.1. Zbiorcze porównanie funkcjonalności bibliotek.

* - konieczność przeliczenia pola na podstawie uzyskanych punktów pomiarowych wykresu funkcji.

Analizując dane zebrane w tabeli 7.1 można zauważyć, że biblioteki Shogun oraz Shark-ML są bardzo zbliżone do siebie typem i liczbą dostępnych metod uczenia maszynowego, jednak Shark-ML posiada więcej rodzajów błędów możliwych do wykorzystania jako funkcje strat, pozwalając na większą swobodę względem dostosowywania procesu uczenia. Najmniejszą liczbą funkcjonalności charakteryzuje się biblioteka Dlib, posiadająca jedynie podzbiór algorytmów dostępnych w innych projektach. Ponadto, posiada ona bardzo ubogie możliwości analizy sprawności modeli, wymagając od użytkownika napisania własnych procedur przetwarzania uzyskanych danych, jak np. procedura obliczania pola pod wykresem krzywej charakterystycznej odbiornika.

7.2. Porównanie wyników dla zadanych przykładów

W celu przetestowania sprawności poszczególnych bibliotek, użyto ich do stworzenia i ewaluacji modeli regresji liniowej oraz maszyny wektorów nośnych dla zestawów danych opisanych w rozdziale 3. Dla każdego modelu użyto 80% zestawu danych w celu treningu, oraz pozostałe 20% jako dane testowe. Tabela 7.2 przedstawia zbiorcze zestawienie uzyskanych wyników ewaluacji. Wyjście programów zostało zaprezentowane na rysunkach 7.1, 7.2 oraz 7.3.

Biblioteka	Metoda uczenia maszynowego	
	Regresja liniowa	Maszyna wektorów nośnych
Shogun	Dane treningowe: MSE = 0,903031; R^2 = 0,407058 Dane testowe: MSE = 1,26897; R^2 = -0.127851	Dane treningowe: AUC ROC: 1 Dane testowe: AUC ROC: 0,5
Shark	Dane treningowe: MSE = 0,38728; R^2 = 0,745707 Dane testowe: MSE = 0,449118; R^2 = 0,60082	Dane treningowe: AUC ROC: 1 Dane testowe: AUC ROC: 0,5
Dlib	Dane treningowe: MSE = 3,08953; Dane testowe: MSE = 1,33014;	Dane treningowe: AUC ROC: 1 Dane testowe: AUC ROC: 1

Table 7.2. Zestawienie uzyskanych wyników poprawności modeli.

```
----- Shogun Linear -----
Train data:
MSE = 0.903031
R^2 = 0.407058

Test data:
MSE = 1.26897
R^2 = -0.127851

----- Shogun SVM -----
Train:
AUC ROC = 1
Test:
AUC ROC = 0.5
```

Figure 7.1. Wynik działania programu dla biblioteki Shogun

```
----- Shark Linear -----
Train data:
MSE: 0.38728
R^2: 0.745707
Test data:
MSE: 0.449118
R^2: 0.600826
Test1
-----Shark SVM-----
Train data:
OpenBLAS Warning : Detect
.
OpenBLAS Warning : Detect
.
OpenBLAS Warning : Detect
.
OpenBLAS Warning : Detect
.
ROC: 1

Test data:
ROC: 0.5
```

Figure 7.2. Wynik działania programu dla biblioteki Shark

```
----- Dlib Linear -----
Train data:
MSE: 3.08953
Test data:
MSE: 1.33014
----- Dlib SVM -----
Train data:
AUC ROC: 1
Test data:
AUC ROC: 1
```

Figure 7.3. Wynik działania programu dla biblioteki Dlib

Analizując powyższe wyniki pod krątem regresji liniowej, stwierdzić można iż pierwsze miejsce zajęła biblioteka Shark-ML cechując się najniższymi błędami średniokwadratowymi. Na drugim miejscu znalazła się biblioteka Shogun. Zauważyć można, że mechanizm wyliczania współczynnika determinacji liniowej R^2 wyszedł dla danych testowych poza dopuszczalny zakres. Podejrzewa się, iż jest to wynik kumulacji błędów reprezentacji zmiennoprzecinkowej wartości podczas obliczeń, wskazując że mechanizm wyliczania składowych potrzebnych do obliczenia R^2 dla tej biblioteki jest niestabilny numerycznie. Ostatnie miejsce zajęła biblioteka Dlib, w przypadku której wartość błędu średniokwadratowego przekroczyła 3 jednostki.

Dużo gorsze rezultaty uzyskane zostały przy próbie dokonania klasyfikacji z wykorzystaniem wyżej wspomnianych bibliotek. Każda z nich cechuje się pozornie idealnym dopasowaniem do danych uczących, oraz w przypadku bibliotek Shark-ML i Shogun dopasowaniem porównywalnym do klasyfikacji za pomocą rzutu monetą. Biblioteka Dlib dodatkowo uzyskała pole pod wykresem krzywej charakterysty-

cznej odbiornika na poziomie 1 dla danych testowych. Powyższe wyniki są nierealne do uzyskania, co udowodniono w rozdziale 3.3.1, wykorzystując dedykowane oprogramowanie statystyczne. W związku z tym rozpoczęto analizę przyczyny uzyskanych wyników.

W celu poprawy uzyskanych wyników i rozwiązania napotkanych błędów zdecydowano się na normalizację wartości predyktorów dla obu zestawów danych do przedziału $[0; 1]$ z wykorzystaniem programu JMP, a następnie analizę kodu realizującego uczenie maszyn wektorów nośnych w poszczególnych bibliotekach. Pierwszą z analizowanych bibliotek była biblioteka Shark-ML, w przypadku której wykorzystano ręczne dostrajanie parametrów gamma oraz regularyzacji. Okazało się, że po normalizacji oraz ustawieniu parametrów gamma na 0,16 oraz regularyzacji na 1 uzyskano prawidłowi i prawdopodobne pole pod wykresem na poziomie 0,933129 dla danych testowych. W drugiej kolejności przetestowano bibliotekę Shogun. Niestety pomimo normalizacji danych nie udało się rozwiązać problemu ujemnego współczynnika R^2 dla danych testowych, natomiast poprawie udało się wykonać klasyfikację, uzyskując wartość pola pod wykresem krzywej charakterystycznej odbiornika wynoszącą 0,638112. Należy zwrócić szczególną uwagę, że normalizacja pogorszyła uzyskane wyniki przed bibliotekę Shogun, zmniejszając uzyskane R^2 dla danych uczących i sprawiając, że R^2 dla danych testowych jeszcze bardziej odbiega od swojego prawidłowego zakresu. Ostatnią z bibliotek była biblioteka Dlib, dla której normalizacja zestawu danych dla zadania regresji znacznie poprawiła uzyskany błąd średniokwadratowy, natomiast w dalszym ciągu uzyskane zostało idealne dopasowanie maszyny wektorów nośnych, co może wskazywać na to, iż nie jest on zaimplementowany prawidłowo w bibliotece. Tabela 7.3 podsumowuje wyniki uzyskane po analizie kodu oraz normalizacji danych. Wyjście programów zostało przedstawione na rys. 7.4, 7.5 oraz 7.6

Biblioteka	Metoda uczenia maszynowego	
	Regresja liniowa	Maszyna wektorów nośnych
Shogun	Dane treningowe: MSE = 0,0286481; R^2 = 0,3127 Dane testowe: MSE = 0.0427863; R^2 = -0.389462	Dane treningowe: AUC ROC: 0,789143 Dane testowe: AUC ROC: 0,638112
Shark	Dane treningowe: MSE = 0,0105995; R^2 = 0,745707 Dane testowe: MSE = 0.0122919; R^2 = 0,600826	Dane treningowe: AUC ROC: 0,921643 Dane testowe: AUC ROC: 0,933129
Dlib	Dane treningowe: MSE = 0,481402; Dane testowe: MSE = 0,179538;	Dane treningowe: AUC ROC: 1 Dane testowe: AUC ROC: 1

Table 7.3. Zestawienie uzyskanych wyników poprawności modeli.

Podsumowując, zauważono że każda z bibliotek jest wyjątkowo wrażliwa na zakres wartości predyktorów i w celu prawidłowego wykorzystania przeważnie wymaga, aby każdy z predyktorów zawierał się w przedziale $[0; 1]$. W przypadku zestawu danych klasyfikacyjnych było to szczególnie widoczne, gdyż ze względu na bardzo duże wartości zmiennych po transformacji odwrotnym wzorem Arrheniusa, istniała

duża rozbieżność między ich zakresem, a zakresami pozostałych predyktorów. Ponadto zauważono, że mechanizm obliczania współczynnika determinancji liniowej w bibliotece Shogun oraz model maszyny wektorów nośnych w bibliotece Dlib najprawdopodobniej posiadają błędy implementacji, a więc należy z nich nie korzystać, lub mieć do nich bardzo ograniczony stopień zaufania. Spośród uzyskanych wyników, najlepszym dopasowaniem do danych charakteryzuje się biblioteka Shark, na drugim miejscu znalazła się biblioteka Shogun, natomiast ostatnie miejsce zajęła biblioteka Dlib.

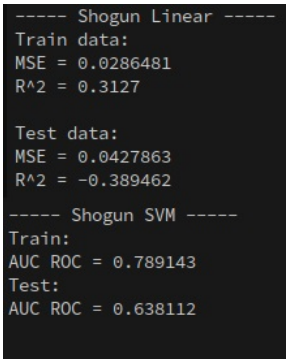

Figure 7.4. Wynik działania programu dla biblioteki Shogun

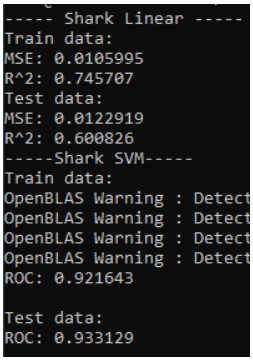

Figure 7.5. Wynik działania programu dla biblioteki Shark

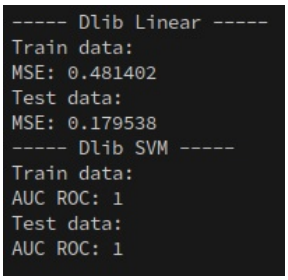

Figure 7.6. Wynik działania programu dla biblioteki Dlib

7.3. Wymagany nakład pracy i jakość źródeł

W procesie pracy z poszczególnymi bibliotekami zauważono, że najmniejszą ilością potrzebnego wkładu pracy charakteryzowała się biblioteka Shark-ML. Wynika to z bardzo przyjaznej dla użytkownika składni, oraz dokładnej dokumentacji dostępnej na stronie internetowej projektu, wraz z przykładami wykorzystania poszczególnych metod. Biblioteka także bez jakichkolwiek problemów została zbudowana i zainstalowana na systemie operacyjnym Ubuntu 22.04 w środowisku WSL2, pozwalając bardzo szybko przejść do badań.

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

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

dostępną na swoim repozytorium. Z racji braku dostępnej online dokumentacji projektu Shogun oraz faktu, że generowane przykłady nie odnoszą się do API biblioteki lecz używają jej w kompletnie odrębny, nienaturalny dla projektu sposób, ustalenie funkcjonalności oraz sposobu realizacji poszczególnych zadań uczenia maszynowego musiało zostać oparte praktycznie wyłącznie o materiały dostępne w formie książkowej. Znacznie utrudnia i wydłuża to proces zastosowania biblioteki do jakiegokolwiek projektu.

Bibliography

[1] Owczarek M. "Charakterystyka systemów ekspertowych i ich zastosowanie w medycynie". In: *Mikroelektronika i informatyka - prace naukowe* 5 (2005), pp. 197 –202. URL: http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.baztech-article-LOD4-0002-0067.

[2] Google. *TensorFlow - API Documentation*. 2013. URL: https://www.tensorflow.org/api_docs.

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

[4] Ayman Mahmoud. *Introduction to Shallow Machine Learning*. 2019. URL: https://www.linkedin.com/pulse/introduction-shallow-machine-learning-ayman-mahmoud/.

[5] Prashant Gupta. *Decision Trees in Machine Learning*. 2017. URL: https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052.

[6] Larry Hardesty. *Explained: Neural Networks - Ballyhooed artificial-intelligence technique known as „deep learning" revives 70-years-old idea*. 2017. URL: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

[7] Raghuram Madabushi. *Neural Network / Machine Learning in resource constrained environments*. 2018. URL: https://medium.com/@raghu.madabushi/neural-network-machine-learning-in-resource-constrained-environments-c934ff1f522.

[8] Data Flair. *Features of C++ | How Programmers use C++ in 10 Unbelievable ways*. 2023. URL: https://data-flair.training/blogs/features-of-c-plus-plus/.

[9] Edytorzy dokumentacji systemu Android. *Neural Networks API*. 2023. URL: https://developer.android.com/ndk/guides/neuralnetworks.

[10] nVidia. *CUDA C++ Programming Guide*. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[11] ONNX Runtime Team. *Get started with ORT for C++*. 2023. URL: https://onnxruntime.ai/docs/get-started/with-c.html.

[12] 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).

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

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

[15] JMP Statistical Discovery. *JMP Pro - Predictive analytics software for scientists and engineers*. 2023. URL: https://www.jmp.com/en_nl/software/predictive-analytics-software.html.

[16] Stephen Lower. *Arrhenius Equation*. 2023. URL: https://chem.libretexts.org/Bookshelves/Physical_and_Theoretical_Chemistry_Textbook_Maps/Supplemental_Modules_(Physical_and_Theoretical_Chemistry)/Kinetics/06%3A_Modeling_Reaction_Kinetics/6.02%3A_Temperature_Dependence_of_Reaction_Rates/6.2.03%3A_The_Arrhenius_Law/6.2.3.01%3A_Arrhenius_Equation.

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

[18] Christian Igel, Verena Heidrich-Meisner, and Tobias Glasmachers. "Shark". In: *Journal of Machine Learning Research* 9 (2008), pp. 993–996.

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

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

[21] 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.

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

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

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

[25] Data Farmers. *Sammon Mapping - A non-linear mapping for data visualisation*. 2019. URL: https://data-farmers.github.io/2019-06-10-sammon-mapping/.