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:
prof. dr hab. eng. Dariusz Uciński

Pracę akceptuję:

....
(data i podpis promotora)

Zielona Góra, June 2023

Summary

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]:

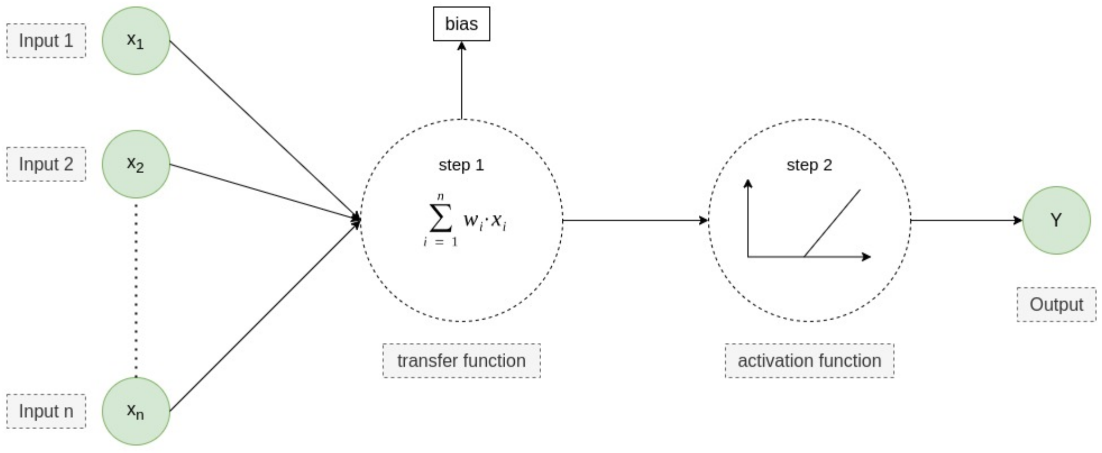


Figure 2.1. Neuron blueprint – Simplelearn.

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

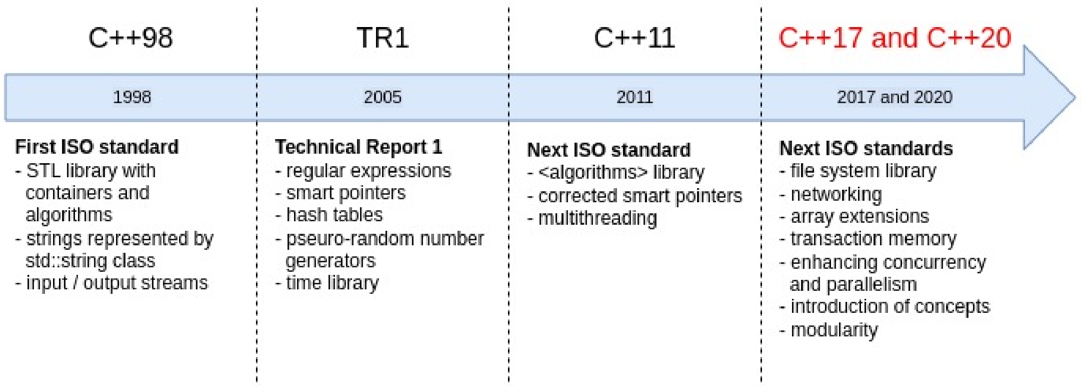


Figure 2.2. Concurrency evolution in modern C++ - Modernes C++.

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. Regression data

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

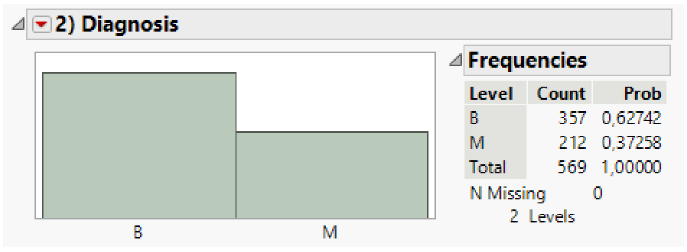
During the process of learning by the model, one of the most crucial steps is to familiarize with the dataset and analyze the distribution of the regressors. JMP Pro [15] was selected as the software for this purpose.

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.



Level	Count	Prob
B	357	0,62742
M	212	0,37258
Total	569	1,00000

N Missing 0
2 Levels

Figure 3.1. Response distribution.

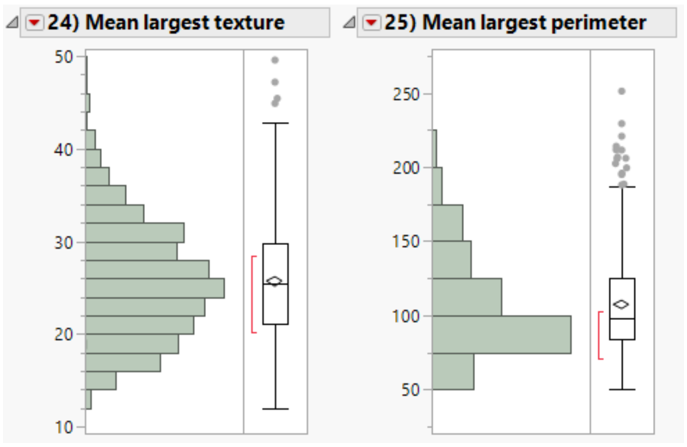


Figure 3.2. Predictors distribution sample.

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*;

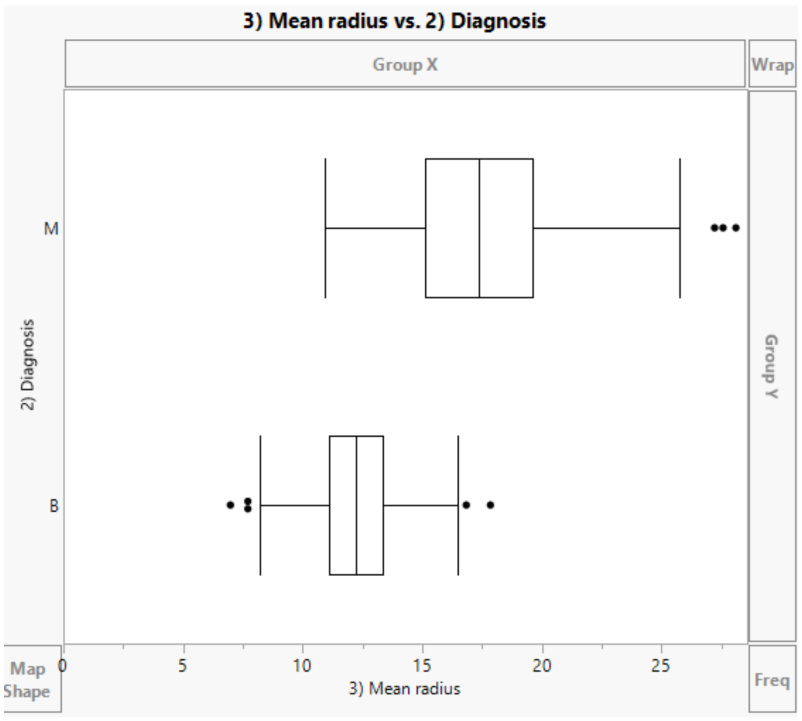


Figure 3.3. Analysis of outliers.

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*;

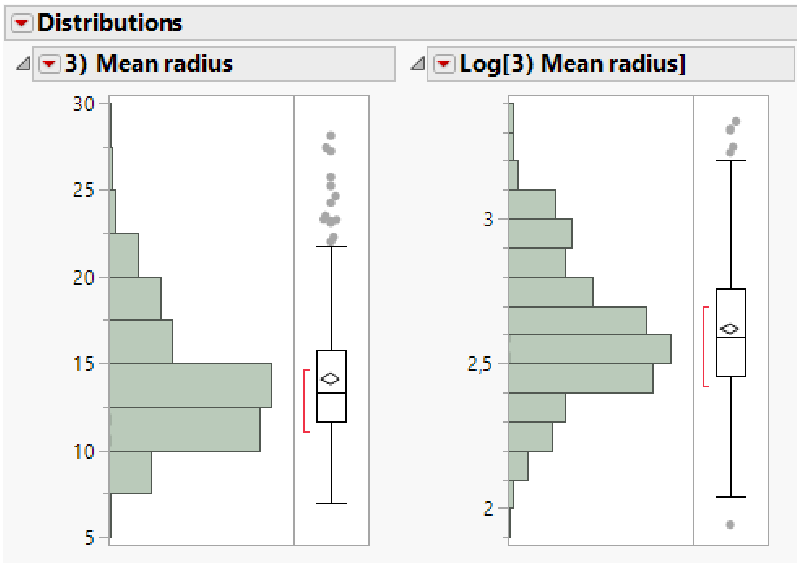


Figure 3.4. Comparison of distributions before and after the logarithmic transformation.

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

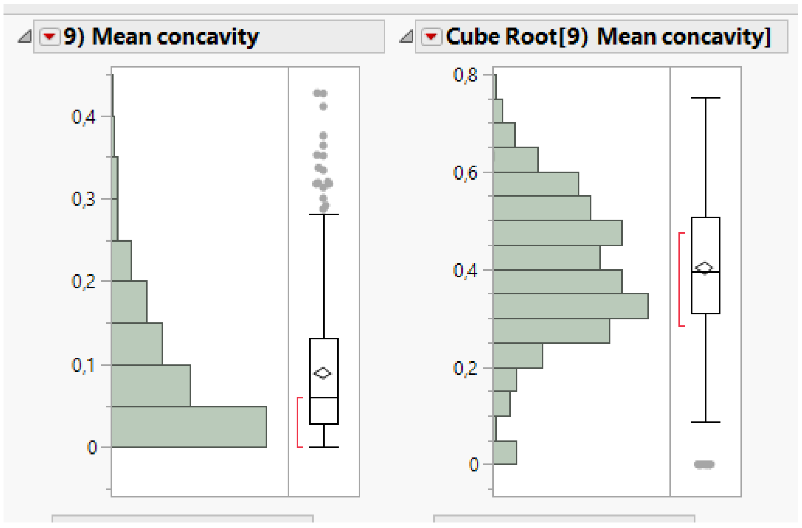


Figure 3.5. Comparison of distributions before and after the cube root transformation.

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*.

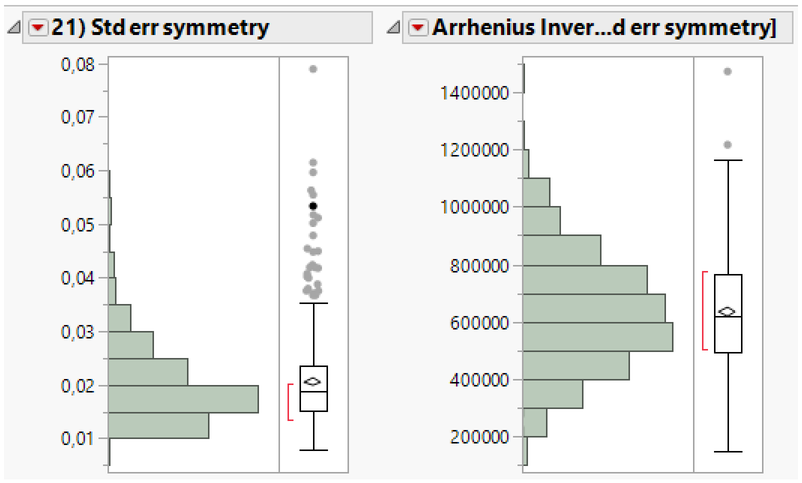


Figure 3.6. Comparison of the distribution before and after inverse Arrhenius transformation.

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.*

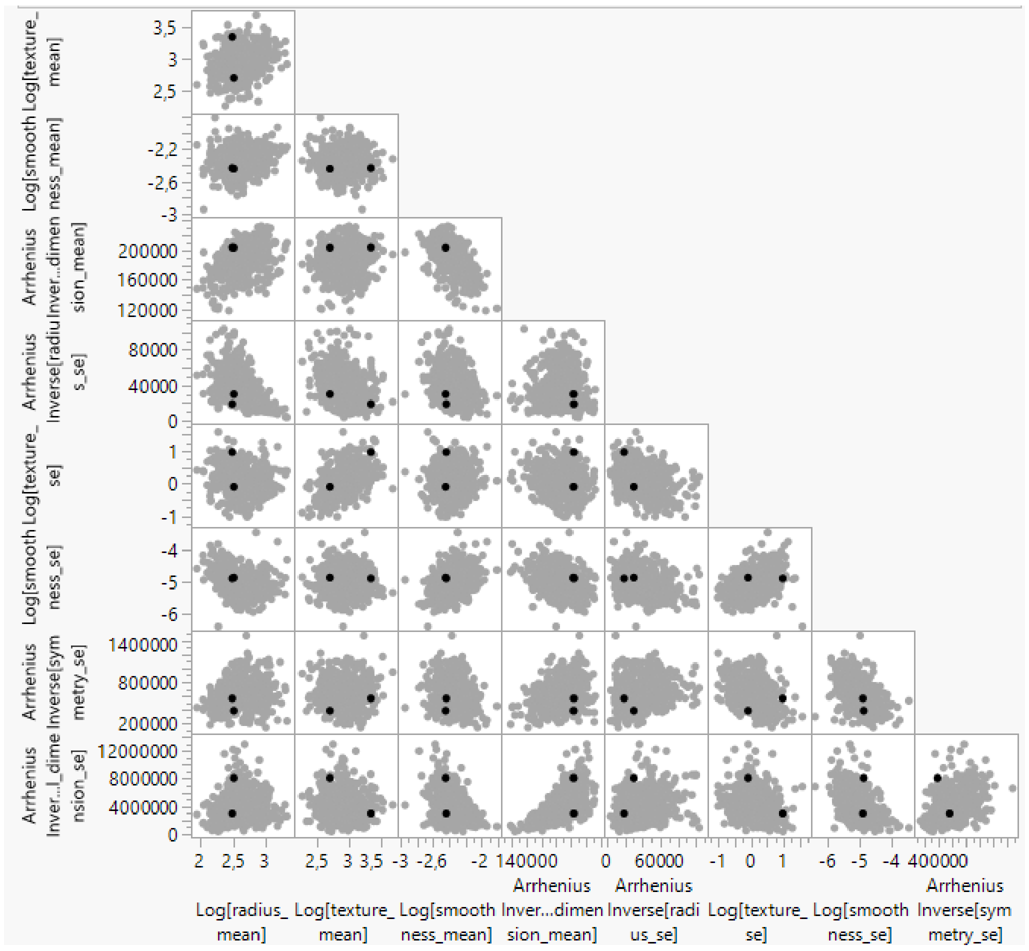


Figure 3.7. Scatterplot matrix after removing correlated predictors

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.

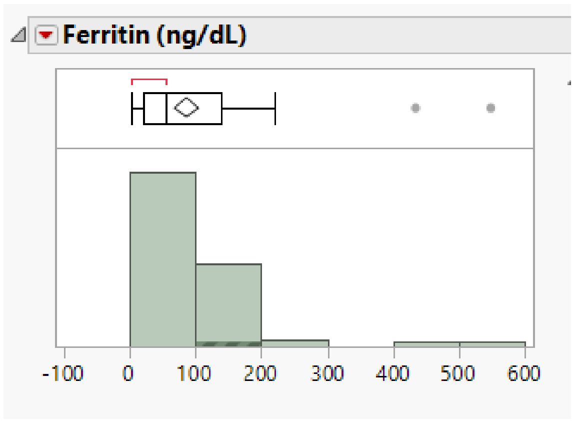


Figure 3.8. Response variable distribution

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.

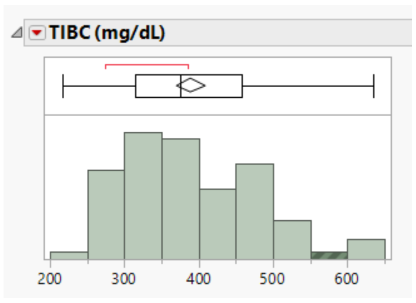


Figure 3.9. Distribution of TIBC.

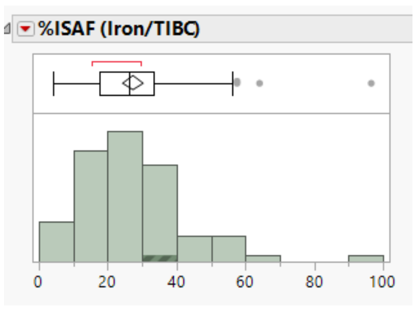


Figure 3.10. Distribution of ISAF.

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/

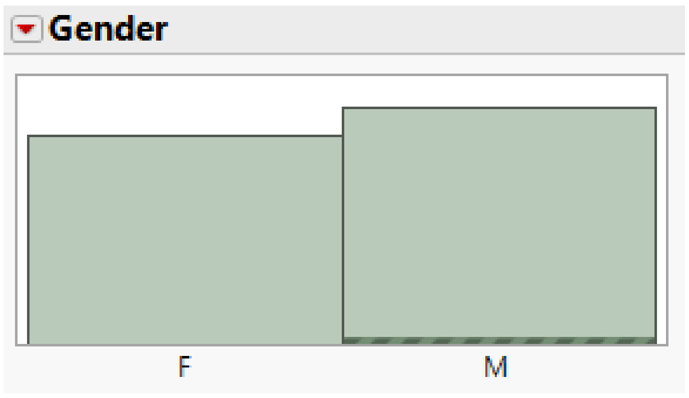


Figure 3.11. Distribution of *Gender*.

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.

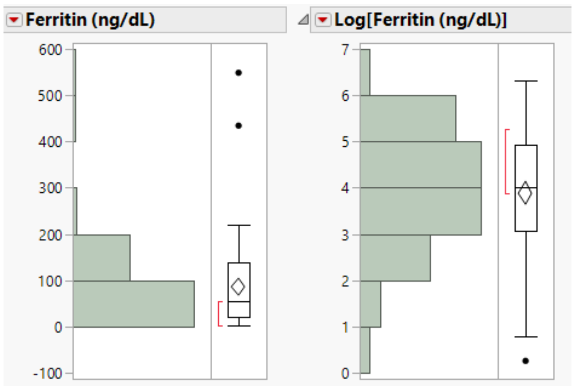


Figure 3.12. Effect of logarithmic transformation on distribution

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

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

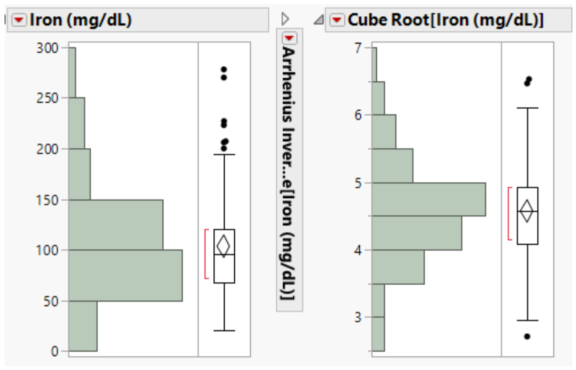


Figure 3.13. Cube root distribution normalization

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 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.

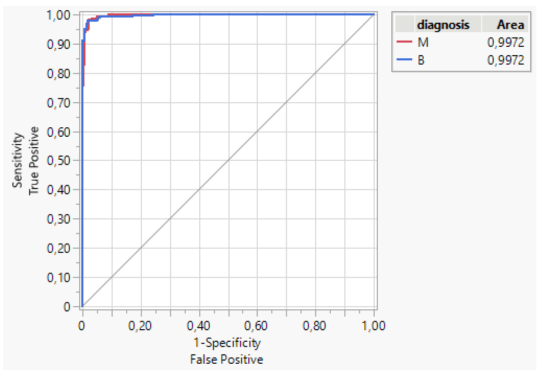


Figure 3.14. ROC curve for the learning data.

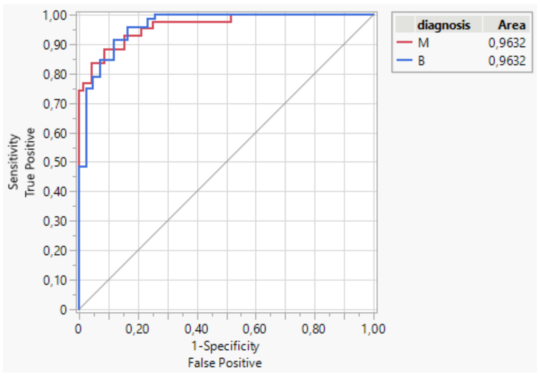
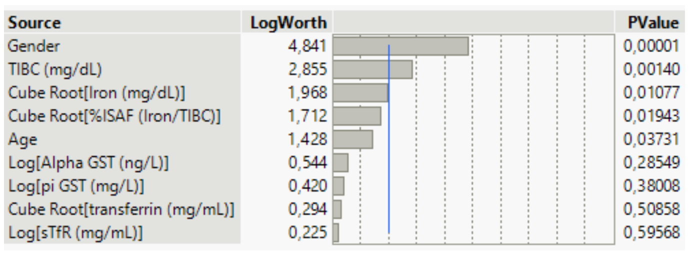


Figure 3.15. ROC curve for the testing data.

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.

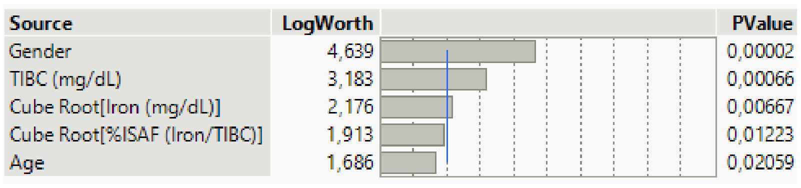
3.3.2. Linear regression

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.



Source	LogWorth	PValue
Gender	4,841	0,00001
TIBC (mg/dL)	2,855	0,00140
Cube Root[Iron (mg/dL)]	1,968	0,01077
Cube Root[%ISAF (Iron/TIBC)]	1,712	0,01943
Age	1,428	0,03731
Log[Alpha GST (ng/L)]	0,544	0,28549
Log[pi GST (mg/L)]	0,420	0,38008
Cube Root[transferrin (mg/mL)]	0,294	0,50858
Log[sTfR (mg/mL)]	0,225	0,59568

Figure 3.16. P-value plot for all predictors.



Source	LogWorth	PValue
Gender	4,639	0,00002
TIBC (mg/dL)	3,183	0,00066
Cube Root[Iron (mg/dL)]	2,176	0,00667
Cube Root[%ISAF (Iron/TIBC)]	1,913	0,01223
Age	1,686	0,02059

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

```cpp
#pragma once

#include <shogun/base/init.h>
#include <shogun/base/some.h>
#include <shogun/io/File.h>
#include <shogun/io/CSVFile.h>
#include <shogun/labels/MulticlassLabels.h>
#include <shogun/lib/SGMatrix.h>
#include <shogun/lib/SGStringList.h>
#include <shogun/lib/SGVector.h>
#include <shogun/preprocessor/RescaleFeatures.h>
#include <iostream>

// helper intermediate packaging for data set
struct Dataset
{
    shogun::SGMatrix<float64_t> trainInputs;
    shogun::SGMatrix<float64_t> testInputs;
    shogun::SGMatrix<float64_t> trainOutputs;
    shogun::SGMatrix<float64_t> testOutputs;
};

// helper struct to distinguish label position
enum class LabelPos
{
    FIRST,
    LAST
};

inline Dataset readShogunCsvData(
                std::string filename, LabelPos labelPos)
{
    using namespace shogun;
    using Matrix = SGMatrix<float64_t>;

    Dataset ret;

    // reading raw csv file content and parsing it into matrix
    auto csvFile = some<CCSVFile>(filename.c_str());
    Matrix data;
    data.load(csvFile);
    // transposing to human-friendly form (column-wise)
    Matrix::transpose_matrix(data.matrix, data.num_rows,
                             data.num_cols);
    // partitioning the matrix into regressors and response variable
    switch(labelPos)
    {
        case LabelPos::FIRST:
            ret.trainInputs = data.submatrix(1, data.num_cols)
                    .clone();
            ret.trainOutputs = data.submatrix(0, 1).clone();
            break;
        case LabelPos::LAST:
            ret.trainInputs = data.submatrix(0, data.num_cols - 1)
                    .clone();
            ret.trainOutputs =
                data.submatrix(data.num_cols - 1, data.num_cols)
                .clone();
            break;
    };

    // transposing back to library-friendly form (row-wise)
    Matrix::transpose_matrix(ret.trainInputs.matrix,
                             ret.trainInputs.num_rows,
                             ret.trainInputs.num_cols);
    Matrix::transpose_matrix(ret.trainOutputs.matrix,
                             ret.trainOutputs.num_rows,
                             ret.trainOutputs.num_cols);
    // splitting data into training and validation
    auto temp = ret.testInputs = ret.trainInputs.submatrix(
        static_cast<long>(0.8 * ret.trainInputs.num_cols),
        ret.trainInputs.num_cols).clone();
    ret.testInputs = std::move(temp);
    auto temp2 = ret.trainInputs.submatrix(
        0, static_cast<long>(0.8 * ret.trainInputs.num_cols))
            .clone();
    ret.trainInputs = std::move(temp2);
    auto temp3 = ret.trainOutputs.submatrix(
        static_cast<long>(0.8 * ret.trainOutputs.num_cols),
        ret.trainOutputs.num_cols).clone();
    ret.testOutputs = std::move(temp3);
    auto temp4 = ret.trainOutputs.submatrix(
        0, static_cast<long>(0.8 * ret.trainOutputs.num_cols))
            .clone();
    ret.trainOutputs = std::move(temp4);
    return ret;
}
```

Listing 4.2. Repackaging data into desired containers

```
#pragma once

#include <inc/shogun/csv.hpp>
#include <inc/shogun/linear.hpp>
#include <inc/shogun/logistic.hpp>
#include <inc/shogun/svm.hpp>
#include <inc/shogun/neural.hpp>

#include <shogun/base/init.h>
#include <iostream>

inline void shogunModels()
{
    using namespace shogun;

    init_shogun_with_defaults();
    // reading data in the custom type
    auto classificationDatasetTemp =
        readShogunCsvData("wdbc_data_with_labels_tn.csv", LabelPos::LAST);
    auto regressionDatasetTemp =
        readShogunCsvData("IronGlutathione_tn.csv", LabelPos::LAST);
    // splitting data into regressors and labels
    auto classificationTrainFeatures =
        some<CDenseFeatures<float64_t>>(
            classificationDatasetTemp.trainInputs);
    auto classificationTestFeatures =
        some<CDenseFeatures<float64_t>>(
            classificationDatasetTemp.testInputs);
    auto classificationTrainLabels =
        some<CMulticlassLabels>(
            classificationDatasetTemp.trainOutputs);
    auto classificationTestLabels =
        some<CMulticlassLabels>(
            classificationDatasetTemp.testOutputs);
    auto regressionTrainFeatures =
        some<CDenseFeatures<float64_t>>(
            regressionDatasetTemp.trainInputs);
    auto regressionTestFeatures =
        some<CDenseFeatures<float64_t>>(
            regressionDatasetTemp.testInputs);
    auto regressionTrainLabels =
        some<CRegressionLabels>(
            regressionDatasetTemp.trainOutputs);
    auto regressionTestLabels =
        some<CRegressionLabels>(
            regressionDatasetTemp.testOutputs);

    // calling the models
    shogunLinear(
        regressionTrainFeatures,
        regressionTestFeatures,
        regressionTrainLabels,
        regressionTestLabels);
    shogunSVM(
        classificationTrainFeatures,
        classificationTestFeatures,
        classificationTrainLabels,
        classificationTestLabels);

    exit_shogun();
}
```

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.

```
#pragma once

#include <shogun/preprocessor/RescaleFeatures.h>

inline void normalize(auto& inputs)
{
    using namespace shogun;

    // creating normalizer
    auto scaler = wrap(new CRescaleFeatures);
    // training and transforming
    scaler->fit(inputs);
    scaler->transform(inputs);
}
```

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].

```cpp
#pragma once

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

    // creating the kernel
    auto gaussKernel = some<CGaussianKernel>(inputs, inputs, 0.5);
    // creating the reductor
    auto pca = some<CKernelPCA>();
    // configuring the reductor
    pca->set_kernel(gaussKernel.get());
    pca->set_target_dim(target_dim);
    // training
    pca->fit(inputs);
    // dimensionality reduction
    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;

    // creating the model
    float64_t tauRegularization = 0.0001;
    auto linear =
        some<CLinearRidgeRegression>(tauRegularization, nullptr, nullptr);
    // training
    linear->set_labels(trainOutputs);
    linear->train(trainInputs);
    // validation
    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>

```

```cpp

inline void shogunLogistic(
    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;

    // creating the model
    auto logReg = some<CMulticlassLogisticRegression>();

    // training
    logReg->set_labels(trainOutputs);
    logReg->train(trainInputs);

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

    std::cout << "Test:" << std::endl;
    auto prediction2 = wrap(logReg->apply_multiclass(testInputs));
    shogunVerifyModel(prediction2, testOutputs);
}
```

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

```cpp
#pragma once

#include <inc/shogun/verify.hpp>

#include <iostream>
#include <shogun/base/some.h>
#include <shogun/features/DenseFeatures.h>
#include <shogun/evaluation/CrossValidation.h>
#include <shogun/labels/MulticlassLabels.h>
#include <shogun/kernel/GaussianKernel.h>
#include <shogun/multiclass/MulticlassLibSVM.h>
#include <shogun/evaluation/MulticlassAccuracy.h>
#include <shogun/evaluation/StratifiedCrossValidationSplitting.h>
#include <shogun/modelselection/ModelSelection.h>
#include <shogun/modelselection/ModelSelectionParameters.h>
#include <shogun/modelselection/GridSearchModelSelection.h>
#include <shogun/modelselection/ParameterCombination.h>

```

```
inline void shogunSVM(
                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;

    // creating kernel
    auto kernel = some<CGaussianKernel>();
    kernel->init(trainInputs, trainInputs);
    // creating and configuring the model
    auto svm = some<CMulticlassLibSVM>(LIBSVM_C_SVC);
    svm->set_kernel(kernel);

    // searching for hyperparameters
    auto root = some<CModelSelectionParameters>();
    // missclasification avoidance degree
    CModelSelectionParameters* c = new CModelSelectionParameters("C");
    root->append_child(c);
    c->build_values(1.0, 1000.0, R_LINEAR, 100.);
    // creating the kernel pointer
    auto paramsKernel = some<CModelSelectionParameters>("kernel", kernel);
    root->append_child(paramsKernel);
    // creating the weights pointer
    auto paramsKernelWidth =
            some<CModelSelectionParameters>("combined_kernel_weight");
    paramsKernelWidth->build_values(0.1, 10.0, R_LINEAR, 0.5);
    paramsKernel->append_child(paramsKernelWidth);
    // creating the partition for cross validation
    index_t k = 3;
    CStratifiedCrossValidationSplitting* splitting =
            new CStratifiedCrossValidationSplitting(trainOutputs, k);
    // creating the accuracy criterion
    auto evalCriterium = some<CMulticlassAccuracy>();
    // creating the cross-validation object
    auto cross =
            some<CCrossValidation>(
                svm, trainInputs, trainOutputs, splitting, evalCriterium);
    cross->set_num_runs(1);
    // creating model selection object
    auto modelSelection = some<CGridSearchModelSelection>(cross, root);
    // selecting and applying the parameters
    CParameterCombination* bestParams =
            wrap(modelSelection->select_model(false));
    bestParams->apply_to_machine(svm);
    bestParams->print_tree();

    // training
    svm->set_labels(trainOutputs);
    svm->train(trainInputs);

    // model evaluation
    std::cout << "-----␣Shogun␣SVM␣-----" << std::endl;
    std::cout << "Train:" << std::endl;
    auto prediction = wrap(svm->apply_multiclass(trainInputs));
    shogunVerifyModel(prediction, trainOutputs);

```

```cpp
    std::cout << "Test:" << std::endl;
    auto prediction2 = wrap(svm->apply_multiclass(testInputs));
    shogunVerifyModel(prediction2, testOutputs);
}
```

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

```cpp
#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/KNN.h>
#include <shogun/distance/EuclideanDistance.h>

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

    // preparing the distance type
    auto distance = some<CEuclideanDistance>(trainInputs, trainInputs);
    // preparing the model
    std::int32_t k = 3;
    auto knn = some<CKNN>(k, distance, trainOutputs);
    // evaluation
    std::cout << "-----␣Shogun␣KNN␣-----" << std::endl;
    std::cout << "Train:" << std::endl;
    auto prediction = wrap(knn->apply_multiclass(trainInputs));
    shogunVerifyModel(prediction, trainOutputs);

    std::cout << "Test:" << std::endl;
    auto prediction2 = wrap(knn->apply_multiclass(testInputs));
    shogunVerifyModel(prediction2, testOutputs);
}
```

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;

    // marking regressors as continuous
    SGVector<bool> featureType(1);
    featureType.set_const(false);
    // creating a binary decision tree
    auto tree = some<CCARTree>(featureType, PT_REGRESSION);
    tree->set_max_depth(3);
    // creating the loss function
    auto loss = some<CSquaredLoss>();
    // creating and configuring the model
    constexpr int iterations = 100;
    constexpr int learningRate = 0.1;
    constexpr int subsetFraction = 1.0;
    auto model = some<CStochasticGBMachine>(tree,
                                            loss,
                                            iterations,
                                            learningRate,
                                            subsetFraction);
    // training
    model->set_labels(trainOutputs);
    model->train(trainInputs);
    // evaluation
    std::cout << "-----␣Shogun␣Gradient␣Boost␣-----" << std::endl;
    std::cout << "Train:" << std::endl;
    auto prediction = wrap(model->apply_multiclass(trainInputs));
    shogunVerifyModel(prediction, trainOutputs);

    std::cout << "Test:" << std::endl;
    auto prediction2 = wrap(model->apply_multiclass(testInputs));
    shogunVerifyModel(prediction2, testOutputs);
}
```

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

```
#pragma once

#include "inc/shogun/verify.hpp"

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

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

    // creating and configuring the model
    constexpr std::int32_t numRandFeats = 1;
    constexpr std::int32_t numBags = 10;
    auto randForest =
        some<CRandomForest>(numRandFeats, numBags);
    auto vote = some<CMajorityVote>();
    randForest->set_combination_rule(vote);
    // marking data as continuous
    SGVector<bool> featureType(1);
    featureType.set_const(false);
    randForest->set_feature_type(featureType);
    // training
    randForest->set_labels(trainOutputs);
    randForest->set_machine_problem_type(PT_REGRESSION);
    randForest->train(trainInputs);
    // evaluation
    std::cout << "-----␣Shogun␣Random␣Forest␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = wrap(randForest->apply_regression(trainInputs));
    shogunVerifyModel(predictions, trainOutputs);

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

4.4.6. Neural network

First step in creating a neural network with this library is to configure the network topology, using *CNeuralLayers* object. It consists of a range of methods, which create layers with given activation function:

- *input()* - input layer with a certain number of inputs;
- *logistic()* - fully connected sigmoid function layer;
- *linear()* - fully connected linear function layer;
- *rectified_linear()* - fully connected ReLU layer;
- *leaky_rectified_linear* - fully connected Leaky ReLU layer;
- *softmax* - fully connected softmax layer;

The order in which we call the methods is important, because it decides about the order of layers within the model. After completing the configuration process, it is possible to confirm the architecture by creating another object via the *done()* method, and using it to initialize *CNeuralNetwork* class. In order to connect the layers, the *quick_connect* method needs to be called, and subsequently, the weights have to be initialized via *initialize_neural_network()* function. It accepts a parameter describing a Gaussian distribution used to initialize weights.

The next step is to configure the optimizer using *set_optimization()* method. The class *CNeuralNetwork* supports optimizing using the steepest descent method and Broyden-Fletcher-Goldfarb-Shannon method. This model has in-built L2 regularization, which can be further configured, similarly to other parameters such as learning coefficient, epoch count, convergence criterion for the loss function, or the size of batches. Unfortunately, user cannot select the loss function in question, since it is established automatically based on the label type. Listing 4.11 shows the full process of building, configuring and teaching a network. Unfortunately, due to the lack of hyperbolic tangential function implementation, a reLU function was used in its stead, which affects the achieved results.

Listing 4.11. Example of neural network use in Shogun.

```cpp
#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;

    // constructing the network architecture
    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();
    // creating the network
    auto network = some<CNeuralNetwork>(allLayers);
    network->quick_connect();
    network->initialize_neural_network();
    // network configuration
    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);
    // training
    network->set_labels(trainOutputs);
    network->train(trainInputs);
    // validation
    std::cout << "-----␣Shogun␣Neural␣Network␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = wrap(network->apply_multiclass(trainInputs));
    shogunVerifyModel(predictions, trainOutputs);

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

4.5. Model analysis methods

4.5.1. Mean squared error

Calculation of mean squared error in the Shogun library comes down to creating an object of type *CMeanSquaredError* as an argument to the *some<>()* template function. It is returned as a pointer to the object. In order to acquire the value of the error for given data, user needs to call the *evaluate()* method, passing prediction results and observed output. Listing 4.12 presents the detailed way to use it.

Listing 4.12. Example of calculating mean squared error in Shogun[3]

```
using namespace shogun;

// [...]

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

4.5.2. Mean absolute error

Mean absolute error calculation is carried out by the *CMeanAbsoluteError* class, which serves as an evaluator. It is created by a call to *some<>()* function and used for evaluation by calling the *evaluate()* method, providing the results of a model and expected labels of regression or classification. Listing 4.13 shows how to use the aforementioned class.

Listing 4.13. Example of mean absolute error calculation using Shogun[3].

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

4.5.3. Logarithmic loss function

Logarithmic loss function is possible to calculate using the Shogun library using *CLogLoss* class, however the public interface of said class indicate that it should be rather used by a model than directly by the user. It provides *get_square_grad()* method which allows for calculation of the squared gradient between prediction and expected result. The use case is presented on listing 4.14.

Listing 4.14. Example of use for *CLogLoss* class.

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

4.5.4. R^2 metric

The Shogun library does not provide implementation for the R^2 metric as such, so despite the ability to use the in-built method of mean squared error calculation, the variance of output needed for calculating the metric needs to be acquired from a user-provided mechanism. Listing 4.15 shows a function that verifies the correctness of models described in previous paragraphs by calculating the R^2 metric.

Listing 4.15. Example of R^2 metric calculation.

```cpp
#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;

    // mean squared error
    auto mseError = some<CMeanSquaredError>();
    auto mse = mseError->evaluate(predictions, targets);
    std::cout << "MSE␣=␣" << mse << std::endl;
    // R^2 metric
    float64_t avg = 0.0;
    float64_t sum = 0.0;
    // mean calculation
    for (index_t i = 0; i < targets->get_num_labels(); i++)
    {
        avg += targets->get_label(i);
    }
    avg /= targets->get_num_labels();
    // variance calculation
    for (index_t i = 0; i < targets->get_num_labels(); i++)
    {
        sum += std::pow(targets->get_label(i), 2);
    }
    float64_t variance = (sum / targets->get_num_labels()) - std::pow(avg, 2);
    // R^2 metric calculation
    auto r_square = 1 - (mse / variance);
    std::cout << "R^2␣=␣" << r_square << std::endl << std::endl;
}

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

    // auc roc calculation
    auto roc = some<CROCEvaluation>();
    roc->evaluate(predictions->get_binary_for_class(1),
                  targets->get_binary_for_class(1));
    std::cout << "AUC␣ROC␣=␣" << roc->get_auROC() << std::endl;
}
```

4.5.5. Accuracy

In order to calculate the accuracy metric for regression tasks, this library provides *CMulticlassAccuracy* class. It also allows for getting a misclassification matrix. The *CAccuracyMeasure* class mentioned in [3] however was not found, which indicates it was removed from the library. Listing 4.16 shows how to use *CMulticlassAccuracy*.

Listing 4.16. Example of accuracy calculation in Shogun.

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

4.5.6. Precision, Recall and F1 metric

The [3] handbook mentions *CRecallMeasure* and *CF1Measure* classes that are supposed to allow the user to calculate the recall and F1 metric of the model, however while working with the library, their definitions were not found, neither any other classes of similar functionality, and as such it was assumed that the Shogun library does not provide the implementation for them. The motivation to touch on this topic was to inform the reader of possible discrepancies between available sources and current state of the library.

4.5.7. ROC curve

The Shogun library provides implementation to calculate ROC curve as the *CROCEvaluation* class. Listing 4.17 shows the way how to use it.

Listing 4.17. ROC curve calculation example for 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-fold cross-validation

Cross-validation inside the Shogun library is a complex mechanism, which in order to use it requires a decision tree of available parameters, represented by the *CModelSelectionParameters* class. The user can select the model and evaluation criterion for it by creating objects of selected classes and providing them to the *CCrossValidation* instance constructor. The next step is to create an instance of *CGridSearchModelSelection* which will select parameters. The last phase is to configure the desired model and complete the teaching process. Precise example of the whole mechanism was presented on listing 4.18.

Listing 4.18. Preparation of multiclass classification via linear regression with cross-validation in Shogun.

```
#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;

    // creating the parameter tree
    auto root = some<CModelSelectionParameters>();
    // regularization coefficient
    CModelSelectionParameters* z = new CModelSelectionParameters("m_z");
    root->append_child(z);
    z->build_values(0.2, 1.0, R_LINEAR, 0.1);
    // creating the decision tree partition strategy
    index_t k = 3;
    CStatifiedCrossValidationSplitting* splitting =
        new CStatifiedCrossValidationSplitting(labels, k);
    // creating the evaluation criterion
    auto evalCriterium = some<CMulticlassAccuracy>();
    // creating the logistic regression model
    auto logReg = some<CMulticlassLogisticRegression>();
    // creating the cross-validation object
    auto cross = some<CCrossValidation>(logReg, trainInputs, trainOutputs,
                                        splitting, evalCriterium);
    cross->set_num_runs(1);
    auto modelSelection = some<CGridSearchModelSelection>(cross, root);
    // selecting model parameters
    CParameterCombination* bestParams =
        wrap(modelSelection->select_model(false));
    // applying parameters to the model
    bestParams->apply_to_machine(logReg);
    // printing the decision tree
    bestParams->print_tree();

    // training
    logReg->set_labels(trainOutputs);
    logReg->train(trainInputs);

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

    std::cout << "Test:" << std::endl;
    auto prediction2 = wrap(logReg->apply_multiclass(testInputs));
    shogunVerifyModel(prediction2, testOutputs);

    delete splitting;
}
```

4.6. Documentation and sources availability

Internet sources such as community forums focus on using Shogun with different programming languages such as eg. Python, however using its source code available on GitHub [17] it is possible to generate examples of its use in C++ language aswell in the *examples* directory. Those examples need to be build using a specific Python script contained within the repository, that generates listings of codes in desired language in JSON files. Unfortunately, they present the use of the library in very unnatural, procedurally generated manner, causing them to be useless when attempting to use the actual project's API. In addition, the only form of project documentation is restricted to the comments within the source code, making the user wander through the repository in search of the needed information.

The Shogun is one of the libraries described within the *"Hands-On Machine Learning with C++"*[3] handbook, which both introduces the reader to basic functionalities of the library, aswell as summarizes the basics of machine learning theory in order to use it. Majority of various model types examples within this book is equipped with listings of main code snippets for the Shogun library. It is important to mention however, that they are different from the examples generated by the build script, showing the use of actual API of the library. During the making of this chapter, this handbook proved to be the only valuable source on the topic.

Chapter 5

Shark-ML

5.1. Introduction

Shark-ML is a machine learning library dedicated for the C++ language. It has an open source and is distributed under the *GNU Lesser General Public License.* The main focus of this library are linear and nonlinear optimization problems (as such containing parts of linear algebra library functionalities), kernel machines (eg. support vector machine) and neural networks [18]. The entity that provides the library is the University of Copenhagen in Denmark, and Neuroinformatics institute of Ruhr-Universitat Bochum in Germany.

5.2. Data sources formats

The library operates on its own representations of matrices and vectors, which are created by wrapping raw arrays using specialized adapters, such as eg. *remora::dense-_matrix_adaptor<>()*, or by using containers from the C++ standard library and *createDataFromRange()* function. This mechanism is the exact same as in case of other libraries mentioned in this dissertation, which gives the user a lot of freedom in terms of what format the data is stored in, and how it's read. The library also contains a dedicated parser for files in the CSV format, however it allows the presence of only numeric data within the file. In order to use it, it is required to create a *ClassificationDataset* or *RegressionDataset* container and the *importCSV()* method, which stores the read data in the aforementioned container object via the return-by-parameter mechanism. One of the arguments of importCSV() method describes which of the columns contain labels, thanks to which the library is able to instantly separate the predictors from labels. The article [19] available on GitHub shows also how to download data from the internet using *curl* library API, and how to process it into the form acceptable by Shark-ML. Current version of the library also contains some in-built functions that support downloading data via the HTTP protocol. Listing 5.1 shows how to read data from .csv file that is present on the user's hard drive.

Listing 5.1. Reading from CSV file in Shark-ML.

```cpp
#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;
}
```

In order to wrap the data contained within C++ standard library containers into objects accepted by the Shark-ML, it is necessary to use the aforementioned adaptor functions which accept the pointer for the data in raw array contained within the container together with expected size of target matrix or vector. The wrapping method was shown on listing 5.2.

Listing 5.2. Data wrapping method in Shark-ML[3].

```cpp
// sample data contained within the C++ STL vector class
std::vector<float> data{1, 2, 3, 4};

// wrapping data into a 2x2 matrix
auto m = remora::dense_matrix_adaptor<float>(data.data(), 2, 2);

// wrapping data into a 1x4 vector
auto v = remora::dense_vector_adaptor<float>(data.data(), 4);
```

5.3. Data exploration and processing methods

5.3.1. Normalization

Shark-ML implements normalization as training class for model *Normalizer*, providing three usable classes to the user:

- *NormalizeComponentsUnitInterval* - processes data as if they are contained within the unit interval;
- *NormalizeComponentsUnitVariance* - recalculates data to achieve unit variance, and sometimes also the mean equal to 0.
- *NormalizeComponentsWhitening* - data is processed in a way that guarantees mean equal to 0 and user-provided variance (unit variance by default).

Those classes base on using the *train()* method on the normalizer object, in order to configure it to process both testing data, and any other data the user wants to introduce into the model. Their additional functionality is data shuffling and separating a validation dataset using *shuffle()* and *splitAtElement()* methods of the *ClassificationDataset* object. Listing 5.3 shows an example of initial data processing using normalization.

Listing 5.3. Initial training data preprocessing in Shark-ML[19].

```cpp
#pragma once

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

    // shuffling the data
    trainData.shuffle();
    // creating the normalizer
    using Trainer = NormalizeComponentsUnitVariance<RealVector>;
    bool removeMean = true;
    Normalizer<RealVector> normalizer;
    Trainer normalizingTrainer(removeMean);
    // teaching the normalizer the mean and variance of the training data
    normalizingTrainer.train(normalizer, trainData.inputs());
    // transforming the training data
    return transformInputs(trainData, normalizer);
}
```

5.3.2. Dimensionality reduction

5.3.2.1. Principal component analysis

Teh dimensionality reduction algorithm utilizing principal component analysis is implemented in Shark as *PCA* class. It uses a linear model as an encoder and accepts target data dimension in the *encoder()* method. The result of this function call is configuring the linear model in a way that allows creating datasets with reduced dimensions. Listing 5.4 shows use case of the *PCA* class.

Listing 5.4. Dimensionality reduction using *PCA* class and an encoder in Shark-ML.

```cpp
// [...]

// creating PCA kernel
shark::PCA pca(data);
shark::LinearModel<> enc;

// configuring the encoder for dimensionality reduction
constexpr int nbOfDim = 2;
pca.encoder(enc, nbOfDim);
auto encoded_data = enc(data);
```

5.3.2.2. Linear discriminant analysis

In case of Shark-ML library, the linear discriminant analysis (*LDA*) is based on finding an analytical solution via configuration of *LinearClassifier* model via *LDA* training class, using the *train()* method. It is also possible to use the same model for classification task, receiving predictions for a data set by calling the linear classifier functor (use of operator()) providing data from *ClassificationDataset* acquired by calling *inputs()* method. Detailed implementation on the dimensionality reduction of data using this method are presented on listing 5.5.

Listing 5.5. Dimensionality reduction example using LDA model in Shark-ML[3].

```cpp
using namespace shark;

void LDAReduction(const UnlabeledData<RealVector>& data,
                  const UnlabeledData<RealVector>& labels,
                  std::size_t target_dim)
{
    // creating LDA object and encoder
    LinearClassifier<> encoder;
    LDA lda;
    // creating dataset
    LabeledData<RealVector, unsigned int> dataset(
        labels.numberOfElements(), InputLabelPair<RealVector, unsigned int>(
            RealVector(data.element(0).size()), 0));
    // populating the dataset
    for (std::size_t i = 0; i < labels.numberOfElements(); ++i)
    {
        // changing the classes indexes to start from 0
        dataset.element(i).label =
            static_cast<unsigned int>(labels.element(i)[0]) - 1;
        dataset.element[i].input = data.element(i)
    }
    // encoder training
    lda.train(encoder, dataset);
    // creating reduced dataset
    auto new_labels = encoder(data);
    auto new_data = encoder.decisionFunction()(data);
}
```

5.3.3. L1 regularization

The Shark library, in contrary to the Shogun library does not contain specifically defined mechanisms for standalone regularization in the models. Instead, there is a possibility to put a regularization object in the trainer object, using the *setREgularization()* function. In order to use the Lasso method, the user needs to put the *shark::OneNormRegularizer* object within the trainer of choice, and next carry out the teaching process.

5.3.4. L2 regularization

Similarly to the Lasso method, using L2 regularization in the trained model is based on providing the right regularizer object to the trainer object. For L2 method it is the *shark::TwoNormRegularizer* class.

5.4. Machine learning models

5.4.1. Linear regression

One of the fundamental models offered by this library is the linear regression. In order to use it, the *LinearModel* class is provided, which offers an analytical solution using the *LinearRegression* trainer class, or iterative approach implemented by the *LineatSAGTrainer* class, which utilizes the statistic average gradient method. In case of more complicated regression models, where the analytical solution might not exist, the iterative method can be used with the optimizer chose by the user. The whole method comes down into teaching the optimizer using a loss function, and subsequently loading the acquired weights into the regression model. Parameters of the model can be read by calling the *offset()*, *matrix()* or *parameterVector()* methods. Listing 5.6 shows the iterative approach, and listing 5.7 presents the analytical method.

Listing 5.6. Example of linear regression using SAG - Shark-ML.

```
#pragma once

#include <iostream>
#include <inc/shark/printEvaluation.hpp>
#include <shark/Algorithms/Trainers/LinearRegression.h>
#include <shark/Models/LinearModel.h>
#include <shark/ObjectiveFunctions/Loss/SquaredLoss.h>

inline void sharkLinear(const shark::RegressionDataset& trainData,
                        const shark::RegressionDataset& testData)
{
    using namespace shark;

    // preparing the model and the trainer
    LinearModel<> model;
    LinearRegression trainer;
    // training
    trainer.train(model, trainData);
    // validation
    std::cout << "-----␣Shark␣Linear␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = model(trainData.inputs());
    printSharkModelEvaluation(
        trainData.labels(), predictions);

    std::cout << "Test␣data:" << std::endl;
    predictions = model(testData.inputs());
    printSharkModelEvaluation(
        testData.labels(), predictions);
}
```

Listing 5.7. Example of linear regression with analytical method - Shark-ML[3].

```
#pragma once

inline void sharkLinear(const shark::RegressionDataset& trainData,
                        const shark::RegressionDataset& testData)
{
    using namespace shark;

    // creating model
    LinearRegression trainer;
    LinearModel<> model;
    // training the model
    trainer.train(model, trainData);
    // reading model parameters
    std::cout << "intercept:␣" << model.offset() << std::endl;
    std::cout << "matrix:␣" << model.matrix() << std::endl;
    // ewaluacja
    std::cout << "-----␣Shark␣Linear␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = model(trainData.inputs());
    printSharkModelEvaluation(
        trainData.outputs(), predictions, Task::REGRESSION);

    std::cout << "Test␣data:" << std::endl;
    predictions = model(testData.inputs());
    printSharkModelEvaluation(
        trainData.outputs(), predictions, Task::REGRESSION);
}
```

5.4.2. Logistic regression

The logistic regression mechanism available in the Shark-ML library by its nature solves a binary classification problem. There is however possibility to use multiple classifiers, which count is described by the equation:

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

where N designates the number of classes present in the problem. Created classifiers are then combined into one object using the right configuration of *OneVersusOneClassifier* class, which then allows for solving a multiclass classification. In order to achieve this, the dataset needs to be iteratively split into binary characteristic subproblems using the in-built *binarySubProblem()* method, which accepts the data set and desired classes. Teaching process is carried out using the *LogisticRegression* trainer object. After the submodel batch is trained, they are loaded into the primary model. The final use of the ready multiclass classifier is no different than using a model acquired eg. in linear classification. Listing 5.8 describes the function for building a multiclass classifier, while listing 5.9 presents the creation of simple binary classification model.

Listing 5.8. Example of multiclass classifier building - Shark[3].

```cpp
using namespace shark;

// [...]

void LRClassification(const ClassificationDataset& train,
                      const ClassificationDataset& test,
                      unsigned int num_classes)
{
    // creating the classifier object and an array of sub-classifiers
    OneVersusOneClassifier<RealVector> ovo;
    auto pairs = num_classes * (num_classes - 1) / 2;
    std::vector<LinearClassifier<RealVector>> lr(pairs);

    // iterative configuration of sub-classifiers
    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)
        {
            // getting a binary subproblem
            ClassificationDataset binary_cls_data =
                binarySubProblem(train, cls2, cls1);

            // training the sub-model
            LogisticRegression<RealVector> trainer;
            trainer.train(lr[n], binary_cls_data);

            // loading sub-model into the series
            ovo_classifiers.push_back(&lr[n]);
        }
        // loading the series into the main classifier
        ovo.addClass(ovo_classifiers);
    }
    // using the model
    auto predictions = ovo(test.inputs());
    // [...]
}
```

Listing 5.9. Example of simple binary linear regression - Shark-ML.

```cpp
#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>
#include <iostream>

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

    // creating model
    std::cout << "Start␣logistic\n";
    LinearClassifier<RealVector> logisticModel;
    LogisticRegression<RealVector> trainer;
    // trainig
    std::cout << "Training\n";
    trainer.train(logisticModel, trainData);
    // evaluation
    std::cout << "-----Shark␣Logistic␣Regression-----" << std::endl;
    std::cout << "Train␣data␣model␣evaluation:" << std::endl;
    auto predictions = logisticModel(trainData.inputs());
    printSharkModelEvaluation(
        trainData.labels(), predictions);

    std::cout << "Test␣data␣model␣evaluation:" << std::endl;
    predictions = logisticModel(testData.inputs());
    printSharkModelEvaluation(
        testData.labels(), predictions);
}
```

5.4.3. Support vector machine

One of the very important from the perspective of Shark-ML library use machine learning methods provided by it is the support vector machines, which is a type of kernel model. It is based on performing linear regression inside the space of characteristics derived from the used kernel. Similarly to the logistic regression case, the API of the library allows directly only for binary classification, however using it to solve a multiclass problem requires a combination of many dichotomous support vector machines into a compound model, which can be achieved using *OneVersusOneClassifier* and requires the submodels count described by the 5.1 equation. Consistently with the characteristic trait of this library, the use of the method is split into creating an instance of the model and a trainer object, which configures the model in the training process. The user is presented with following classes:

- *GaussianRbfKernel* - it is responsible for calculating the similarity between provided characteristics using a radial basis function;
- *KernelClassifier* - it carries out the linear regression within the space provided by the kernel;
- *CSvmTrainer* - trainer class which performs the training in accordance with configured parameters;

The user can configure the model with following parameters:

- model throughput - provided to the *GaussianRbfKernel* constructor as a number from the interval of $[0; 1]$;
- regularization - a real number set in the constructor of *CSvmTrainer*. By default the support vector machines uses L1 norm as a punishment for crossing the desired boundary;

- bias - a binary flag (bool) describing whether the model is supposed to the bias term, set in the *CSvmTrainer* constructor;
- *sparsify* - a parameter stating whether the model should keep the non-support vectors, available via the *sparsify()* trainer method;
- minimal precision for finishing the training - it allows the user to specify the satisfying precision of the model. It is available as a member of the *stopping-Condition()* structure for the trainer;
- cache size - set via the *setCacheSize()* method of the trainer;

The use method of the model is identical to the other models, via calling the operator() member function. Listing 5.10 shows an example of creating and configuring the model based on the available documentation of the library, however listing 5.11 shows the creating of support vector machine for multiclass problems inside a function that accepts training and validation datasets.

Listing 5.10. Example of binary support vector machine - Shark-ML.

```
#pragma once
#define SHARK_CV_VERBOSE 1
#include <inc/shark/printEvaluation.hpp>
#include <shark/Algorithms/KMeans.h>
#include <shark/Algorithms/Trainers/CSvmTrainer.h>
#include <shark/Data/Dataset.h>
#include <shark/Models/Classifier.h>
#include <shark/Models/Kernels/GaussianRbfKernel.h>
#include <shark/ObjectiveFunctions/Regularizer.h>

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

    // creating the kernel
    double gamma = 0.16;
    GaussianRbfKernel<> kernel(gamma);
    KernelClassifier<RealVector> svm;
    double regularization = 1;
    bool bias = true;
    // creating and configuring the model
    CSvmTrainer<RealVector> trainer(
        &kernel, regularization, bias);
    trainer.sparsify() = false;
    // training
    trainer.train(svm, trainData);
    // evaluation
    std::cout << "-----Shark␣SVM-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = svm(trainData.inputs());
    printSharkModelEvaluation(
        trainData.labels(), predictions);

    std::cout << "Test␣data:" << std::endl;
    predictions = svm(testData.inputs());
    printSharkModelEvaluation(
        testData.labels(), predictions);
}
```

Listing 5.11. Example of multiclass support vector machine - Shark[3].

```
using namespace shark;

void SVMClassification(const ClassificationDataset& train,
                       const ClassificationDataset& test,
                       unsigned int num_classes)
{
    double gamma = 0.5;
    GaussianRbfKernel<> kernel(gamma);
    // creating the composite model
    OneVersusOneClassifier<RealVector> ovo;
    // creating container for sub-problems
    unsigned int pairs = num_classes * (num_classes - 1) / 2;
    std::vector<KernelClassifier<RealVector>> svm(pairs);
    for (std::size_t n = 0, cls1 = 1; cls1 < num_classes; cls1++)
    {
        // creating a set of classifiers for a given class
        using BinaryClassifierType =
            OneVersusOneClassifier<RealVector>::binary_classifier_type;
        std::vector<BinaryClassifierType*> ovo_classifiers;
        for (std::size_t cls2 = 0; cls2 < cls1; cls2++, n++)
        {
            // creating a binary sub-problem
            ClassificationDataset binary_cls_data =
                binarySubProblem(train, cls2, cls1);
            // training the sub-model
            double c = 10.0;
            CSvmTrainer<RealVector> trainer(&kernel, c, false);
            trainer.train(svm[n], binary_cls_data);
            ovo_classifiers.push_back(&svm[n]);
        }
        // adding the set of classifiers to the main model
        ovo.addClass(ovo_classifiers);
    }
    // using the model
    auto predictions = ovo(test.inputs());
}
```

5.4.4. K-nearest neighbors algorithm

One of the classification methods provided by the Shark-ML library is the nearest neighbor model which can be equipped in various algorithms, one of which is the K-nearest neighbors algorithm (kNN). The model is represented by the *NEarestNeighborModel* class. The library allows for using a brute-force solution, or a solution based on space partition trees, using the *KDTree* class and *TreeNearestNeighbors*. In contrary to previously shown methods, performing a multiclass classification in this case does not require compound models or providing the number of classes to the model. It is automatically configured based on the provided training data. Listing 5.12 shows how to prepare the kNN classifier.

Listing 5.12. Example of kNN classifier - Shark-ML.

```cpp
#pragma once

#define SHARK_CV_VERBOSE 1
#include <inc/shark/printEvaluation.hpp>
#include <shark/Algorithms/KMeans.h>
#include <shark/Algorithms/NearestNeighbors/TreeNearestNeighbors.h>
#include <shark/Data/Dataset.h>
#include <shark/Models/Classifier.h>
#include <shark/Models/NearestNeighborModel.h>
#include <shark/Models/Trees/KDTree.h>
#include <iostream>

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

    // creating and configuring the tree and algorithm
    KDTree<RealVector> tree(trainData.inputs());
    TreeNearestNeighbors<RealVector, unsigned int> algorithm(
        trainData, &tree);

    // model configuration
    const unsigned int K = 2; // neighbor count for kNN algorithm
    NearestNeighborModel<RealVector, unsigned int> KNN(&algorithm, K);

    // evaluation
    std::cout << "-----Shark␣KNN-----" << std::endl;
    std::cout << "Train␣data␣:" << std::endl;
    auto predictions = KNN(trainData.inputs());
    printSharkModelEvaluation(
        trainData.labels(), predictions);

    std::cout << "Test␣data:" << std::endl;
    predictions = KNN(testData.inputs());
    printSharkModelEvaluation(
        testData.labels(), predictions);
}
```

5.4.5. Ensemble algorithm

Besides commonly known algorithms, the Shark-ML library also provides more complex structures, such as eg. ensemble algorithm models, which bases on using many component algorithms based on fragments on the characteristics space, in order to combine acquired results to increase the predictions precision. Unfortunately, the only mechanism present in this library that utilizes this technique is random forest composed of decision trees, allowing only for solving classification problems (it does not allow for using it for regression purposes). A classical approach for this library is taken, where the implementation is performed by creating a trainer object - in this case of *RFTrainer* class - which allows for configuring the parameters and subsequently training the model represented by the *RFClassifier* object. Besired the random forest algorithm, there is a possibility of using this library for creating a model in the stacking technique, however due to the fact that this option does not occur natively, it is outside of this dissertation scope. Listing 5.13 shows how to create and use the random forest model.

Listing 5.13. Creation of ensemble random forest model - Shark-ML[3].

```cpp
using namespace std;

void RFClassification(const shark::ClassificationDataset& train,
                      const shark::ClassificationDataset& test)
{
    using namespace shark;

    // creating and configuring the trainer
    RFTrainer<unsigned int> trainer;
    trainer.setNTrees(100);
    trainer.setMinSplit(10);
    trainer.setMaxDepth(10);
    trainer.setNodeSize(5);
    trainer.minImpurity(1.e-10);
    // creating the classifier
    RFClassifier<unsigned int> rf;
    trainer.train(rf, train);
    // evaluation
    ZeroOneLoss<unsigned int> loss;
    auto predictions = rf(test.inputs());
    double accuracy = 1. - loss.eval(test.labels(), predictions);
    std::cout << "Random␣Forest␣accuracy␣=␣" << accuracy << std::endl;
}
```

5.4.6. Neural network

Construction of neural network using the Shark-ML library utilizes some mechanisms offered by the *LinearModel<>* class. It allows for deciding the type and number of inputs, outputs and whether to use the bias term. Each layer consists of singular linear model, where the number of inputs describes the neurons count within the layer. Neuron activation function configuration happens during passing the data types to the model template. Full list of activation functions can be found in the library documentation [20]. The next step is to prepare an object of *ErrorFunction<>* class based on one of the available loss functions, which will be configured for use by optimizer which carries out the training process. After preparing the loss function, it needs to be initialized with random weights and the user needs to create and configure an optimizer object of choice. At this stage, the network is ready for training. The process is based around iteratively executing a *step()* function of the optimizer. Due to the need of using a user-defined loop, there is a possibility for describing user's own stop conditions to be evaluated in each epoch, such as epoch count or passing a certain threshold by the loss function. Inside the outside epoch loop, another loop needs to be placed that will go through each data batches, performing the optimizer step on them. After the training is completed, the model needs to be configured by passing the weights acquired by the optimizer to it. As a result, user gets a fully set up neural network model. Listing 5.14 shows the code snippets performing the whole process, which is an example from the handbook [3].

Listing 5.14. Example of neural network with two hidden layers for classification task - Shark[3].

```
#pragma once

#include <iostream>
#include <inc/shark/printEvaluation.hpp>
#include <shark/Algorithms/GradientDescent/SteepestDescent.h>
#include <shark/Models/ConcatenatedModel.h>
#include <shark/Models/LinearModel.h>
#include <shark/Data/Dataset.h>
#include <shark/ObjectiveFunctions/ErrorFunction.h>
#include <shark/ObjectiveFunctions/Regularizer.h>
#include <shark/ObjectiveFunctions/Loss/CrossEntropy.h>

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

    // defining network layers
    using DenseTanhLayer = LinearModel<RealVector, TanhNeuron>;
    using DenseLogisticLayer = LinearModel<RealVector, LogisticNeuron>;
    DenseTanhLayer layer1(inputDimension(trainData), 5, true);
    DenseTanhLayer layer2(5, 5, true);
    DenseLogisticLayer output(5, numberOfClasses(trainData), true);
    // connecting the layers
    auto network = layer1 >> layer2 >> output;
    // creating and configuring the loss function
    CrossEntropy<unsigned int, RealVector> loss;
    ErrorFunction<> error(trainData, &network, &loss, true);

    // regularization
    TwoNormRegularizer<> regularizer(error.numberOfVariables());
    double weightDecay = 0.01;
    error.setRegularizer(weightDecay, &regularizer);
    error.init();

    // weights initialization
    initRandomNormal(network, 0.001);
    // creating and configuring the optimizer
    SteepestDescent<> optimizer;
    optimizer.setMomentum(0.5);
    optimizer.setLearningRate(0.1);
    optimizer.init(error);
    // training
    std::size_t epochs = 1000;
    std::size_t iterations = trainData.numberOfBatches();
    // loop that goes through epochs
    for (std::size_t epoch = 0; epoch != epochs; ++epoch)
    {
        double avgLoss = 0.0;
        // loop that goes through batches
        for (std::size_t i = 0; i != iterations; ++i)
        {
            // performing optimizer step
            optimizer.step(error);
            // saving partial loss
            if (i % 100 == 0)
            {
                avgLoss += optimizer.solution().value;
            }
        }
        // calculating average loss value
        avgLoss /= iterations;
        std::cout << "Epoch␣" << epoch << "␣|␣Avg.␣loss␣" << avgLoss
                  << std::endl;
    }
    // configuring the final model
    network.setParameterVector(optimizer.solution().point);
    std::cout << "In␣Shape=" << network.inputShape() << std::endl;
    std::cout << "Out␣Shape=" << network.outputShape() << std::endl;
    Classifier<ConcatenatedModel<RealVector>> model(network);

    // validation
    std::cout << "-----Shark␣Neural␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    std::cout << "trainData.numberOfBatches()␣=␣" <<
            trainData.numberOfBatches() << std::endl;
    auto predictions = network(trainData.inputs());
    std::cout << "predictions.numberOfBatches()␣=␣" <<
            predictions.numberOfBatches();
    printSharkModelEvaluation(
        trainData.labels(), predictions);

    std::cout << "Test␣data:" << std::endl;
    predictions = network(testData.inputs());
    printSharkModelEvaluation(
        testData.labels(), predictions);
}
```

5.5. Model analysis methods

5.5.1. Loss functions

Shark-ML library offers a range of loss functions allowing for proper verification of model accuracy. Some of them are[21]:

- **mean absolute error** - available as *AbsoluteLoss* class;
- **mean squared error** - available as *SquaredLoss* class;
- **zero-one loss** - available as *ZeroOneLoss* class;
- **discrete loss** - available as *DiscreteLoss* class;
- **cross entropy** - available as *CrossEntropy* class;
- **hinge loss** - available as *HingeLoss* class;
- **squared hinge loss** - available as *SquaredHingeLoss* class;
- **epsilon hinge loss** - available as *EpsilonHingeLoss* class;

- **squared epsilon hinge loss** - available as *SquaredEpsilonHingeLoss* class;
- **Huber loss** - available as *HuberLoss* class;
- **Tukey biweight loss** - available as *TukeyBiweightLoss* class.

Each of the aforementioned classes is used in a schematic way, through the creation of the loss class object of choice, and subsequently calling it as a function, passing expected values and acquired predictions. Listing 5.15 presents the way to use a loss function via an example of mean squared error.

Listing 5.15. Mean squared error loss function use - Shark-ML.

```
using namespace shark;

// [...]

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

5.5.2. R^2 and adjusted R^2 metrics

Shark-ML does not implement direct representation of the R^2 metric, unlike the loss functions. However, it provides a function to calculate the variance of given data, which allows for fairly easy manual implementation of both metrics. Listing 5.16 shows how to calculate them, given the value of mean squared error is known.

Listing 5.16. Implementation of R^2 and adjusted R^2 metrics - Shark-ML.

```
using namespace shark;

// [...]

// mean squared error
SquaredLoss<> mse_loss;
auto mse = mse_loss(train_data.labels(), predictions);

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

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

5.5.3. ROC curve

The ROC curve is one of commonly used metrics for evaluating model's correctness, and as such, it could not be missed in the Shark-ML library. It is available under the *NegativeAUC* class, which can be used similarly to the loss functions. In contrary to the classic approach, this class calculates the compliment of the ROC curve, so it can be used as a minimized goal in the training process. Listing 5.17 shows how to use it to calculate the mentioned metric.

Listing 5.17. Example of calculating ROC curve metric - Shark-ML.

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

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

    // repackaging data from unsigned int to RealVector
    std::vector<RealVector> v;
    std::for_each(dataContainer.elements().begin(),
                  dataContainer.elements().end(),
                  [&](const auto e){
                      RealVector rv(1);
                      rv(0) = static_cast<double>(e);
                      v.emplace_back(rv); });
    return v;
}

inline void printSharkModelEvaluation(
    const shark::Data<unsigned int>& labels,
    const auto& predictions)
{
    using namespace shark;

    // preparing the auc roc solver
    constexpr bool invert = false;
    NegativeAUC<unsigned int, RealVector> auc(invert);

    // repackaging data
    auto predVec = repackToRealVectorRange(predictions);
    auto predData = createDataFromRange(predVec);
    // calculating auc roc
    auto roc = auc(labels, predData);
    std::cout << "ROC:␣" << (-1 * roc) << std::endl << std::endl;
}
```

5.5.4. K-fold cross validation

The process of acquiring the best hyperparameters value in Shark-ML consists of performing multiple internal training processes of given model, however it focuses on comparing the results, and because of that, its description was placed in this paragraph. In order to use the K-fold cross validation implementation provided by the library, three classes are required. First of them is *CVFolds*, which is tasked with storing data that is split into a number of segments. The second one is *CrossValidationError* - a template accepting model type, for which the validation error will be calculated, and an object of the loss function, which will perform the calculation.

The last but not least is *GridSearch*, which iteratively selects fragments for the training process and hyperparameters for the model. This procedure results in acquiring the best set of hyperparameters for training of the model. User needs to invoke the *step()* method of the *GridSearch* object only once. Listing 5.18 shows an example of this mechanism from the handbook[3], where the author uses the aforementioned classes with manually implemented polynomial regression model.

Listing 5.18. Example of K-fold cross validation in Shark-ML[3].

```
using namespace shark;

// [...]

// processing the training data
const unsigned int num_folds = 5;
CVFolds<RegressionDataset> folds =
    createCVSameSize<RealVector, RealVector>(train_data, num_folds);

// preparing the target parameters for the model
double regularization_factor = 0.0;
double polynomial_degree = 8;
int num_epochs = 300;

// configuring the target model
PolynomialModel<> model;
PolynomialRegression trainer(regularization_factor, polynomial_degree,
                             num_epochs);

// creating the loss object and cross validation
AbsoluteLoss<> loss;
CrossValidationError<PolynomialModel<>, RealVector> cv_error(
    folds, &trainer, &model, &trainer, &loss);

// generating a grid
GridSearch grid;
std::vector<double> min(2);
std::vector<double> max(2);
std::vector<std::size_t> sections(2);

// regularization
min[0] = 0.0;
max[0] = 0.00001;
sections[0] = 6;

// polynomial degree
min[1] = 4;
max[1] = 10.0;
sections[1] = 6;
grid.configure(min, max, sections);

// training and configuration process
grid.step(cv_error);
trainer.setParameterVector(grid.solution().point);
trainer.train(model, train_data);
```

5.6. Documentation and sources availability

The Shark-ML library has a shortened documentation available on the main website of the project, alongside example source code files added to the repository. It is also mentioned in the handbook *"Hands-On Machine Learning with C++"*[3], which shows examples of using selected functionalities. What makes it stand out from all the libraries described in this dissertation however is the fact, that it is dedicated for the C++ language, and as such, community threads and articles describing creation of various types of models with it, and sample source codes are way more available. The main page documentation is clear and extensive enaugh to make using the available API way easier.

Chapter 6

Dlib

6.1. Introduction

Dlib is a machine learning library written in modern C++, made for industrial and scientific purposes[22]. Similarly to previously discussed libraries, it has an open source using *Boost Software License*[23]. Among the fields in which this library is utilized are robotics, embedded systems, telecommunication and high performance software. The project's code is equipped with unit tests, which allows for easier maintaining the quality of provided solutions. Interesting is the fact, that Dlib is not only a library, but also a toolbox, which provides functionalities beyond the scope of machine learning.

6.2. Data formats

Dlib uses C++ standard STL library for representing vectors within its code. Additionally, there is a possibility of initializing them using the comma operator, and wrapping a raw array. This means, that similarly to the Shogun library, data can be stored and passed to the program in any way that puts them inside eg. a raw array, to later process it into form accepted by the library. The raw array method works also with the STL containers which allow access to their internal buffers in the array form via the *data()* method. Similarly to previous solutions, Dlib also provides the CSV format support, which is subject to the same restrictions as in Shogun's case. It is provided by an overloaded stream operator which accepts an *std::ifstream* object of the C++ standard library. An example source code showing this mechanism is presented on listing 6.1.

Listing 6.1. Code snippet illustrating read of CSV file - Dlib[3].

```
#include <Dlib/matrix.h>
#include <fstream>
#include <iostream>

using namespace Dlib;

// [...]

matrix<double> data;
```

```cpp
std::ifstream file("data_file.csv");
file >> data;
std::cout << data << std::endl;
```

6.3. Data processing and exploration

6.3.1. Normalizing

The library provides data normalization via standardization, performed by the *Dlib::vector_normalizer* class. The main constraint of its use is the fact, that the whole training data cannot be placed in it at once, forcing the user to separate it into a number of vectors, and after the normalizing, placing them back together for further processing. An example was shown on listing 6.2.

Listing 6.2. Normalizer function - Dlib.

```cpp
#pragma once

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

inline std::vector<dlib::matrix<double>> dlibNormalize(
    const std::vector<dlib::matrix<double>& data)
{
    using namespace dlib;
    // creating and training the normalizer
    vector_normalizer<matrix<double>> normalizer;
    normalizer.train(data);
    // processing data
    std::vector<matrix<double>> processedData(data.size());
    for (auto dataMatrix : data)
        processedData.emplace_back(normalizer(data));
    return processedData;
}
```

6.3.2. Dimensionality reduction

6.3.2.1. Principal component analysis

The Principal component analysis method implementation in the Dlib library is offered via *dlib::vector_normalizer_pca* class, which besides the dimensionality reduction also performs data normalization. It proves useful, as it guarantees that the reduction will be performed on correctly prepared observation values. Listing 6.3 shows a function that uses the aforementioned method.

Listing 6.3. Example of dimensionality reduction with PCA - Dlib.

```cpp
#pragma once

#include <dlib/matrix.h>
#include <dlib/matrix/matrix_utilities.h>
```

```cpp
#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;
    // creating and training the reductor
    vector_normalizer_pca<matrix<double>> pca;
    pca.train(data, desiredDimensions/data[0].nr());
    // preparing container for transformed data
    std::vector<matrix<double>> processedData;
    processedData.reserve(data.size());
    // data transformation
    for(auto& sample : data)
    {
        processedData.emplace_back(pca(sample));
    }
    return processedData;
}
```

6.3.2.2. Linear discriminant analysis

The second of provided algorithms in Dlib is linear discriminant analysis. It is available as a *dlib::compute_lda_transform*, which translates the matrix containing observations into a data transformation matrix. Due to the supervised nature of the algorithm, it is necessary to provide the values of labels, however the input data itself, in contrary to the PCA method, can be contained within a singular matrix object. Reduction is performed by multiplying the acquired matrix with the transposed vector containing an observation. Precise algorithm application was shown on listing 6.4.

Listing 6.4. EXample of dimensionality reduction with LDA - Dlib.

```cpp
#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;
    // creating objects needed for reduction
    matrix<double, 0, 1> mean;
    auto transform = data;
    // converting data into reduction matrix
    compute_lda_transform(transform, mean, labels, desiredDimensions);
    // preparing container for transformed data
    std::vector<matrix<double>> transformedData;
    transformedData.reserve(data.nr());
    // dimensionality reduction
    for (long i = 0; i < data.nr(); ++i)
    {
        transformedData.emplace_back(transform * trans(rowm(data, i)) - mean);
    }
    // returning the transformed rows vector
    return transformedData;
}
```

6.3.2.3. Sammon mapping

One algorithm that makes the Dlib library stand out among the others is an implementation of dimensionality reduction method via multidimensional scaling using a nonlinear algorithm - so called Sammon mapping[24]. The whole algorithm is implemented via the *dlib::sammon_projection* class, and its use comes down to creating its instance. In order to use this method, the user needs to invoke the *operator()*, providing the data vector and expected number of dimensions, receiving the transformed data. According to this, the exact use of the reduction function is limited to two lines of code. Precise example is shown by listing 6.5.

Listing 6.5. Example of dimensionality reduction with Sammon mapping - Dlib.

```
#pragma once

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

inline std::vector<matrix<double>> dlibSammon(
    const std::vector<matrix<double>>& data,
    long desiredDimensions)
{
    using namespace dlib;
    // creating the mapping object
    dlib::sammon_projekction sammon;
    // dimensionality reduction
    return sammon(data, desiredDimensions);
}
```

6.3.3. L2 regularization

The Dlib library contains a trainer function which allows for using ridge regression, which performs L2 regularization, called *rr_trainer* for linear regression and *krr_trainer* for non-linear regression. Examples of use of each one of those methods were shown in the paragraphs describing linear and ridge kernel regression models.

6.4. Machine learning models

6.4.1. Linear regression

The realization or linear regression model within the Dlib library is indirect. It is performed by using a kernel ridge regression, passing a linear kernel. Subsequently the training process is carried out, storing the desired model into a functor. Listing 6.6 shows the details of aforementioned mechanism.

Listing 6.6. Example of linear regression - Dlib.

```cpp
#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;
    // creating and configuring the trainer and kernel
    using linearKernel = linear_kernel<matrix<double, 5, 1>>;
    krr_trainer<linearKernel> trainer;
    trainer.set_kernel(linearKernel());
    // training
    decision_function<linearKernel> model = trainer.train(
        trainData, trainLabels);
    // evaluation
    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. Support vector machine

In order to perform a multiclass classification using support vector machine, the Dlib library provides a decision function class called *dlib::one_vs_one_decision_function*.

It stores the resulting model of support vector machine inside the *one_versus_one* class, two which an SVM trainer is passed. Detailed example is shown on listing 6.7.

Listing 6.7. Example of support vector machine - Dlib.

```cpp
#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>>;
    // creating support vector machine trainer
    svm_nu_trainer<Kernel> svmTrainer;
    svmTrainer.set_kernel(Kernel(0.16));
    // creating multiclass classifier trainer
    OVOTrainer trainer;
    trainer.set_trainer(svmTrainer);
    // creating the model
    one_vs_one_decision_function<OVOTrainer> model =
        trainer.train(trainData, trainLabels);
    // evaluation
    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. Neural network

Neural network construction in case of Dlib begins with defining the network's architecture, using special template chain. The parameters create a network of nested layers, from the most inner one to the most outer one. Dlib provides the user with a separation of layer type from its activation functions, allowing user to precisely configure the networks characteristics. However, the syntax of created architecture is very clouded, which significantly complicates its analysis. After the architecture is created, the user needs to create and configure a series of solvers. The most common one offed by the library is stochastic gradient descent algorithm, implemented under the *dlib::sgd* class. The third step is to configure a deep neural network trainer, provided by the *dlib::dnn_trainer* class, by setting parameters such as:

- learning rate coefficient;
- learning rate change coefficient;
- batch size;
- max epoch count.

During the creation of the trainer object, it accepts a reference to the network architecture and solver object. Training process is carried out by invoking the *train()* method, and the resulting model is written in-place into the network architecture object. Listing 6.8 shows an example of building a neural network using this library.

Listing 6.8. Neural network example - Dlib.

```cpp
#pragma once

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

#include <inc/dlib/eval.hpp>

#include <vector>

inline void dlibNeural(
    std::vector<dlib::matrix<double>> trainData,
    std::vector<dlib::matrix<double>> testData,
    std::vector<double> trainLabels,
    std::vector<double> testLabels)
{
    using namespace dlib;
    // defining the network architecture
    using Architecture = loss_mean_squared<fc <1,
                            htan<fc<5,
                            htan<fc<5,
                            input<matrix<double>>>>>>>>;
    // creating the network
    Architecture model;
    // creagting and configuring the optimizing algorithm
    float weightDecay = 0.0001f;
    float momentum = 0.5f;
    sgd solver(weightDecay, momentum);
    // creating and configuring the trainer
    dnn_trainer<Architecture> trainer(model, solver);
    trainer.set_learning_rate(0.1);
    trainer.set_learning_rate_shrink_factor(1);
    trainer.set_mini_batch_size(64);
    trainer.set_max_num_epochs(100);
    trainer.be_verbose();
    // training
    trainer.train(trainData, trainLabels);
    model.clean();
    // evaluation
    std::cout << "-----␣Dlib␣Neural␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = model(trainData);
    dlibEval(predictions, trainLabels, Task::CLASSIFICATION);

    std::cout << "Test␣data:" << std::endl;
    predictions = model(testData);
    dlibEval(predictions, testLabels, Task::CLASSIFICATION);
}
```

6.4.4. Kernel ridge regression

Preparation of multiclass kernel ridge regression model in case of Dlib library is almost identical to the way of makign a multiclass support vector machine. The main difference is used trainer object, in this case of *dlib::krr_trainer* class. It is also possible to use the same kernel, as in support vector machine case. Listing 6.9 describes an example of the model use.

Listing 6.9. Kernel ridge regression example - Dlib.

```
#pragma once

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

#include <inc/dlib/eval.hpp>

#include <vector>

inline void dlibKRR(
    std::vector<dlib::matrix<double>> trainData,
    std::vector<dlib::matrix<double>> testData,
    std::vector<double> trainLabels,
    std::vector<double> testLabels)
{
    using namespace dlib;

    using OVOTrainer = one_vs_one_trainer<any_trainer<double>>;
    using Kernel = radial_basis_kernel<double>;
    // creating support vector machine trainer
    krr_trainer<Kernel> krrTrainer;
    krrTrainer.set_kernel(Kernel(0.1));
    // creating multiclass classifier trainer
    OVOTrainer trainer;
    trainer.set_trainer(krrTrainer);
    // creating model
    one_vs_one_decision_function<OVOTrainer> model =
        trainer.train(trainData, trainLabels);
    // evaluation
    std::cout << "-----␣Dlib␣KRR␣-----" << std::endl;
    std::cout << "Train␣data:" << std::endl;
    auto predictions = std::vector<double>(trainData.size());
    for (auto& sample : trainData)
```

```cpp
    {
        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);
}
```

6.5. Model analysis method

6.5.1. Pole pod wykresem krzywej charakterystycznej odbiornika

The Dlib library contains implementation for a function that calculates ROC curve, however it requires a certain degree of data processing both before and after its use. In order to use it, data needs to be split into correctly and incorrectly classified. The result of this mechanism is a vector containing coordinates of points of the ROC curve, which allow the user to plot the curve. The area under the curve needs to be calculated manually, utilizing eg. one of the numerical integration methods. Listing 6.10 shows a function that calculates the values of model predictions, including calculation of the area under the ROC curve for classification task, by performing numerical integration via a trapezoid method.

Listing 6.10. Calculating area under the ROC curve - Dlib.

```cpp
#pragma once

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

enum class Task
{
    CLASSIFICATION,
    REGRESSION
};

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

    if (task == Task::CLASSIFICATION)
    {
        // preparing data containers
        std::vector<double> correct;
        std::vector<double> incorrect;
        // preparing detector values
        constexpr double positiveDetectionScore = 0.75;
        constexpr double negativeDetectionScore = 0.25;
        // data split
        for (int i = 0; i < predictions.size() && i < labels.size(); ++i)
        {
            if (predictions[i] == labels[i])
                correct.emplace_back(positiveDetectionScore);
            else
                incorrect.emplace_back(negativeDetectionScore);
        }
        // calculating roc curve
        auto roc = compute_roc_curve(correct, incorrect);
        // calculating auc roc
        double aucRoc = 0.0;
        for (int i = 0; i < roc.size() - 1; i++)
        {
            if (roc[i+1].false_positive_rate != 0.0)
            {
                aucRoc += (roc[i].true_positive_rate + roc[i+1].true_positive_rate)
                        * (roc[i+1].false_positive_rate - roc[i].false_positive_rate) / 2;
            }
        }
        std::cout << "AUC␣ROC:␣" << aucRoc << std::endl;
    }
    else
    {
        // calculating the sum of squared differences
        auto sum = 0.0;
        for (int i = 0; i < predictions.size() && i < labels.size(); ++i)
        {
            sum += pow(labels[i] - predictions[i], 2);
        }
        // calculating the mean squared error
        auto mse = sqrt(sum / predictions.size());
        std::cout << "MSE:␣" << mse << std::endl;
    }
}
```

6.5.2. K-fold cross validation

Carring out a K-fold cross validation using Dlib is a complex process. It is multiphase, beginning with defining own function calculating a metric that interests the user, based on the *dlib::cross_validate_regression_trainer()*. The part of acquiring target hyperparameter values is achieved via *dlib::find_min_global()* function, which accepts an address of the optimizer function, containers storing the information about minimal and maximal allowed values for each hyperparameter and the number of allowed calls to the optimizer. In order to read the resulting values, the user needs to read subsequent fields of the x member of the structure returned by *find_min_global()*. The details were presented on an example from the handbook [3] on listing 6.11.

Listing 6.11. Example of K-fold cross validation - Dlib.

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

```cpp
#include <cmath>

inline void dlibCrossValidate(std::vector<dlib::matrix<double>> data,
                              std::vector<double> labels)
{
    using namespace dlib;
    // splitting data
    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());
    // creating cross validation function
    auto crossValidationScore = [&](const double gamma, const double c,
                                    const double degreeIn)
    {
        using namespace dlib;

        auto degree = std::floor(degreeIn);
        // defining the kernel
        using Kernel = polynomial_kernel<double>;
        // creating and configuring the trainer
        svr_trainer<Kernel> trainer;
        trainer.set_kernel(Kernel(gamma, c, degree));
        // calculating the cross validation metric
        matrix<double> result = cross_validate_regression_trainer(
            trainer, trainData, trainLabels, 10);
        // returning the mean squared error
        return result(0, 0);
    }
    // carrying out the optimization
    auto result = find_min_global(
        crossValidationScore,
        {0.01, 1e-8, 5}, // minimal value
        {0.1, 1, 15},    // maximal value
        max_function_calls(50));
    // reading the determined hyperparameters
    auto gamma = result.x(0);
    auto c = result.x(1);
    auto degree = result.x(2);
    // creating model based on determined hyperparameters
    using Kernel = polynomial_kernel<double>;
    svr_trainer<Kernel> trainer;
    trainer.set_kernel(Kernel(gamma, c, degree));
    auto model = trainer.train(trainData, trainLabels);

    // evaluation
    std::cout << "-----␣Dlib␣CrossValidated␣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::REGRESSION);
    predictions.clear();
    std::cout << "Test␣data:" << std::endl;
    for (auto& sample : testData)
    {
        predictions.emplace_back(model(sample));
    }
    dlibEval(predictions, testLabels, Task::REGRESSION);
}
```

6.6. Documentation and sources availability

Dlib contains a set of examples in the form of source code snippets presenting various mechanisms, available on the main page of the project [**dlib:home**]. It is also one of the main libraries described within the aforementioned handbook [3]. Unfortunately, majority of the community forums focus on using Dlib via its Python interface, which can make searching for specific problems much harder. It is worthy to mention, that besides machine learning functionalities, Dlib also offers other solutions, such as networking, which makes the navigation across the project site slightly more difficult due to the presence of potentially uninteresting features.

Chapter 7

Group comparison and summary

7.1. Offered functionalities

Libraries discussed within this dissertation differ in types and numbers of offered functionalities. The table 7.1 summarizes each aspects shown in the previous chapters.

Functionality	Library		
	Shogun	Shark-ML	Dlib
Reading data	std::vector, support for CSV format	raw arrays, support for CSV format, support for HTTP	std::vector, support for CSV format
Normalizacja	min-max	unit interval, unit variance, zero mean and selected variance	standardization
Dimensionality reduction	PCA, Kernel PCA, MDS, IsoMap, ICA, Factor analysis, t-SNE	PCA, Linear discriminant analysis	PCA, Linear discriminant analysis, Sammon mapping
Regularization	L1 i L2 automatic	L1 i L2	L2
Linear regression	Yes	Yes	Yes
Logistic regression	Yes	Yes	No
Support vector machine	Yes	Yes	Yes
K-nearest neighbors algorithm	Yes	Yes	No
Ensemble algorithm	Gradient boosting, random forest	Random forest	No
Neural network	Yes	Yes	Yes

Kernel ridge regression	No	No	Yes
Mean squared error	Yes	Yes	No
Mean absolute error	Yes	Yes	No
Zero-one error	No	Yes	No
Discrete error	No	Yes	No
Cross entropy	No	Yes	No
Hinge loss	No	Yes	No
Mean squared hinge loss	No	Yes	No
Epsilon hinge loss	No	Yes	No
Mean squared epsilon hinge loss	No	Yes	No
Huber loss	No	Yes	No
Tukey loss	No	Yes	No
Logarithmic loss	Yes	No	No
R^2 metric	Yes	Yes	No
Accuracy	Yes	No	Yes
Area under the ROC curve	Yes	Yes	Yes*
K-fold cross validation	Yes	Yes	Yes

Table 7.1. Group comparison of the functionalities.

** - need to calculate the area under ROC curve using data points of the function plot.*

By analyzing the data collected in the table 7.1 it can be noticed, that the Shogun and Shark-ML libraries are fairly similar with types and number of available machine learning methods, however Shark-ML contains significantly more loss function types which can be used, allowing for more freedom when customizing the training process. The least amount of functionalities is displayed by the Dlib library, which contain only a subset of the algorithms present in the other projects. What is more, it has very limited model quality analysis possibilities, requiring making of own procedures for data processing by the user, such as eg. area under ROC curve calculation procedure.

7.2. Comparison of the results from test examples

In order to test the proficiency of each library, they were used for creating and evaluating linear regression and support vector machine models for the datasets discussed in chapter 3. Each model uses 80% of the dataset for training, and the remaining 20% as validation. Table 7.2 shows group comparison of achieved results. Output of the programs utilizing the libraries was presented on figures 7.1, 7.2 and 7.3.

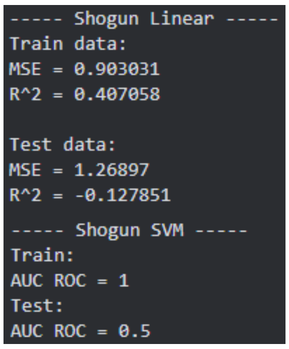
Library	Machine learning method	
	Linear regression	Support vector machine

Shogun	Training:
	MSE = 0,903031; R^2 = 0,407058
	Validation:
	MSE = 1,26897; R^2 = -0.127851
		Training:
		AUC ROC: 1
		Validation:
		AUC ROC: 0,5

Shark	Training:
	MSE = 0,38728; R^2 = 0,745707
	Validation:
	MSE = 0,449118; R^2 = 0,60082
		Training:
		AUC ROC: 1
		Validation:
		AUC ROC: 0,5

Dlib	Training:
	MSE = 3,08953;
	Validation:
	MSE = 1,33014;
		Training:
		AUC ROC: 1
		Validation:
		AUC ROC: 1

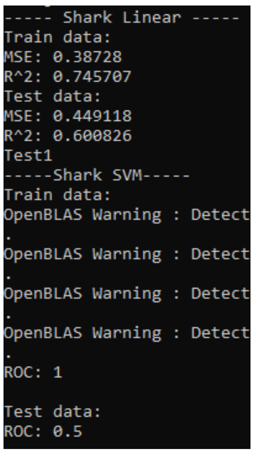
Table 7.2. Summary of achieved model quality results.



```
----- 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
```

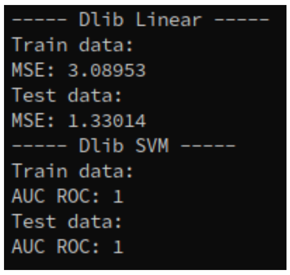
Figure 7.1. Output for Shogun library



```
----- 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. Output for Shark-ML library



```
----- 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. Output for Dlib library

By analyzing the above results for linear regression, it can be concluded that the first place was taken by the Shark-ML library, which is characterized by the least values of the mean squared error. On second place is Shogun library. It can be noticed that the R^2 calculation mechanism crossed outside the allowed interval for the metric for validation data. It is suspected that it is a result of cumulative floating point representation errors during calculations, pointing that the mechanism of calculating the R^2 metric components may be numerically unstable. The last place was taken by Dlib, in case of which the value of mean squared error exceeded 3 units.

Much worse results were acquired for the classification task using the aforementioned libraries. Each of them is characterized by seemingly perfect fit to the training data, and in case of Shark-ML and Shogun libraries, a fit equal to coin flip for the validation data. The Dlib library additionally achieved the area under the ROC curve equal to 1 for the validation data. Above results are considered impossible to achieve, which was proven in chapter 3 section 3.3.1 using dedicated statistical software. Because of that, and analysis of the causes was undertaken.

In order to improve the acquired results and to solve encountered errors it was decided to normalize the predictor values for both datasets into an interval of $\langle 0; 1\rangle$ using JMP software, and subsequently to analyze the code which performs the support vector machine training for each of the libraries. First of them was Shark-ML, where manual tuning of gamma and regularization parameters was applied. It turned out that after data normalization, setting gamma parameter to 0.16 and setting regularization to 1 a correct and plausible result was achieved, where the area under the ROC curve is equal to 0.933129 for the validation data. Next the Shogun library was tested. Unfortunately, despite data normalization, the problem of the R^2 metric value crossing boundaries for validation data was not solved. However, a successful classification was performed, acquiring the value of the area under the ROC curve equal to 0.638112. It is important to notice, that the normalization process made the achieved regression results worse for the Shogun library, decreasing the R^2 metric for training data, and making the value of this metric for validation data stray even further from its correct boundaries. The last of the libraries was Dlib, for which the data normalization significantly improved acquired mean square error values, however the support vector machine still achieved a perfect fit, which can point to the fact, that the mechanism might be implemented incorrectly. Table 7.3 summarizes the results achieved after code analysis and data normalization. Output of the programs was presented on figures 7.4, 7.5 and 7.6.

Library	Machine learning method	
	Linear regression	Support vector machine

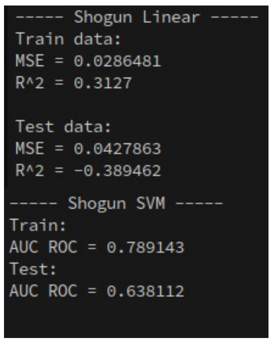
Shogun	Training:
	MSE = 0,0286481; R^2 = 0,3127
	Validation:
	MSE = 0.0427863; R^2 = -0.389462
		Training:
		AUC ROC: 0,789143
		Validation:
		AUC ROC: 0,638112

Shark	Training:
	MSE = 0,0105995; R^2 = 0,745707
	Validation:
	MSE = 0.0122919; R^2 = 0,600826
		Training:
		AUC ROC: 0,921643
		Validation:
		AUC ROC: 0,933129

Dlib	Training:
	MSE = 0,481402;
	Validation:
	MSE = 0,179538;
		Training:
		AUC ROC: 1
		Validation:
		AUC ROC: 1

Table 7.3. Summary of achieved model quality results after analysis.

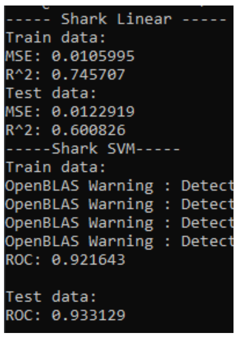
To summarize, it was noticed that each of the libraries is especially sensitive to the value ranges of the predictors, and in order to properly use them, in most cases it is needed that the predictors values are contained within the interval of $\langle 0; 1\rangle$. It was especially visible in case of the classification dataset, where due to the inverse Arrhenius transformation there was a significant discrepancy between the value range of those predictors, and the intervals of other predictors. Additionally it was noticed, that the mechanism of calculating the R^2 metric within the Shogun library and support vector machine in Dlib library likely contain implementation errors, so it is recommended to either not use them, or to have a very limited trust to them. Among the achieved results, the best fit was present in case of Shark-ML library, with Shogun on second place and Dlib on the last place.



```
----- Shogun Linear -----
 Train data:
 MSE = 0.0286481
 R^2 = 0.3127

 Test data:
 MSE = 0.0427863
 R^2 = -0.389462
----- Shogun SVM -----
Train:
AUC ROC = 0.789143
Test:
AUC ROC = 0.638112
```

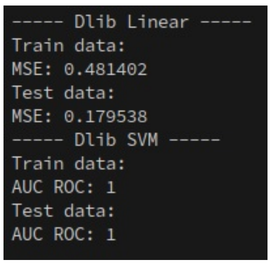
Figure 7.4. Output for Shogun library after analysis



```
----- Shark Linear -----
Train data:
MSE: 0.0105995
R^2: 0.745707
Test data:
MSE: 0.0122919
R^2: 0.600826
-----Shark SVM-----
Train data:
OpenBLAS Warning : Detect
OpenBLAS Warning : Detect
OpenBLAS Warning : Detect
OpenBLAS Warning : Detect
ROC: 0.921643

Test data:
ROC: 0.933129
```

Figure 7.5. Output for Shark-ML library after analysis



```
----- Dlib Linear -----
Train data:
MSE: 0.481402
Test data:
MSE: 0.179538
----- Dlib SVM -----
Train data:
AUC ROC: 1
Test data:
AUC ROC: 1
```

Figure 7.6. Output for Dlib library after analysis

7.3. Sources quality and needed effort

During working with each of the libraries it was noticed, that the least amount of effort was needed when working with Shark-ML library. It is due to the very user-friendly syntax and precise documentation available on the project's main website, alongside use examples for each of the methods. The library was also effortlessly build and installed in the Ubuntu 22.04 system in the WSL2 environment, allowing to quickly move along with the research.

The second library in the effort metric turned out to be the Dlib toolbox. It has its own project website with all the available classes and functions listed, however the description of each method is very short, and there is a lack of available examples. The syntax can prove challenging to the user, as it is not always obvious and sometimes makes the analysis of the performed operations harder.

The most effort-intensive library was Shogun. During the making of this dissertation, both official repository of the project, and the repository of Ubuntu operation system turned out to be incomplete. It rendered installing the library using the in-built packet manager impossible, aswell as building it due to the not fixed dependencies to moved external repositories. Despite slightly more user-friendly syntax than the Dlib library, previously mentioned issue caused that in order to download the library, it was necessary to install a special packet manager called *nix* which contain an older version of the Shogun project in its own repository. Because of the lack of documentation and the fact, that the generated examples do not use API provided by the library, and instead use it in completely separate, unnatural for the project way, researching the functionalities and the way to implement each of the machine learning tasks had to be based almost solely on the book sources. It significantly impares and prolongs the process of utilizing it into any project.

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

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

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