BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

APPLICATION OF MACHINE LEARNING ALGORITHMS FOR GENERATION OF CHECKING CIRCUITS

VYUŽITÍ ALGORITMŮ STROJOVÉHO UČENÍ PRO KONSTRUKCI HLÍDACÍCH OBVODŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR Bc. OLIVÉR LELKES
AUTOR PRÁCE

SUPERVISOR Ing. JAN KAŠTIL, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

This thesis deals with the application of machine learning algorithms for generation of online checking circuits. It contains description of the principles of checking circuits and presents existing checking circuit implementations relevant to this thesis. The work is focused on applying checking circuits on hardware components with sequential logic. Machine learning algorithms are trained on data sets consisting of the hardware components' input-output sequences, stored as time series data. Processing time series requires special type of machine learning algorithms, which are described and compared. The individual algorithms are utilized as machine learning classifiers in order to determine their suitability for use in checking circuits. The experiments of the thesis were performed on a low-pass FIR filter. The settings of the employed machine learning classifiers are presented and the results with the individual classifier settings are evaluated. Based on the obtained results it is discussed which machine learning algorithms are applicable in checking circuits.

Abstrakt

Tato diplomová práce se zabývá využitím algoritmů strojového učení pro konstrukci hlídacích obvodů. Práce obsahuje popis principů hlídacích obvodů, jejich existující implementace a ostatní teoretické znalosti vztahující se k systémům odolným proti poruchám. Práce je zaměřena na aplikaci hlídacích obvodů na hardware komponentech se sekvenční logikou. Algoritmy strojového učení jsou trénovány pomocí datových množin, které se skládají ze vstup-výstup sekvencí hardwarových komponentů a ukládají se jako časové řady. Cílem práce je určení vhodnosti jednotlivých algoritmů pro jejich aplikaci v hlídacích obvodech. Pro dosažení tohoto cíle, bylo provedeno srovnání vybraných algoritmů strojového učení. Součástí práce je popis parametrů algoritmů a generování datových sad. Práce taktéž zahrnuje experimenty provedeny na dolnopropustném FIR filtru a jejich vyhodnocení. Podle výsledků experimentů je diskutováno, které algoritmy jsou použitelné v hlídacích obvodech.

Keywords

machine learning, checking circuits, classification, supervised learning, time series, FIR filter

Klíčová slova

strojové učení, hlídací obvody, detekce chyb, klasifikace, učení s učitelem, časové řady, FIR filter

Reference

LELKES, Olivér. *Application of Machine Learning Algorithms for Generation of Checking Circuits.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Kaštil Jan.

Application of Machine Learning Algorithms for Generation of Checking Circuits

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Mr. Jan Kaštil. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Olivér Lelkes
May 24, 2017

Acknowledgements

I would like to thank Mr. Jan Kaštil for his help during solving the thesis, for his valuable guidance and hints and time spent on consultations.

Contents

Chapter 1

Introduction

Technology of digital systems develops at a very quick pace. The endeavour to increase logic density and reduce power consumption requires more and more smaller transistors on the chips. However, decreasing the size of transistors can lead to negative effects on the system's reliability. Due to sensitivity to radiation, systems are more prone to faults, which is why ensuring reliability is necessary. High reliability is essential in many fields of life and science: healthcare [27], spacecraft [2] or aircraft [52], etc. The focus of this thesis is on improving reliability by utilizing *fault tolerance* methods. Systems employing fault tolerance operate correctly also in the presence of faults [3].

In this thesis the technique of *checking circuits* is used for achieving fault tolerance. Checking circuit, also denoted as online checker, operates as a supervisor of a given hardware component which is desired to be fault-tolerant. The aim of the online checker is to detect faults occurring in the supervised hardware component. After a successful fault detection a corrective action can be taken.

The proposed checking circuit of this thesis contains the implementation of a certain *machine learning* algorithm. Machine learning is a branch of artificial intelligence[1] that deals with the construction and study of systems that can learn from data. The machine learning algorithm (classifier) present in the checking circuit, is taught (trained) on the observed hardware unit's input-output data. After the learning process finished, the classifier should be able to determine whether the observed hardware component works correctly or a fault has occurred. The fault detection is based on the hardware unit's recent input-output values, which are classified by the machine learning classifier into two main classes: 0 (faulty), 1 (correct).

The main objective of this thesis is to assess which machine learning algorithms are suitable to use for construction of online checkers. Secondly, it is aimed to determine what settings the chosen algorithms require in order to maximize their efficiency and accuracy of fault detection.

This thesis is focused on hardware components with sequential logic, whose output values are dependent not only on the present input values of the unit, but also on a certain number of past input values. The subsequent input-output values of the hardware unit constitute a *time series.* Time series consists of observations (input-output values) which were taken sequentially in time. For handling time series data special machine learning algorithms are needed, which are studied and described in this paper.

[1]Part of computer science that is concerned with making computer programs that can solve problems creatively and behave like humans.

Chapter 2 deals with the principles of checking circuits and with the basic concepts of fault tolerance. The ensuing chapter 3 introduces machine learning and its basic approaches. The chapter also describes the theoretical properties of the machine learning algorithms which were chosen to perform the experiments of this thesis. The application of the individual machine learning algorithms for checking circuit generation is presented in chapter 4. In this chapter the format of the used data sets and the process of data set generation, including fault injection are also explained. Chapter 5 contains the description of the experiments, performed on a finite impulse response filter. The results are evaluated and discussed in each experiment individually. The last chapter 6 contains the summary of this thesis, including final evaluation of the results and plans for future work.

Chapter 2

Principle of checking circuits used in field programmable gate arrays

Field Programmable Gate Arrays (FPGAs) are integrated circuits containing an array of logic blocks. The blocks can be programmed using a hardware description language (VHDL, Verilog, etc.) after the manufacturing of the FPGA. They can also include memory units, like flip-flops or more complex blocks of memory. The blocks' configuration can be changed by partial reconfiguration which makes FPGA a flexible device. FPGA offers high computational power, fast time-to-market and relatively cheap production. Another advantage of FPGAs is that they are suitable for developing checking circuits (see [28] and [46]).

Checking circuit, also called as *online checker* (see figure 2.1) works as a supervisor on a chosen function unit where it can validate the unit's outputs depending on its inputs. Any occurring errors are reported in real time which is why their operation is also described as *Concurrent Error Detection* (CED). This fault-detection process plays a major role in making the system more dependable. The concepts of dependability are further presented in section 2.1.

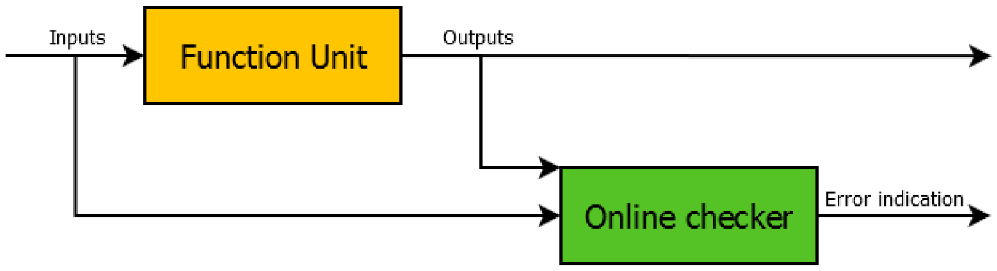


Figure 2.1: Online checker

Achieving fault tolerance requires not only to detect faults, but also to recover from the occurred error. Checking circuits which have this capability are suitable for implementing *fault-tolerant systems* (FTS). The properties and realization of FTS are described in section 2.2.

Section 2.3 deals with the specific utilizations of checking circuits used in FPGA for constructing FTS.

2.1 Concepts of dependability

The primal goal of checking circuits is to make the supervised system as dependable as possible. Dependability has 3 fundamental elements: *attributes*, *threats* and *means*. These are described based on [4] and [8].

2.1.1 Attributes of dependability

The attributes of dependability are properties of a system which can be assessed in order to determine the overall dependability of a system. According to [8] there are 3 primal attributes of dependability:

- *availability* - the system must be always ready for providing the correct service
- *reliability* - the provided service must be continuous
- *safety* - the provided service must not have catastrophic consequences on the user and the environment

Other possible attributes (without being exhaustive) [4]:

- *maintainability* - the system can be repaired and modified
- *testability* - the system can be tested, giving confidence about correctness
- *confidentiality* - absence of unauthorized disclosure of information
- *integrity* - absence of improper system state alterations

The significance of the individual attributes relies upon the given system's priorities. For example in case of a pacemaker device the continuous functioning (reliability) is crucial, while for a nuclear power plant system, safety is of greater importance.

2.1.2 Threats to dependability

The threats to dependability which are also called as dependability impairment, are usually defined in terms of *faults*, *errors* and *failures*.

A *fault* is physical defect, imperfection or flaw that occurs in some hardware or software component. Faults can be distinguished based on different criteria, for instance based on

- phase of their creation: *developmental*, *operational*
- system boundaries: *internal*, *external*
- domain: *software*, *hardware*
- persistence: *permanent*, *transient*

Permanent faults are active in the given hardware unit until a corrective action is taken. They can be induced by some kind of physical defect in the hardware, such as shorts in a circuit, stuck bits in a memory or broken interconnections.

Transient faults remain active only for a short period of time. They are causing malfunction only in unspecified (usually random) time intervals and they disappear without any intervention. Often they are detected only through errors that results from their propagation.

An *error* is a deviation from correctness or accuracy in computation, which occurs as a result of a fault. Errors are usually associated with incorrect values in the system state. Errors can cause subsequent failures.

A *failure* is an event that occurs when the service delivered by the system deviates from the correct service. The delivery of incorrect service can manifest itself in performance of some function in subnormal quality or quantity. For example deterioration or instability of operation.

2.1.3 Means of dependability

Term *means of dependability* encompasses all methods and techniques which are utilizable for building a dependable system. According to [4] these methods are the following:

- *fault tolerance* - ensuring that the system works correctly also in presence of faults
- *fault prevention* - preventing of occurrence or introduction of faults
- *fault removal* - reduction of number or severity of faults
- *fault forecasting* - estimating the present/future number of faults and their consequences

2.2 Fault-tolerant systems

A fault-tolerant computing system according to [3] can be described as a system which has the built-in capability to preserve the continued correct execution of its programs and input/output (I/O) functions (without external assistance) in the presence of a certain set of operational faults. An operational fault is an unspecified (failure-induced) change in the value of one or more logic variables in the hardware of the system.

Faults occurring in a fault-tolerant system can be handled by exploiting and managing two basic types of redundancy: *space redundancy* and *time redundancy*. Space redundancy can be further classified into *hardware*, *software* and *information redundancy* based on the the type of resources added to the system. Software redundancy is applicable mainly for tolerating faults in the software-part of a system (e.g. by implementing the same program multiple times and comparing the results). Since this thesis is focused on hardware faults, software redundancy will not be further discussed. The description of hardware, information and time redundancy are based on [23] and [8].

2.2.1 Hardware redundancy

Hardware redundancy can be implemented by adding extra hardware components into the existing design in order to detect or override the effects of a failed component. Since it often goes together with a large area overhead, it is mainly used when other techniques (better components, quality control, etc.) have been exhausted or proved to be inefficient. Hardware redundancy have 3 basic subtypes: *active redundancy*, *passive redundancy* and *hybrid redundancy.*

Active redundancy is used in FTS where faults are rather temporary and occasional. In this type of redundancy fault tolerance is ensured by both detecting the fault and recovering from it in specified interval of time. There are various active redundancy techniques presented in [8]: *duplication with comparison* (duplex), *standby sparing*, *pair-and-a-spare*.

The duplex system (see 2.2) consists of two function units doing the same computation in parallel and a comparator which decides based on their outputs if a fault has occurred. If the outputs are identical, the comparator assumes that the results are error-free. After a successful fault-detection an error handling can take place. However, duplex system can guarantee the detection of only a single fault, two or more faults could cause a failure in the system.

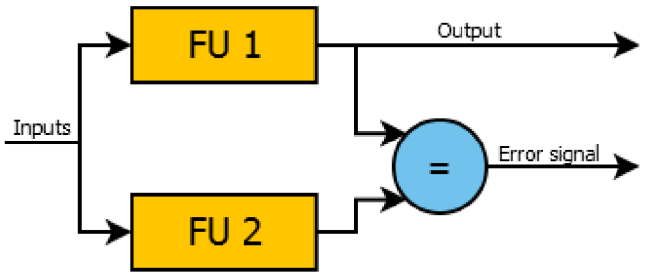


Figure 2.2: Duplication with comparison

Standby sparing is another scheme for active redundancy where only one of n function units (FU) is operational, the remaining $n-1$ units serve as spares (see figure 2.3). Spares are redundant components which are not needed for the regular functioning of the system. Faults are detected by the fault-detection units (FD). If a fault is detected the currently operating function unit is switched to one of the spares if at least one is still available. In case there are no more spares left, the system can still detect the faults, but cannot recover from them. A standby sparing system with n function units can tolerate $n-1$ function unit faults, the n-th fault is only detected.

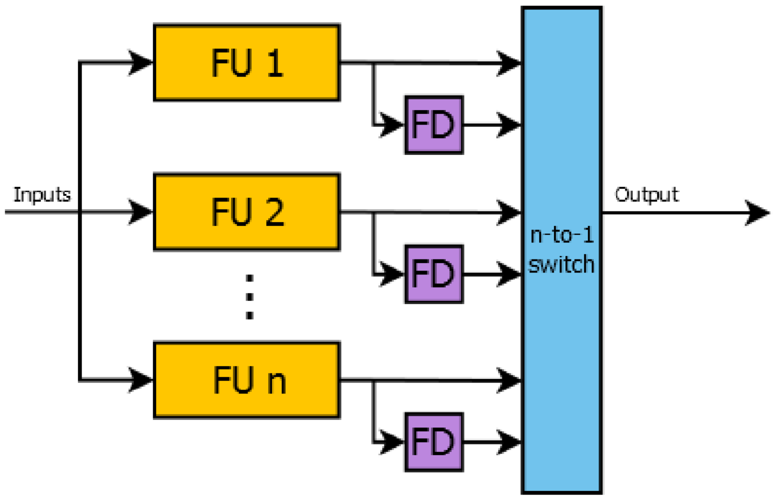


Figure 2.3: Standby sparing with n function units

Pair-and-a-spare technique is combined from standby sparing and duplication with comparison approaches (see figure 2.4). Contrary to the standby sparing mechanism, here two function units operate in parallel instead of one. The results of the two function units are compared to detect disagreement. In case of an error signal is detected, the two operating FUs' FD unit send their reports to the switch for analysing. The faulty FU is replaced with a spare FU, if any is still available. A pair-and-a-spare system can tolerate only $n-2$ FU faults, because after the $n-1$-st FU fault there are no more spare FUs which could be used as replacement. When the $n-1$-st fault occurs the system produces a correct output and it is reduced to a simplex system which is not able to detect any further faults. As a possible improvement, the system could be also reduced to a standby sparing model. It would mean that the $n-1$-th FU fault could be still tolerated and the n-th FU fault could be detected in the same way as in case of the original standby sparing technique.

Using the previously proposed improvement, this technique is still not better than the standby sparing technique, because both of them can tolerate the same number of faults. However, it could be possible that in some cases it is better to use the pair-and-a-spare technique. The main difference between the two techniques is that the pair-and-a-spare technique does not use its FD units each time when the output is checked for faults. Instead of that the results of two FUs are compared. If the given FU requires a more complex FD unit, it could be beneficial to use the pair-and-a-spare technique, which uses FD units only after the fault was detected by the FU-comparator. Hence, in certain cases pair-and-a-spare could operate faster than standby sparing.

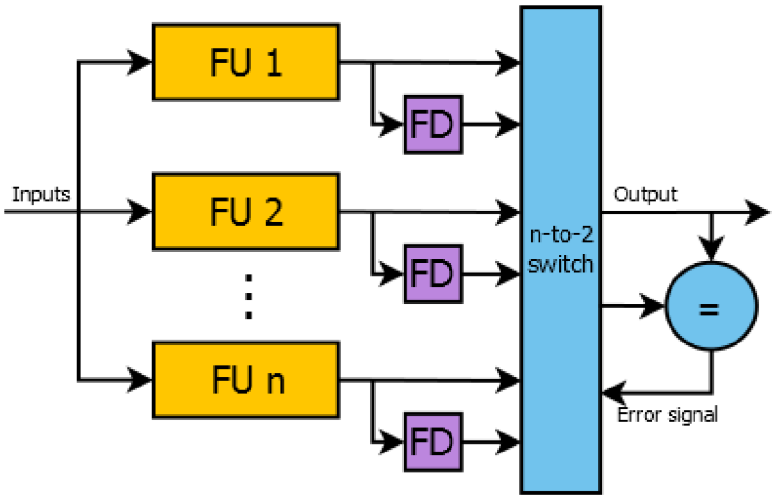


Figure 2.4: Pair-and-a-spare system with n function units

Passive redundancy, contrary to the active redundancy, does not warn the system about the possibly occurring faults, but masks them in order to insure that only correct values are passed to the system output. An example for this is a Triple Modular Redundancy system (TMR, see 2.5) which can be constructed by binding three hardware components, computing the same function for the same inputs. The outputs of the individual components are processed by a majority voting system which produces the final output of the system. If one of the three components fails, the other two components can take care about the correcting and masking of the fault.

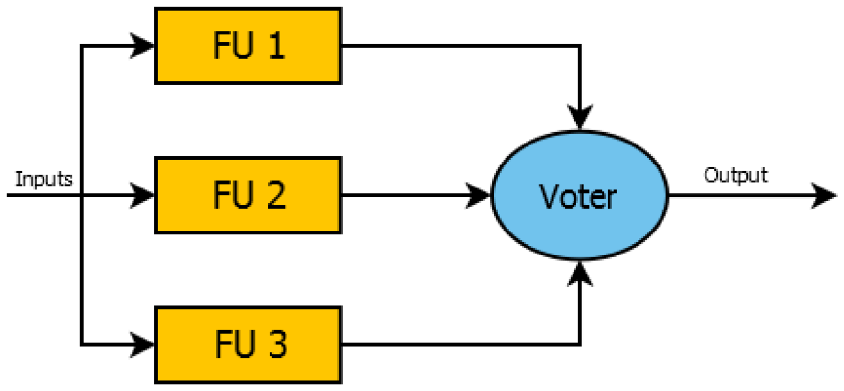


Figure 2.5: Triple-modular redundancy system

Hybrid redundancy is a combination of the previous two approaches (active and passive) by both masking the momentary erroneous results and detecting faults from which the system can subsequently recover. Based on [8] among the hybrid redundancy systems can be mentioned: *N-modular redundancy with spares*, *triplex-duplex redundancy* and *self-purging redundancy* which is illustrated on the below figure.

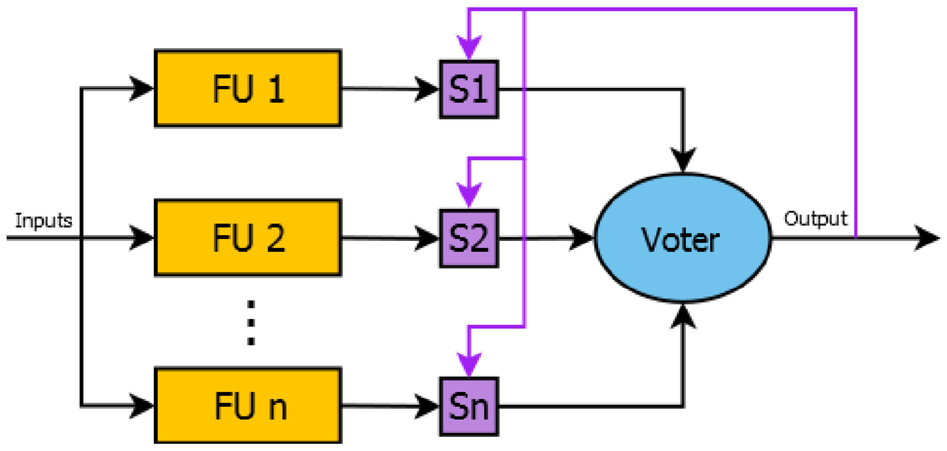


Figure 2.6: Self-purging redundancy system

Self-purging redundancy consists of n function units which can mask faults of $n - 2$ units in the following way: The outputs of the modules are compared by the voter, which works as a threshold gate, receiving outputs of the FUs with weights of 1 (active) or 0 (purged). If the voter detects any disagreement in the FUs' outputs, the system removes the faulty FU by forcing its weight to zero, hence it will not take part in any further voting. This removal process is also called as *purging*. After purging $n - 2$ units from the system, the remaining two units work as in case of the above presented duplex system (see 2.2). However, this system includes certain limitations regarding its observed function units. Firstly, the FUs cannot produce valid outputs with zero value because it would cause their undesired purging. Secondly, if a disagreement in the voter occurs, the FUs cannot pass new outputs to the voter till the faulty FU is purged.

2.2.2 Information redundancy

In case of information redundancy, usually extra bits are added to the original data bits, in order to detect or even correct the occurred errors. The most known forms of information redundancy are systems working with certain versions of error detecting codes [8]: parity codes (odd or even parity), M-of-n codes, Berger codes, CRC codes, the Hamming code, etc.

Parity code works with a single additional bit, called as *parity bit.* Considering a code with n bits, the parity bit, as the n-th bit of the code, contains information about the number of 1's in the preceding $n-1$ bits which is the *codeword.* We can distinguish *odd* or *even* parity codes based on what the parity bit signalizes, the even or odd number of 1's in the codeword. Parity code is able to detect single-bit errors, but their localization (and correction) is not possible.

2.2.3 Time redundancy

Time (execution) redundancy consists of repeating and acknowledging machine operations at various levels [3]:

- micro-operations
- single instructions
- program segments
- entire programs, etc.

While hardware and information redundancy techniques have impact on physical entities like weight, size, cost, power consumption, etc., time redundancy can be applied in cases when time is less important than extra hardware. The simplest way to attain time redundancy is the re-execution of the same program on the same hardware. If the same fault occurs after every re-execution of the program then it is a *permanent fault*, if it disappears we are talking about a *transient fault*, which is the primal utilization of time redundancy. In case of transient faults the faulty hardware unit is often still usable which can be handled using some error-recovering method instead of switching off the operation. Based on how many times the program is re-executed it can be decided how the occurring faults should be handled. If the program is executed only 2-times the fault can be only detected. If it is executed 3 or more times it can be also corrected.

2.3 Checking circuit implementations

The basic idea of implementing checking circuits in fault-tolerant systems is to create a new function unit which receives the same inputs as the supervised function unit (see 2.1), but is implemented differently and is able to recognize the faults occurring in the original unit. Checking circuits do not have to necessarily implement the same functionality as their tested function unit, their goal is limited to the validation of the supervised unit which is done based on the tested unit's inputs and outputs. On the contrary, using a replica of the original function unit should be avoided, because it could happen that both the tested and replica unit fail in the same manner and thus the fault detection might be unsuccessful.

An extended version of the TMR system is presented in [44] where each function unit (FU) has its own checker (see figure 2.7). In case of an error, both the voter and the adherent checker of the failing FU signalizes it. This mechanism unambiguously determines which FU produced the error and hence the erroneous FU can be easily removed from the system. The system remains fault-tolerant till there are at least two checker-FU pairs operating. Using a single checker and FU faults cannot be tolerated only detected.

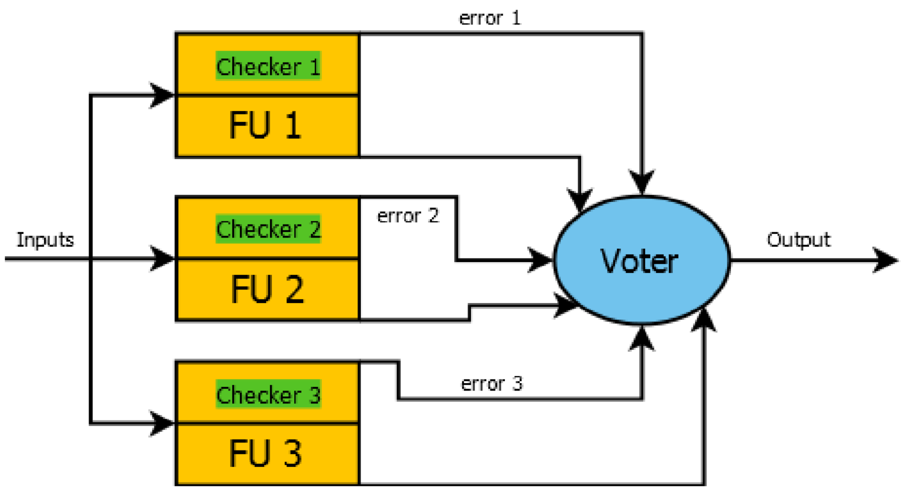


Figure 2.7: TMR system with online checkers

In paper [45] a methodology for generating checking circuits for specific digital circuits is introduced. It is based on a formal definition language describing the valid input-output signal combinations of the tested function unit and also its correct and erroneous states. The proper description of the tested unit is convertible to a finite state machine (FSM) from which the final checking circuit in VHDL/Verilog hardware description language can be generated. Finally, the checking circuit with the original function unit are synthesizable into FPGA. The advantage of this approach is that the generation of the checking circuit does not require the presence of an experienced designer, only the mentioned formal definition of the function unit needs to be described. In [46] is presented how the formal description is transformed to FSM and subsequently to VHDL. It also contains some real hardware designs (counter-decoder, register with multiplexer, duplex, etc.) in which the introduced checker generation is utilized.

Another approach for checking circuit generation is introduced in [28]. It uses active automata learning (combination of finite automata theory and machine learning techniques) for their generation which allows a more automatic construction than previously presented in [45]. The function units in this method are treated like black box, only some knowledge about the signals of the tested unit's interface is needed. This reduces the need for a hardware designer and therefore the probability of human mistakes.

The aim of this thesis is to use machine learning algorithms for checking circuit generation. Similarly to [28], function units are considered as black box where only the input and output signals of the tested units are significant. Training of machine learning algorithms is performed using big amount of data, which means thousands of input-output combinations gained from the given function unit. The individual machine learning algorithms used in this thesis are described in the next chapter (see 3).

Chapter 3

Machine learning

This chapter is an introduction to the machine learning techniques used in this thesis. It defines what machine learning is and presents its fundamental types and approaches.

According to T. M. Mitchell [31] machine learning can be defined as follows:

Definition 1. *„A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.“*

A set of data combined with possible learning signals is called *train data set*. This is received by the learner and after its processing (learning), the learner should be able to predict the output for previously unseen data. Since this data was not part of the training process, its use for the learner's evaluation should give an objective feedback about how accurately the learning system works. Hence, a data set with previously unseen data is called *test data set*.

We can distinguish various types of machine learning approaches based on the type and content of the train data set and steps required for its processing. The individual types are described in section 3.1.

Usually, application of machine learning to real world problems is realized in form of machine learning algorithms. These can have different varieties based on their utilized computational structure, presented in [33]:

- Functions
- Logic programs and rule sets
- Finite-state machines
- Grammars
- Problem solving systems

Machine learning algorithms can be also categorized by their capability to learn from sequential (time series) data. Section 3.2 describes algorithms which are capable to learn from time series data without any additional help. Subsequent section 3.3 contains description of machine algorithms (base algorithms) that require a wrapper algorithm to transform (reduce) the data into a non-sequential form first. On the reduced data one of the base algorithms can be applied.

3.1 Types of machine learning

This section is devoted to the description of the individual machine learning types. The machine learning approaches used in this thesis work with pre-generated sets of data, with randomly chosen subsets for training and testing. These type of learning methods can be categorized as *passive learners* because their input data is predefined, there is no further interaction between the learner and the underlying (learnt) system. The learner processes the previously generated train data set and produces the resulting machine learning model or classifier (see schema 3.1). Passive learning systems can be further divided based on the presence of a learning signal into two main categories: *supervised learning* and *unsupervised learning*. These are described in the below subsections 3.1.1 and 3.1.2.

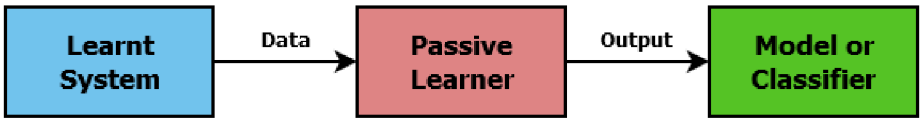


Figure 3.1: General schema for a passive learner

The second category of machine learning approaches encompasses a group of *active learners* which contrary to the passive ones, gather data by asking queries and receiving responses from the learnt system (see schema 3.2). The individual queries do not need to be set in advance. Instead, the answers to the previous queries can influence the next queries. The model or classifier can be considered as final when all the desired query-responses were processed. Active learning is out of the scope of this thesis, hence it will not be discussed in more detail.

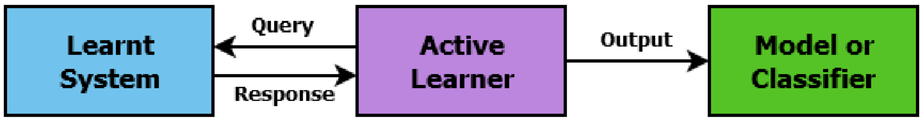


Figure 3.2: General schema for an active learner

3.1.1 Supervised learning

Supervised learning works with fully labelled train data sets where data is represented as a pair of $\{X, Y\}$ where Ys are actual labels of the data elements in X. The train data set is required to be labelled by a supervisor before the learning begins. The main goal of supervised learning is the prediction of labels Y_{new} for a new set of data X_{new} that are without labels. There are several approaches and algorithms that use supervised learning, for example [1]: neural network training using the backpropagation algorithm, decision tree learning, naive Bayes classifier, etc.

In supervised learning there are two possible types of algorithms that can analyse the training data and produce an inferred function. These are *classification* and *regression* [32].

Classification assigns each data element X a class Y from a group of discrete values. There are two main classification types depending on the number of classes to which data elements can be mapped. If there are only two classes, it is called *binary classification* ($X \rightarrow \{Y_1, Y_2\}$), otherwise it is *multi-class classification* ($X \rightarrow \{Y_1, Y_2, Y_3 ...\}$). The below figures (see 3.3) show the two classification types on data elements consisting of two attributes

X_1 and X_2. Classification has many utilizations in real-world applications, such as document (text) classification [30], email spam filtering [50], image classification [6], handwrite recognition [11] and face recognition [53], etc.

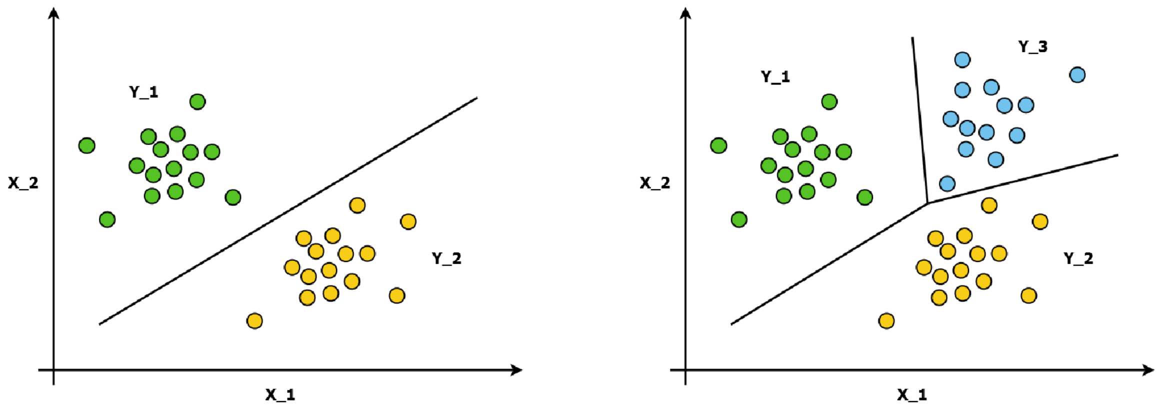


Figure 3.3: Binary classification (left) and multi-class classification (right)

Regression is similar to classification except here the response variable Y is continuous. It is about finding a function for the training data in order to learn Y as a function of X. Depending on the type of the found function we can distinguish different types of regressions. For example, *linear regression* is fitting the data with a linear function $Y = w_1X + w_0$ where w_1 and w_0 must be suitable. In cases where the linear model is too restrictive, *polynomial regression* can be applied, for example with a quadratic fitting function: $Y = w_2X^2 + w_1X + w_0$. The two regression types are illustrated on the below figures (see 3.4). There are several opportunities to use regression in the field of engineering, economics, medical researches and marketing.

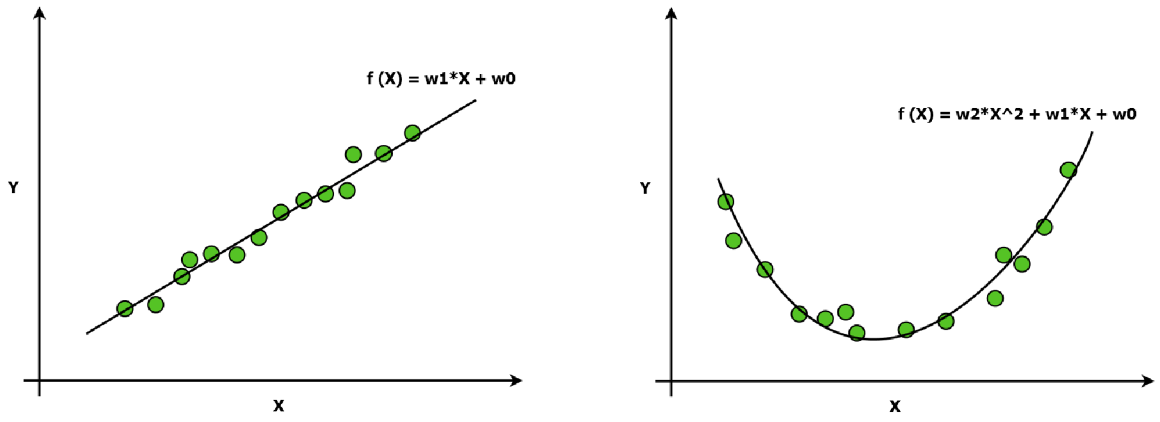


Figure 3.4: Linear regression (left) and polynomial regression (right)

3.1.2 Unsupervised learning

In this type of learning problems only data elements X are contained by the train data set, there are no labels Y. In cases when the presence of a supervisor is not possible, unsupervised learning is applied.

Unsupervised learning is closely related to the problem of *density estimation* in statistics (see in [1]). The aim of density estimation in unsupervised learning is to build a model

$p(X_i|\theta)$ which will estimate the distribution of the observed data. Since the distribution is created purely based on the observed data X_i, in case of unsupervised learning we are talking about unconditional density estimation. Contrary to this, supervised learning techniques can build models using conditional density estimation. Conditional density estimation can be formally described as $p(Y_i|X_i, \theta)$, where Y_i is the provided learning signal.

One of the possible methods for density estimation is *clustering* (described in [32]). It aims to find clusters or groupings of input based on the observed data elements. However, in unsupervised learning it is not clear to how many possible clusters can the data be distributed. Therefore, the first goal of clustering is to estimate the distribution over the number of clusters $p(K|\mathcal{D})$ where K denotes the number of clusters. The second goal is to estimate which cluster each data element belongs to. It can be formally described as $Z_i \in \{1, \dots, K\}$ where Z_i represents the cluster to which data element i is assigned. Since Z_i is not observable in the train data set, it can be also described as a hidden or latent variable. Computing the distribution of the individual data elements X_i to the clusters Z_i can be expressed as: $Z_i^* = \arg\max_k p(Z_i = k|X_i, \mathcal{D})$. On figure 3.5 is illustrated clustering in 2 dimensional space with 3 clusters. Clustering is applicable in many aspects of life: image compression, bioinformatics, astrophysical measurements, customized targeted advertising based on the users web-purchasing behaviour, etc. (see in [1] and [32]).

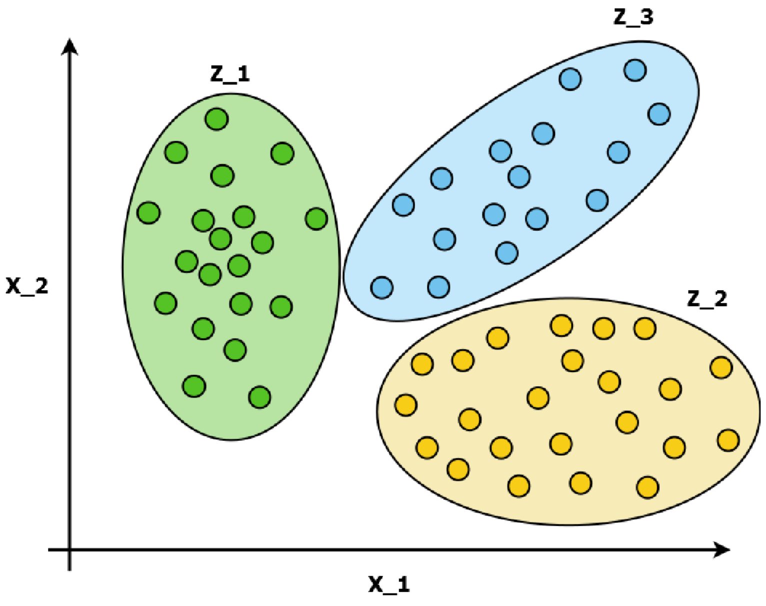


Figure 3.5: Clustering with $K = 3$

3.2 Machine learning algorithms for processing time series

A time series is a sequence of observations taken sequentially in time. The data points of the series are recorded at equally spaced time intervals. Time series data has an explicit order dependence between the individual observations, which can be considered as an extra *time dimension*. This time dimension provides additional information to the data stored in the elements of the series. Time series are utilized in several fields of science and technology: weather forecasting [47], marketing [34], signal processing [7], bioinformatics [19], etc. In this thesis it is used for storing subsequent inputs and outputs of hardware components with sequential logic.

This section is a description of machine learning algorithms which can be used for processing time series data. The difference between these and the standard learning algorithms (decision trees, feed-forward neural networks, etc.) is that the model of these algorithms does not grow proportionally with the length of the learnt data sequences. For example, if we would like to learn time series data using a decision tree algorithm like Random Forest, the number and the size of the individual trees of the forest would be needed to adjust depending on the length of the learnt series.

In this thesis the below presented Markov models (see 3.2.2) were used from this category of machine learning algorithms. These work based on a *stochastic process* (described in 3.2.1).

Except Markov models there are also other algorithms, having similar, native capability of processing time series data. Here can be mentioned recurrent neural networks [16], dynamic time warping techniques [18] and support vector machines [20], etc. Since these were not utilized in the experiments of this thesis, they will not be further described.

3.2.1 Stochastic process

Stochastic (random) process is a collection of random variables ordered by an index set. A formal definition can be read in [9]:

Definition 2. *Given a probability space* $(\Omega, \mathscr{F}, P)$*, a* ***stochastic process*** *is any collection of random variables* $\{X(t)\}_{t \in \mathcal{T}}$ *where* $\mathcal{T}$ *is the index set and* $X(t)$ *denotes the value of the stochastic process at time* t*.*

We can distinguish discrete-time and continuous-time stochastic processes, defined as:

Definition 3. *Given a stochastic process* $\{X(t)\}_{t \in \mathcal{T}}$ *with index set* $\mathcal{T} = \{0, 1, 2, \dots\}$ *or* $\mathcal{T} = \mathbb{N}_0$*,* $\{X(t)\}_{t \in \mathcal{T}}$ *is a* ***discrete-time stochastic process*** *which can be also denoted as* $\{X(n)\}_{n \in \mathbb{N}}$*.*

Definition 4. *Given a stochastic process* $\{X(t)\}_{t \in \mathcal{T}}$ *with index set* $\mathcal{T} = [0, \inf)$*,* $\{X(t)\}_{t \in \mathcal{T}}$ *is a* ***continuous-time stochastic process*** *which can be also denoted as* $\{X(t)\}_{t >= 0}$*.*

3.2.2 Markov models

Markov models (MM), named after Andrey Markov[1], are stochastic models, consisting of a set of states and a transition function. All Markov models have a common property, *Markov property*, which says that future states depend only on the current state, thus they are always independent from sequences of past states. It can be formally defined as:

Definition 5. *Given a stochastic process* $\{X(t)\}_{t \in \mathcal{T}}$*, its sequence of random variables has the* ***Markov property*** *if for any* $t \in \mathcal{T}$ *the future process* $(X(m), m > t, m \in \mathcal{T})$ *is independent of the past process* $(X(m), m < t, m \in \mathcal{T})$*, conditionally on* $X(t)$*.*

The types of Markov models can be distinguished based on two main properties:

- *controllability of the model's stochastic process*

[1]Russian mathematician, best known for his work on stochastic processes.

- *observability of the model's states*

Markov models with controllable state transitions belong to the category of *Markov decision processes* (MDP). MDPs are described in subsection 3.2.2.1.

This thesis is focused on MMs with autonomous stochastic processes. Based on the observability of these model's states can be distinguished two main types of MMs, *Markov chains* and *Hidden Markov models*. These are described in more detail in the below sub-sections 3.2.2.2 and 3.2.2.3.

3.2.2.1 Markov decision processes

Markov decision processes (MDP) are controlled stochastic processes satisfying the Markov property and assigning reward values to state transitions. According to [43], formally they can be described as a 5-tuple (S, A, T, p, r) where:

- S - is the state space in which the process' evolution takes place
- A - is the set of all possible actions which control the state dynamics
- T - is the set of time steps where decisions need to be made
- $p()$ - denotes the state transition probability function
- $r()$ - provides the reward function defined on state transitions

On figure 3.6 we can see a general MDP diagram. Actions $a_t \in A$ are made on state $s_t \in S$ in each time step $t \in T$, which affects the transition to next state $s_{t+1} \in S$. The reward obtained for the given transition in time step t is denoted as $r(s_t, a_t)$.

In case the states of the MDP are not fully observable we are talking about *Partially observable Markov decision processes*. Since MDPs are not used in this thesis, they are not described in more detail.

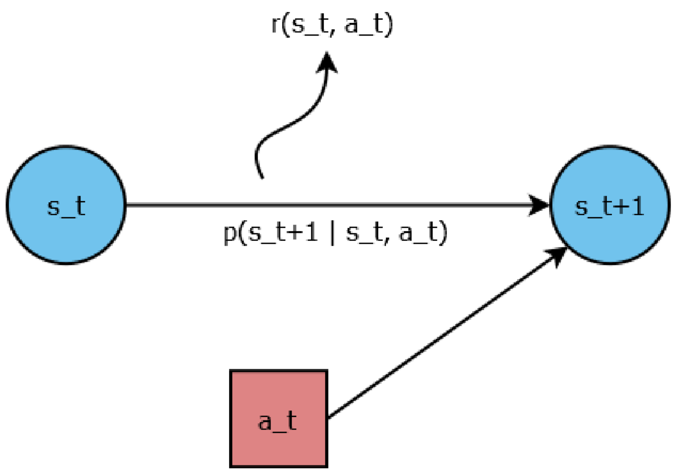


Figure 3.6: Influence diagram of a general Markov decision process

3.2.2.2 Markov chain

A Markov chain (MCH) is a stochastic process referring to a sequence of random variables which satisfy the Markov property, and where the states of the model are fully observable.

Similarly to stochastic processes, Markov chains can be also defined in discrete-time and continuous-time. The complete definition of discrete-time Markov chains can be formulated as follows [42]:

Definition 6. *A stochastic process $X = \{X(n), n \in \mathbb{N}\}$ over the state space S is a **discrete-time Markov chain** if*

- *for every $n \geq 0, X(n) \in S$*
- *for every $n \in \mathbb{N}$ and for all $i_n, i_{n-1}, \ldots, i_0 \in S$ we have*

$$\Pr\{X(n) = i_n \mid X(n-1) = i_{n-1}, \ldots, X(0) = i_0\} = \Pr\{X(n) = i_n \mid X(n-1) = i_{n-1}\},$$

*when both conditional probabilities are defined. The probabilities $p_{ij} = \Pr\{X(n) = i_n \mid X(n-1) = i_{n-1}\}$ are called **transition probabilities**.*

Definition 7. *A discrete-time Markov chain with a finite number of states is called **finite-state discrete-time Markov chain**. The transition probabilities p_{ij} of a finite-state discrete-time Markov chain are represented in a **transition matrix** P with entries p_{ij}.*

This thesis deals with finite-state discrete-time Markov chains only, hence they are further described without explicit mention. Markov chains can be represented using a state-transition diagram (see figure 3.7) which describes the transition probabilities between the individual states of the chain. The below example shows a weather prediction problem, where the weather of tomorrow (next state) is determined based on the today's weather (current state). The individual states of an MCH can be *transient* or *absorbing*. These can be defined as follows:

Definition 8. *A state $s_i \in S$ of a Markov chain is called **absorbing** if the transitions to other than the current state are impossible: $p_{ii}(t, t+1) = 1$ for all $t \geq 0$.*

Definition 9. *A state $s_i \in S$ of a Markov chain is called **transient** if there is at least one transition to other than the current state: $p_{ii}(t, t+1) < 1$ for any $t \geq 0$.*

A Markov chain is absorbing if it has at least one absorbing state, and if from every state it is possible to go to an absorbing state (not necessarily in one step). Therefore, the below Markov chain is not absorbing because it has only transient states.

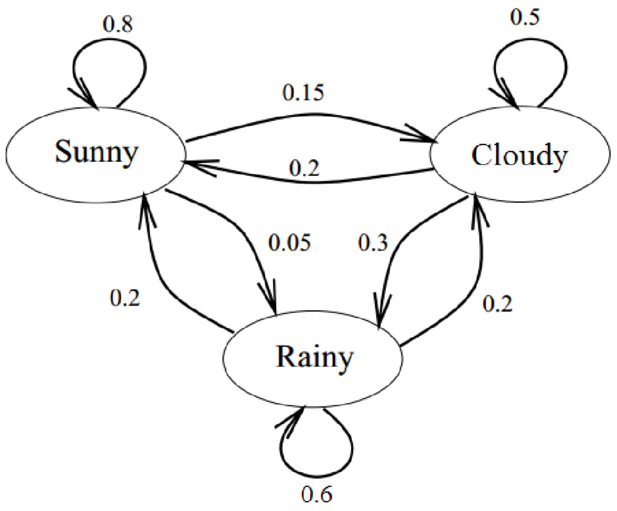


Figure 3.7: State-transition diagram of a Markov chain used for weather prediction [49]

3.2.2.3 Hidden Markov model

Hidden Markov model (HMM) is a Markov chain where the states are only partially observable. The description of the elements, types and problems of HMMs were described based on the book of L. R. Rabiner and B. H. Juang [39].
HMM has 3 primal driving elements which determine its operation:

- *states* - it has finite, N number of states
- *transition probability distribution* - based on it the model decides which state to enter at clock time t
- *observation (output) probability distribution* - represents a random variable or stochastic process, which is held fixed for each state individually and is responsible for producing observation variables

The following figure (see 3.8) illustrates a general HMM architecture, where the observed outputs y_t have an unobserved state x_t at time t.

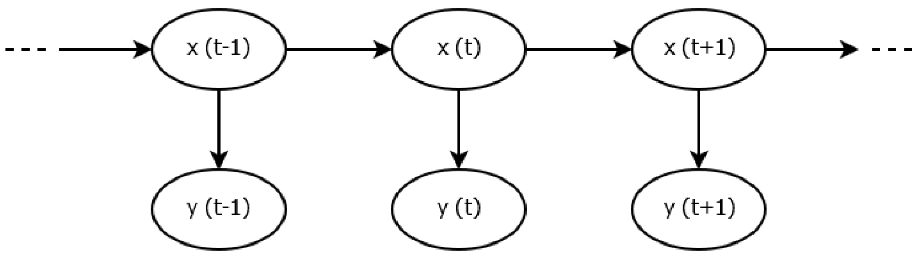


Figure 3.8: General HMM architecture

A classic example for an HMM is the *urn and ball* model (see 3.9) which shows the functionality of HMMs on N urns containing balls with M possible colors. In this case, the observation variables are the colors of the balls, which are produced based on the next steps:

1. Select a starting urn based on an initial state distribution.
2. Pull out a ball from the chosen urn according to the output probability distribution of the chosen urn.
3. Record the color of the drawn ball and return it to the urn.
4. If the sequence of the recorded colors reached the desired length T, terminate the process.
5. Select a new urn based on the state transition probability distribution of the current urn. Continue with step 2.

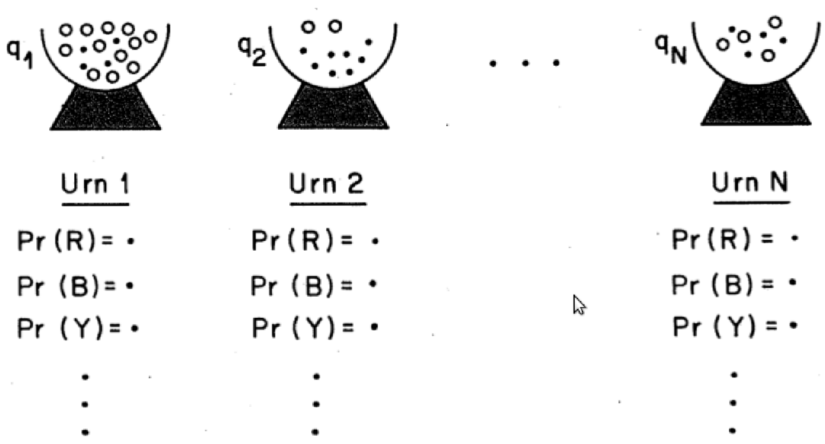


Figure 3.9: Urn and ball model [39]

The type of an HMM is determinable based on different properties of the model. Regarding the HMM's state connections, the model can have three basic configurations, linear (see 3.10), left-to-right (or Bakis, see 3.11) or ergodic (see 3.12).

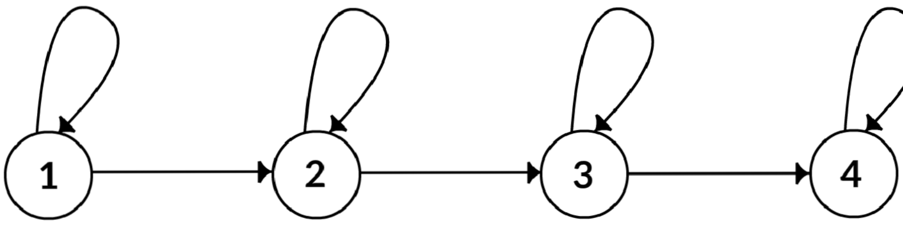


Figure 3.10: Linear HMM configuration

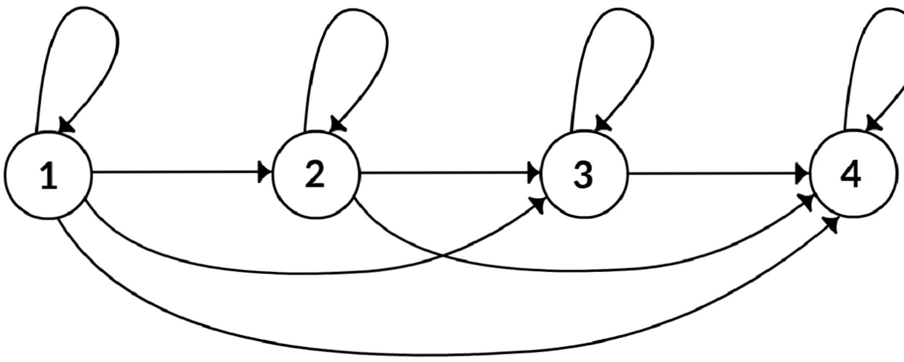


Figure 3.11: Left-to-right (Bakis) HMM configuration

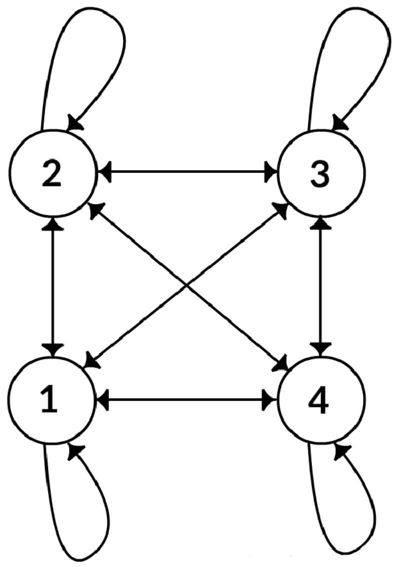


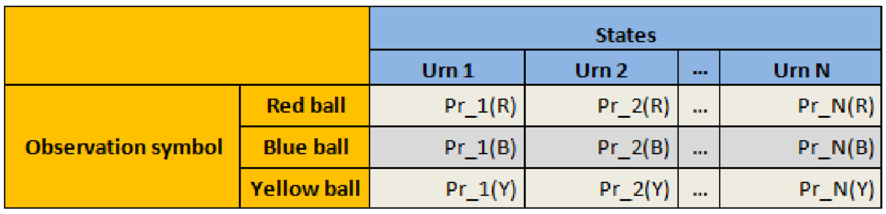
Figure 3.12: Ergodic HMM configuration

Based on the type of the produced observation variables we can distinguish *discrete* and *continuous* HMMs. These are explained in the next two paragraphs.

Discrete HMM produces symbols from a categorical distribution and its model consists of the following components:

1. T = length of the observation sequence (total number of clock times)
2. $Q = \{q_1, q_2, \dots q_N\}$, states
3. $V = \{v_1, v_2, \dots v_M\}$, observation symbols
4. $A = \{a_{ij}\}, a_{ij} = \Pr(q_j \text{ at } t+1 \mid q_i \text{ at } t)$, state transition probability distribution
5. $B = \{b_j(k)\}, b_j(k) = \Pr(v_k \text{ at } t \mid q_j \text{ at } t)$, observation symbol (output) probability distribution in state j
6. $\pi = \{\pi_i\}, \pi_i = \Pr(q_i \text{ at } t = 1)$, initial state distribution

An example for a discrete HMM is the above presented urn and ball model (see 3.9). In this case the observation probability distribution between the individual states (urns) can be represented as a table:



		States			
		Urn 1	Urn 2	...	Urn N
Observation symbol	Red ball	Pr_1(R)	Pr_2(R)	...	Pr_N(R)
	Blue ball	Pr_1(B)	Pr_2(B)	...	Pr_N(B)
	Yellow ball	Pr_1(Y)	Pr_2(Y)	...	Pr_N(Y)

Figure 3.13: Table representation of the observation probability distribution

Continuous HMM outputs observation variables using output probabilities based on a continuous density function. Therefore the output probability distribution $b_j(k)$ of the discrete HMM is replaced here by a continuous probability density function $b_j(x)dx, 1 \leq j \leq N$ where x is a continuous vector. This function gives a probability that the observation vector lies between x and $x + dx$ [39]. There are different forms of the $b_j(x)$ function from which one of the most common is its representation as a mixture of Gaussians which was also used in this thesis [39]:

$$b_j(O) = \sum_{m=1}^{M} c_{jm}\mathcal{N}[O, \mu_{jm}, U_{jm}], \qquad 1 \leq j \leq N \tag{3.1}$$

where O is the vector being modelled, c_{jm} is the mixture coefficient for the m^{th} mixture in state j and $\mathcal{N}$ is the normal density with mean vector μ_{jm} and covariance matrix U_{jm} associated with state j and mixture m.

Besides the utilization of continuous HMMs in this thesis, it also has several real-world applications, among others in speech processing and recognition [38].

HMM in machine learning

The use of HMM in machine learning algorithms is closely related to the solution of the three commonly known HMM problems. The description of these problems and their solutions are presented based on [39]:

1. **Scoring:** Computing the probability of a given model's observation sequence. Formally specified as $\Pr(O|\lambda)$ where $O = O_1, O_2, \dots, O_T$ is the observation sequence and $\lambda = (A, B, \pi)$ is the HMM model.
2. **Matching/Decoding:** Choosing a sequence of states $Q = q_1, q_2, \dots, q_T$, optimal for a given observation sequence $O = O_1, O_2, \dots, O_T$ in model $\lambda = (A, B, \pi)$**.**
3. **Training:** Adjusting the parameters of a given model $\lambda = (A, B, \pi)$ to maximize the probability of a given observation sequence $\Pr(O \mid \lambda)$.

1. Scoring problem

The scoring problem usually appears when there are more HMM models and we need to decide by which model a given observation sequence was generated. In case of classification, the individual HMM models each represent a class. Hence, by calculating which HMM model produced the given observation sequence, it is determined to which class the given sequence is assigned.

The most straightforward solution to this problem is calculating the probability of the given observation sequence $O = O_1, O_2, \dots, O_T$ for each possible state sequence $Q = q_1, q_2, \dots, q_T$, formally described as:

$$\Pr(O \mid Q, \lambda) = b_{q_1}(O_1)b_{q_2}(O_2)\dots b_{q_T}(O_T)$$

The probability of state sequence Q can be expressed as:

$$\Pr(Q \mid \lambda) = \pi a_{q_1q_2}a_{q_2q_3}\dots a_{q_{T-1}q_T}$$

The product of the above terms is a joint probability of O and Q, describing that they must occur simultaneously:

$$\Pr(O, Q \mid \lambda) = \Pr(O \mid Q, \lambda) \Pr(Q \mid \lambda)$$

The scoring problem can be solved by summing the above joint probability over all possible state sequences:

$$\Pr(O \mid \lambda) = \sum_{\text{all } Q} \Pr(O \mid Q, \lambda) \Pr(Q \mid \lambda) = \sum_{q_1, q_2, \ldots, q_T} \pi_{q_1} b_{q_1}(O_1) a_{q_1 q_2} b_{q_2}(O_2) \ldots a_{q_{T-1} q_T} b_{q_T}(O_T)$$

However, this calculation is computationally unfeasible, because considering N states to go through, it requires $2TN^T$ calculations ($N^T(2T-1)$ multiplications and $N^T - 1$ additions). Therefore, a more efficient algorithm should be used, commonly known as the *forward algorithm (procedure).*

Forward algorithm computes the probability of a given observation sequence O on a given HMM model λ by introducing the *forward variable* $\alpha_t(i)$ defined as:

$$\alpha_t(i) = \Pr(O_1, O_2, \ldots, O_t, q_t = S_i \mid \lambda)$$

The forward variable of a given model λ expresses the probability of the partial observation sequence (until time t) and state S_i at time t. It can be solved inductively (N denotes the number of possible states in the model):

1. $\alpha_1(i) = \pi_i b_i(O_1)$, for $1 \leq i \leq N$
2. $\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right] b_j(O_{t+1})$, for $t = 1, 2, \ldots, T-1$; $1 \leq j \leq N$
3. The probability of the given observation sequence O in model λ is then:

$$\Pr(O \mid \lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

This algorithm requires only N^2T calculations ($N(N+1)(T-1) + N$ multiplications and $N(N-1)(T-1)$ additions) [39] which is a lot more efficient than the direct calculation.

Backward algorithm can be used for computing the probability of the partial observation sequence $O_{t+1}, O_{t+2}, \ldots, O_T$ when state S_i at time t and model λ are given. This probability can be expressed using a *backward variable*:

$$\beta_t(i) = \Pr(O_{t+1}, O_{t+2}, \ldots, O_T \mid q_t = S_i, \lambda)$$

Similarly to the forward algorithm, $\beta_t(i)$ can be also solved inductively:

1. $\beta_T(i) = 1$, for $1 \leq i \leq N$
2. $\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1})$, for $t = T-1, T-2, \ldots, 1$; $1 \leq i \leq N$

This algorithm is not necessary for solving the Scoring problem, but it is related to the forward algorithm and it can be used for other HMM problems (e.g. training). The subsequent computation of forward and backward variables is commonly called as *forward-backward algorithm (procedure).*

2. Matching/Decoding problem

Finding the optimal state sequence Q for a given observation sequence O in a model λ can be achieved in different ways. One way is to calculate the individually most likely states for each time step using the forward-backward algorithm. The probability of being in state S_i at time t, given the observation sequence O and model λ can be expressed as:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\Pr(O \mid \lambda)}$$

Using $\gamma_t(i)$, the individually most likely state q_t at time t is:

$$q_t^* = \arg\max_{1 \le i \le N} \Big[\gamma_t(i)\Big] \text{ for } 1 \le t \le T$$

However, this method does not consider any disallowed transitions between the model's states. Therefore, if $a_{ij} = 0$ for some i and j, this procedure can lead to an impossible state sequence Q. In order to avoid such incorrect state sequences, techniques for finding the *single best state sequence* should be used. This thesis utilized the *Viterbi algorithm* for this purpose.

Viterbi algorithm can find the most likely sequence of hidden states, also called as *Viterbi path.* We define $\delta_t(i)$ as the probability of the highest probability path at time t [1]:

$$\delta_t(i) = \max_{q_1, q_2, \dots q_{t-1}} p(q_1, q_2, \dots q_{t-1}, q_t = S_i; O_1, O_2, \dots O_t \mid \lambda)$$

After recursive calculation of $\delta_{t+1}(i)$, the Viterbi path can be read by backtracking from T. The backtracking process requires saving the *best previous state*, denoted as $\psi_t(i)$, which maximizes $\delta_t(i)$ at time $t-1$. Viterbi path can be computed using the following 4 steps [1]:

1. Initialization:
$$\delta_1(i) = \pi_i b_i(O_1) \text{ for } 1 \le i \le N$$
$$\psi_1(i) = 0$$

2. Recursion, for $2 \le t \le T$ and $1 \le j \le N$:
$$\delta_t(j) = \max_{1 \le i \le N} \Big[\delta_{t-1}(i) a_{ij}\Big] b_j(O_t)$$
$$\psi_t(j) = \arg\max_{1 \le i \le N} \Big[\delta_{t-1}(i) a_{ij}\Big]$$

3. Termination:
$$p^* = \max_{1 \le i \le N} \Big[\delta_T(i)\Big]$$
$$q_T^* = \arg\max_{1 \le i \le N} \Big[\delta_T(i)\Big]$$

4. Path (state sequence) backtracking, for $t = T-1, T-2, \dots, 1$:
$$q_T^* = \psi_{t+1}(q_{t+1}^*)$$

3. Training problem

Training an HMM model is about adjusting the model's parameters (A, B, π) to maximize the probability of the observation sequence O in the given model. This maximum likelihood problem is usually solved iteratively. In this thesis the *Baum-Welch algorithm* was used, described based on [39].

Baum-Welch algorithm uses the *Expectation-maximization (EM) algorithm* for the iterative re-estimation of the model's parameters. The EM algorithm consists of two steps: Expectation (E) and Maximization (M), which are described in the below paragraphs.

The calculation of the **E step** requires the definition of the probability of a path being in state S_i at time t and making a transition to state S_j at time $t+1$, given the observation sequence O and model λ:

$$\xi_t(i,j) = \Pr(q_t = S_i, q_{t+1} = S_j \mid O, \lambda)$$

This probability can be further expressed by utilizing the forward-backward algorithm, calculating the α, β variables:

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\Pr(O \mid \lambda)}$$

The probability of being in state q_i at time t, previously defined as $\gamma_t(i)$, given the observation sequence O and model λ can be defined as:

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j)$$

Using $\gamma_t(i)$ and $\xi_t(i,j)$ can be computed the two expectations of the E step, needed by next step M:

- Expected number of transitions from state S_i:

$$\sum_{t=1}^{T-1} \gamma_t(i)$$

- Expected number of transitions from state S_i to S_j:

$$\sum_{t=1}^{T-1} \xi_t(i,j)$$

The **M step** is about re-estimating the model's parameters based on the previously calculated expectations. This step is repeated multiple times to maximize the probability of sequence O, being observed from the model λ, until some limiting point is reached. The M step's re-estimation formulas for π, A and B are the following:

1. $\overline{\pi_i} = \gamma_1(i)$ for $1 \leq i \leq N$
2. $\overline{a_{ij}} = \dfrac{\sum\limits_{t=1}^{T-1} \xi_t(i,j)}{\sum\limits_{t=1}^{T-1} \gamma_t(i)}$

3. $$\overline{b_j(k)} = \frac{\sum\limits_{t=1;\ O_t=k}^{T} \gamma_t(j)}{\sum\limits_{t=1}^{T} \gamma_t(j)}$$

After re-estimating the model's parameters $\lambda(\pi, A, B)$ we gain a new model $\overline{\lambda}(\overline{\pi}, \overline{A}, \overline{B})$ where $\overline{\pi} = \{\overline{\pi_i}\}$, $\overline{A} = \{\overline{a_{ij}}\}$ and $\overline{B} = \{\overline{b_j(k)}\}$. It can be stated that the new model $\overline{\lambda}$ can produce the same observation sequence O with greater or equal probability than the former model λ: $\Pr(O \mid \overline{\lambda}) \geq \Pr(O \mid \lambda)$. Therefore, after each $\overline{\lambda}$ calculation, λ is replaced by $\overline{\lambda}$. This iterative approach leads to the convergence of the model's parameters to their optimal values to produce the given observation sequence O.

3.3 Machine learning algorithms applicable on transformed time series

While the algorithms described in the previous chapter (see 3.2) were designed to be applicable on time series data directly, there is also another type of machine learning algorithms that does not have this capability. The algorithms described in this section are usually far less effective to use on time series data directly, because they would require much more resources and computation to handle the higher number of data elements, contained by time series data. For instance a feed-forward neural network would require more neurons or hidden layers. In case of decision tree algorithms it could mean an increase of the number of the used trees. Another problem is that the length of time series data is not always predefined. It makes hard to configure the parameters of these algorithms (how many neurons/trees are needed). Therefore, instead of applying these algorithms directly, the time series data goes through some reduction or redistribution of its data elements first. Finally, the originally given algorithm (base algorithm) is applied for the transformed data. In this thesis feed-forward neural networks (see 3.3.1) and decision tree algorithms (see 3.3.2) were used as base classifiers.

The transformation of time series data can be performed in different ways, but mainly depends on the 3 following aspects:

- capabilities and properties of the chosen base algorithm
- difference between the data elements of the time series: if there is no significant difference between the individual data elements of the series and all the elements have numeric type, it might be possible to reduce the series using some averaging-method.
- coherence between the subsequent data elements: if the subsequent data elements of the series do not lose their meaning if they are separated, it could be possible to redistribute the individual elements of the series to separate instances which could be used for training and testing the base algorithm.

The used transformation method implementations are described in section 4.3.4.

3.3.1 Artificial neural networks

Artificial neural networks (ANNs) are models constructed based on the knowledge gained in the field of different biological sciences in order to attain such a mechanism that will be able to solve problems like the human brain. The human brain has around 86 to 120 billion neurons (see [15]), which makes it very hard to imitate. ANNs also consist of neurons but their behaviour is far more simple than ours.

The neurons in ANNs (also known as units or cells) have two basic tasks: receive inputs from their neighbours and after computing their adherent outputs, propagate these to other neurons. They have weighted connections between each other which influence the propagation of the outputs. Based on the position of these neurons within the neural system, three types of neurons can be distinguished: input neurons, output neurons and hidden neurons. After computing the output signals it is also needed to adjust the weights of the connections.

Based on the patterns of connections between the neurons, two types of network topologies can be distinguished [24]:

- **Feed-forward networks** are such types of networks where data can flow through several layers from input to output neurons without feedback connections. A classic example of feed-forward ANNs is the Perceptron whose multilayer variant is characterized in the next subsection.

- **Recurrent networks** (RNNs) contain also feedback connections. In contrast to the feed-forward ANNs, RNNs can process time series data directly, hence they belong to the previous category of machine learning algorithms, described in section 3.2. Recurrent neural networks are one of the best approach for language modelling [29]. Since no recurrent network was used for the purpose of this thesis, the ANNs in the ensuing parts of this paper should be considered as feed-forward networks.

Multilayer Perceptron

Multilayer Perceptron (MLP) is a feed-forward artificial neural network model that maps sets of input data onto a set of appropriate output. MLP is a modification of the standard linear perceptron with the difference that MLP can distinguish non-linearly separable data. An MLP consists of three or more layers (an input layer, an output layer and one or more hidden layers). We can imagine it as a finite directed acyclic graph where the nodes are the neurons with a non-linear activation function (see Figure 3.14). Each neuron has a weighted input which is mapped by the given activation function to the neuron's output. There are different activation function types, for example it can be a *logistic sigmoid function* $f(z) = 1/(1+e^{-z})$ with an output range from 0 to 1 or a *hyperbolic tangent function* $f(z) = tanh(z)$ with a range from -1 to 1 [40]. The neurons of i-th layer serve as input features for neurons of $i+1$-th layer. The nodes in the input layer are called *input neurons*. These are not a target for any connection. An MLP with n dimensions must have n input neurons (one for each dimension). The nodes in the output layer are called *output neurons*. These are not a source of any connection and their number depends on the way how target values of the training patterns are described. The nodes that are neither input neurons nor output neurons are called *hidden neurons*. Hidden neurons belong to the hidden layer. The connections that hop over several layers are called *shortcuts*. The weight between neuron i and j is denoted as $w_{i,j}$, which is a real number ($w_{i,j} \in \mathbb{R}$). The activation (output) value for neuron i is denoted as O_i. The input received by a neuron j in the hidden layer is given by the formula:

$$Net_j = \sum_{i=1}^{M} w_{i,j} O_i$$

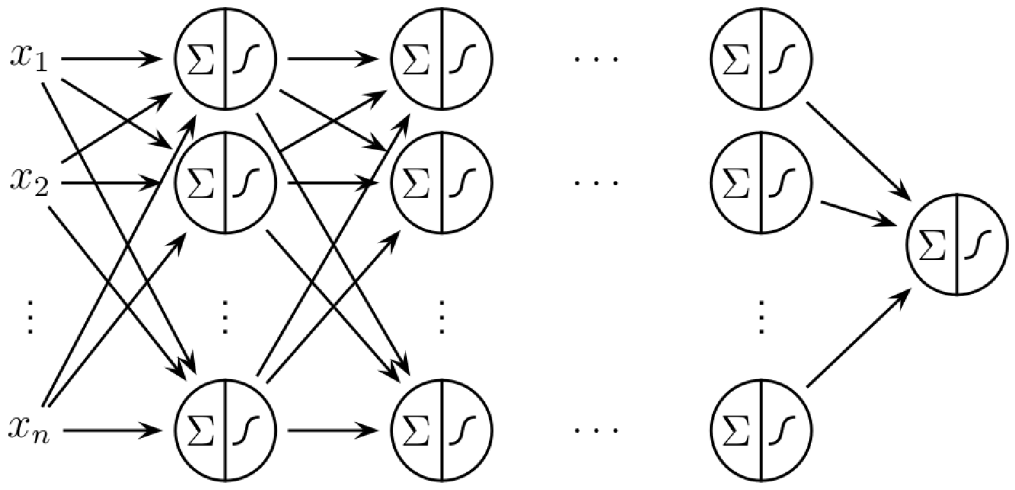


Figure 3.14: Multilayer Perceptron [40]

where M is the number of neurons feeding into neuron j. Activation value O_i can be expressed using the activation function f:

$$O_i = f(Net_i)$$

Backpropagation is a supervised learning technique which can be used for training the MLP. This algorithm consists of two main processing steps [48]. The first step includes forward-propagation of inputs, generation of output activations and subsequent backward-propagation of errors. The algorithm calculates delta-differences from the actual and desired output values of the individual neurons. The desired output values are determined by the train data set. The difference between the actual output value O_k at neuron k and the expected output value t_k is denoted as:

$$\Delta_k = t_k - O_k$$

From the computed Δ_k differences are computed the *error signals* (δ_k) for the individual output neurons:

$$\delta_k = \Delta_k f'(Net_k)$$

where f' is the derivative of the activation function f. When the error signals are computed for each neuron, the second step of the algorithm begins: weight-update of the connections in the network. Updating weight $w_{i,j}$ between neuron i and j can be performed based on the following formulas:

$$\Delta w_{i,j} = l_r \delta_j O_i$$

$$w_{i,j} = w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j}$ is the change in the weight between neurons i and j. l_r is the *learning rate*, indicating the relative change of the calculated weight. Usually it is a small constant (0 to 1) which influences the speed and quality of the learning. If l_r is too low, the MLP will learn slowly but usually more accurately. However, if l_r is too low it can also induce that the network will not be trained properly. On the contrary, if it is set too high, the learning is faster but the optimal weight is not always reached, because it can be overshot. Propagation and weight updating is repeated until the performance of the network is satisfactory, meaning that the error on the output neurons becomes minimal. To minimize the error on all output neurons over all training patterns presented to the network, the following *error function* is defined:

$$E = \frac{1}{2} \sum_p \left[\sum_k (t_{pk} - O_{pk})^2 \right]$$

where p is the subscript for the training pattern and k is the subscript for the output neurons. Accordingly to this, t_{pk} is the desired value of output neuron k for pattern p and O_{pk} is the actual output value of neuron k for pattern p.

3.3.2 Decision trees

A decision tree (DT) is a hierarchical data structure that can be applicable in supervised learning. It can be utilized for both classification and regression. The DT algorithms belong to the group of non-parametric methods where the input space is divided into local regions based on a distance measure (for example Euclidean norm). For each region a local model is computed [1].

Like any other tree structures, DTs also consist of nodes, which are called *decision nodes*. A decision node which is a leaf node in the DT, is commonly called as *terminal node*. By each decision node a discrete function is implemented, based on which the node's output is chosen from one of its branches. This process is repeated recursively from the root node until one of the tree's leaf node will be found. The leaf nodes' input value is equal to the output of the tree which can have two possible representations: a class code in case of classification or a numeric value if the tree was used for regression.

Decision trees can have many different implementations. One of the possible approaches is to perform a top-down, greedy search through the space of possible nodes of the tree [31]. This approach is utilized by the Iterative Dichotomiser 3 (ID3) algorithm [36] and also by its successor, the C4.5 algorithm [37]. Since these methods are not used in this thesis, they will not be described in more detail.

This thesis utilized another type of tree algorithms, called as Classification And Regression Trees (CART). Specifically it focused on the use of the *Random Forest algorithm*, described in the next subsection 4.3.2.

Random Forest

Random Forest (RF) is an ensemble learning method which was developed by Leo Breiman [5]. The idea of ensemble learning is to build a predictive model by integrating multiple models [41]. In case of RF these models are unpruned trees that can be used for both classification and regression. The individual trees are gained by selecting random samples from the training data. The main purpose of Random Forest is to make separate predictions firstly by each individual decision tree member and then aggregate these predictions. The aggregation process is distinct based on whether it is a classification (majority vote) or regression (averaging). All these properties are conducive to the RF classifier's accuracy and its very fast classification. On figure 3.15 we can see a schematic construction of an RF.

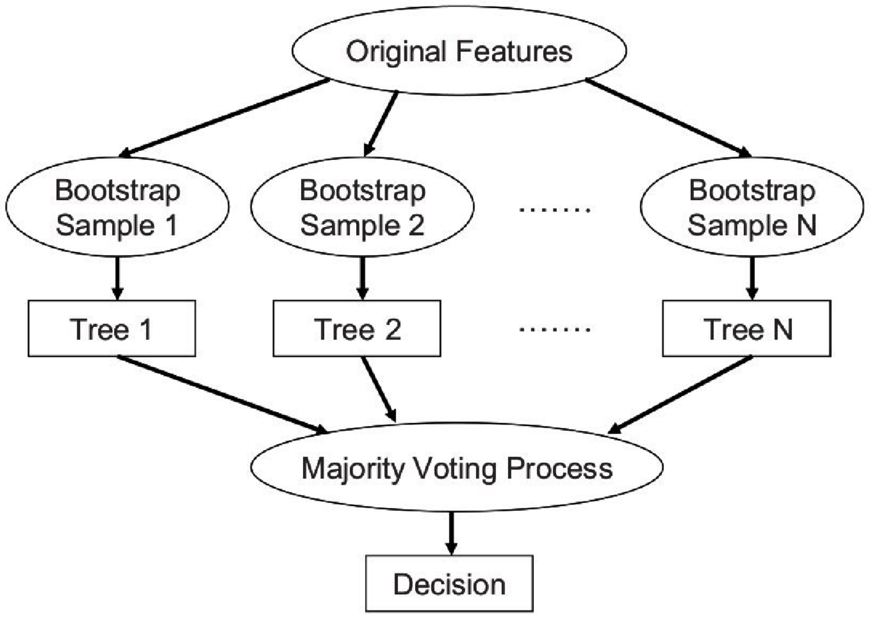


Figure 3.15: Random Forest construction schema [14]

For growing trees, RF uses the CART (classification and regression trees) methodology and grows its trees to maximum size without pruning. According to [14], growing trees with the CART methodology has 2 basic aims: predict continuous dependent variables (regression) and predict categorical predictor variables (classification). CART can be divided into 4 basic steps which are the following:

1. Tree building: This is performed by splitting of nodes recursively. The terminal nodes are assigned to classes depending on the classes probability distribution of the dependent variable at terminal node.

2. Stopping tree building: It is about the stopping of the tree's growing process.

3-4. Tree pruning and optimal tree selection: These final two steps are applied in the general CART methodology only. Random Forest is a special case where trees remain unpruned, hence this step is omitted.

The feature with the help of which RF can yield an improved classification accuracy is the ensemble of trees which are voting individually for the most promising class during the classification process. The RF algorithm's tree building process has certain differences compared to the simple CART methodology. The reason why RF can be more successful than simple decision tree algorithms lies in the two randomized procedures applied during its tree building. Let's define the following parameters for latter explanations:

- N is the number of all data set instances in the training data set.
- n is a sample gained from all data set instances N, also called as bootstrap sample. This is the number of instances which will be randomly chosen from the train data set (with replacement) to create a subset.
- M is the number of attributes in the data set instances.

- m is the number of randomly selected attributes from M, which should be used for the decision making at the individual nodes of the tree.

After choosing parameters n and m, the algorithm calculates the best split on the bootstrap sample. Contrary to the general CART method, RF calculates the split based on the reduced number of instances n and attributes m. The bootstrap sample is further divided into two parts: The first part, approximately the two-thirds of the sample, is used for training the classifier in the given ensemble. The second part, commonly named as *out of bag data*, is used for estimating an unbiased test error for each tree. The averaged test error of the individual trees is called *out of bag error*. The out of bag error can be used for internal evaluation of the classifier's accuracy, no extra validation data set is necessary. The random selection of the mentioned n instances and m attributes is repeated before each new tree-building process.

Chapter 4

Application of algorithms for checking circuit generation

The previous chapter has described various machine learning algorithms suitable for processing time series. Since the aim of this thesis is to apply these algorithms for automatic construction of checking circuits for hardware components with sequential logic, this chapter deals with the specific implementation of these algorithms and with other processes that their use requires. The used machine learning algorithm implementations are described in section 4.3.

The primal role of the machine learning algorithms in the generated checking circuits is to verify whether the chosen hardware component works correctly or not. The operation of this verification process is illustrated on the below figure (4.1). The hardware component, implemented in a hardware description language (HDL), computes the outputs for its received inputs. The input and output values are both passed to a machine learning classifier which is able to decide whether the component works correctly or a fault has been occurred.

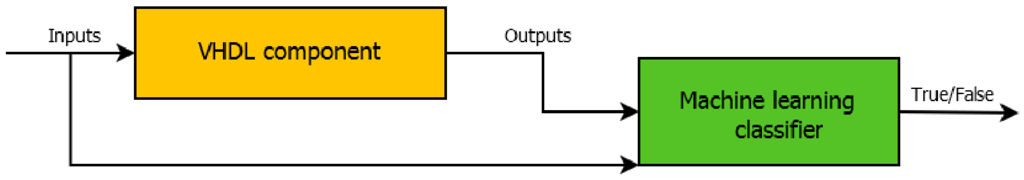


Figure 4.1: Verification process with machine learning classifier as online checker

In order to be able to precisely detect the possible faults in the monitored hardware component, the machine learning classifier must be well-trained. The training of the classifiers requires big amount of data sets whose generation is just as important as the classifier itself. In the below sections can be read a thorough description of the data set generation process, including the used data set formats.

4.1 Fault injection

Machine learning algorithms also require data, produced by some faulty behaviour, in order to be able to detect the occurring faults. Generation of faulty data instances can be achieved by employing various fault injection (FI) techniques.

In this thesis fault injection is realized by parametrizing the original hardware component and inverting specific bits in the component's logic based on the values of the added parameters. The parameters are added as new input ports for the hardware component, so they can be driven from a testbench program (described in 4.2.2). Fault injection can be turned on/off using a boolean parameter `gen_err`. If it has `false` value, no fault injection is performed and the values of the other FI parameters have no effect on the functionality of the hardware component. Otherwise, the rest of the FI parameters specify where the given fault should be injected (in which signal, which bit should be inverted). The time-interval when a fault should be injected is adjustable using the the main parameter `gen_err`. By setting it to `true`, without setting it back to `false`, permanent faults can be injected. In the same manner also transient faults can be injected: set it to `true`, wait based on what is the desired time-interval of the fault, and set it back to `false`.

There are also other FI techniques which have more advanced settings, for example in [25] is presented a fault injection system which works on both combinational and sequential digital circuits and is able to inject both transient and permanent faults.

Generation of data sets is usually performed by simulating the given hardware component. Therefore, it would be also possible to inject faults using the functions of the used simulator (for example ModelSim[1]). Most of the simulators support forcing the simulated hardware components signals to another (faulty) value. The advantage of this technique is that the original hardware component does not need to be extended with further parameters and their processing. Even if the currently used approach works well for the purpose of this thesis, it could be considered as a possible future improvement to use this technique instead.

4.2 Data sets for machine learning

The data sets used in machine learning have a few common properties which need to be defined before the description of the individual data set formats. A data set can be defined as a set of *instances* where usually one data set instance is one row of the data set file. The size of the data set is given in terms of number of instances. Each data set instance consists of a specific number of *attributes* containing the data for the algorithm's learning. In case of supervised learning, data set instances must also contain a special *class* attribute which provides an extra information about the currently learned data set instance's affiliation (e.g. to which class the given data set instance belongs). The representation of the data set attributes can be different for each data set format which is described in the below subsections.

4.2.1 Data set formats

Since the experiments of this thesis were performed using the Weka machine learning framework, the data set types and their format are also introduced through its terminology.

Weka data set format

The Weka machine learning framework supports two main data set formats. Its data sets can be stored either in `.csv` (comma-separated values)[2] files or in special `.arff` (*Attribute-*

[1] <`https://www.mentor.com/products/fv/modelsim/`>
[2] <`https://en.wikipedia.org/wiki/Comma-separated_values`>

Relation File Format)[3] files. The `.csv` file format is rather a universal file format, while the `.arff` format was designed to store specifically machine learning data sets for training and testing. Therefore, in this thesis the `.arff` format was used. The content of an *.arff* file can be divided into two sections:

1. **header section** - contains the relation declaration and the declaration of n attributes:

   ```
   @relation <relationName>
   @attribute <attributeName_1> <dataType_1>
   @attribute <attributeName_2> <dataType_2>
   ...
   @attribute <attributeName_n> <dataType_n>
   ```

   In `.arff` 4 attribute types are supported: *numeric*, *string*, *date* and *nominal*. A typical nominal attribute is the *class* of data set instances. It is declared as a set of possible values, for example if the instances should be classified based on the color of the described object:

   ```
   @attribute class {black , white , red , green}
   ```

2. **data section** - contains the data declaration and the list of data set instances. Each line (instance) contains comma-separated values of the individual attributes:

   ```
   @data
   <attributeValue_1>,<attributeValue_2 >,... ,< attributeValue_n>
   <attributeValue_1>,<attributeValue_2 >,... ,< attributeValue_n>
   ...
   ```

The `.arff` file format can have one of the 2 following sub-formats:

- **propositional format** - flat file format which similarly to the `.csv` file format cannot be used for describing relations between the individual attributes of one data set instance. The above described properties apply to this format.
- **multi-instance format** - each data set instance has a hierarchical structure where on the first level there are only 3 attributes:
  - ***bag-id*** - nominal attribute which holds an unique identifier for the given *bag*
  - ***bag*** - relational attribute containing another level of m attributes into which coherent data is generated. The bag attribute introduces a new attribute type: *relational* which can be defined as follows:

    ```
    @attribute bag relational
            @attribute <bagAttributeName_1> <dataType_1>
            @attribute <bagAttributeName_2> <dataType_2>
            ...
            @attribute <bagAttributeName_m> <dataType_m>
    @end bag
    ```

[3] `<http://weka.wikispaces.com/ARFF+\%28book+version\%29>`

The values for these attributes can be specified multiple times in a data set instance, creating a nested sequence of instances. The number of nested instances in the bag can be defined as the length of the bag: *bag_len*. It means that the bag contains *bag_len*-times *m* attributes. The values of the *bag*'s attributes are surrounded by quotes in the data section of the file. The nested instances of the *bag* are separated by line-feeds (\n):

```
@data
<bag_id>,"<b_1-i_1>,<b_2-i_1>,...,<b_m-i_1>\n
              <b_1-i_2>,<b_2-i_2>,...,<b_m-i_2>\n...",<class>
```

– ***class*** - nominal attribute specifying the class label for all the instances residing in the given *bag*

The following example shows how time series data can be stored in the *multi-instance format* which was used in this thesis. Let's consider a hardware component with sequential logic, processing time series data. The component has 1 numeric input value and 2 numeric output values in each time-step. The instances of the data set can be classified as *OK* or *ERR*. For the sake of simplicity the data set will consist of 4 instances only. The header section of the `.arff` file can be defined then as:

```
@attribute bag-id {id_0, id_1, id_2, id_3}
@attribute bag relational
        @attribute input numeric
        @attribute output_one numeric
        @attribute output_two numeric
@end bag
@attribute class {OK, ERR}
```

First, it needs to be determined how many subsequent input-output values of the component we would like to have in one data set instance. Let's consider having 3 I/O values in the data set instances, which means that the *bag* attribute will have 3-times 1 input and 2 output values. The data section of the `.arff` file could look like as follows (with random numeric values):

```
@data
id_0,"1,5,6\n7,2,7\n5,4,1",ERR
id_1,"5,4,4\n1,1,5\n4,7,2",OK
id_2,"7,3,4\n5,5,1\n9,5,3",OK
id_3,"5,1,5\n4,6,9\n4,1,6",ERR
```

4.2.2 Data set generation

Data set generation is based on the given hardware component which needs to be verified whether it works correctly. The first phase of data set generation includes simulation of the given hardware component to gain its input-output values for further processing (see 4.2). The simulation is performed using ModelSim 10.0[4] developed by Mentor Graphics. The *Reference component* (or *Golden model*) is the original hardware component which produces correct outputs for its input values during the simulation. Fault injection (FI) is

[4] `<https://www.mentor.com/products/fv/modelsim/>`

performed on the *Fault-prone component* which is identical with the *Reference component*, containing some additional parameters suitable for fault injection (described in 4.1). To simulate the involved components a *Testbench program* was implemented. The testbench passes data read from the *Input file* to the two components without modification. It is also responsible for passing FI parameters to the *Fault-prone component* which leads to a faulty behaviour. The *Testbench program* was designed to set the FI parameters either randomly or based on some generic parameters (e.g., received from command line). One of these generic parameters is the length of the component's I/O series generated into one data set instance (in `.arff`: *bag_len*). The time-interval, when a fault should be injected, is specified relatively to the I/O series in the individual data set instances by specifying the starting time-step (index) and the length of the fault. For example if we set *bag_len* to 5, the starting time-step to 2 (counted from 0) and the length of the fault to 2, the data instance in `.arff` would look like:

```
<bag_id>,"<OK_0>\n<OK_1>\n<ERR_2>\n<ERR_3>\n<OK_4>",<ERR_class>
```

where <OK_*i*>/<ERR_*i*> represent correct/faulty I/O values at time-step *i*.
While simulating the two components (reference and fault-prone) in parallel, the testbench stores the inputs and the subsequently received outputs of the components to separate files. *File of correct I/O data* will contain only the inputs and outputs from the *Reference component* and *File of faulty I/O data* only the inputs and outputs from the *Fault-prone component*. Therefore, the input values in both files will be the same, only the related output values will be different. The files have `.csv` file format where the comma-separated input-output values have binary representation. The simulation process is illustrated on the below figure 4.2.

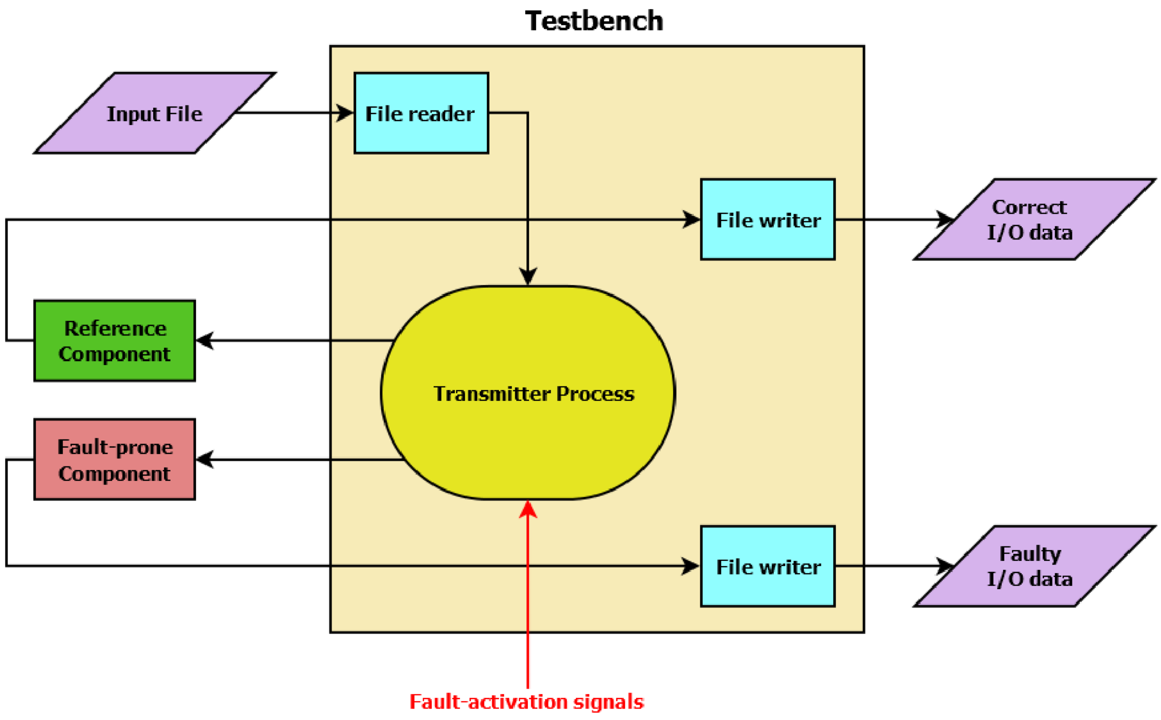


Figure 4.2: Simulation using testbench

The second phase of the data set generation is about post-processing the data produced by the first phase. Since the data stored in the two output files *File of correct I/O data* and *File of faulty I/O data* is not suitable for immediate processing by a machine learning algorithm, it needs to be transformed first. The files are read by the *Data set generator*, which is a Python script for converting and mixing the correct and faulty I/O data into the final data set files: *Train data set*, *Validation data set* and *Test data set*. The *Data set generator* works based on a set of input parameters which specify how the conversion and mixing should be performed. The conversion usually means transformation of the individual I/O values from binary to decimal representation, but it can also involve redistribution or omission of some I/O values. For example if a component has an input or output value which has no real impact on the subsequent learning process, it can be left out from the final data set. The generator is also responsible for setting the class attribute for the individual instances in the generated data sets. The class attribute can have only two values in the experiments of this thesis: „*0*“ for faulty data set instances and „*1*“ for correct data set instances. Therefore, its value depends on how the given data set instance is mixed from the two I/O data files. If the data set instance contains at least one output value taken from the *File of faulty I/O data*, the class will have „*0*“ value. Otherwise, it is considered as a correct instance, with class value „*1*“. An exception is when the output values of the two mixed files are identical for the same input value. This means that the fault injected to the *Fault-prone component* did not induce a faulty-behaviour. In such cases the given data set instance will also have class value „*1*“. The process of data set generation is summarized in the following flowchart diagram:

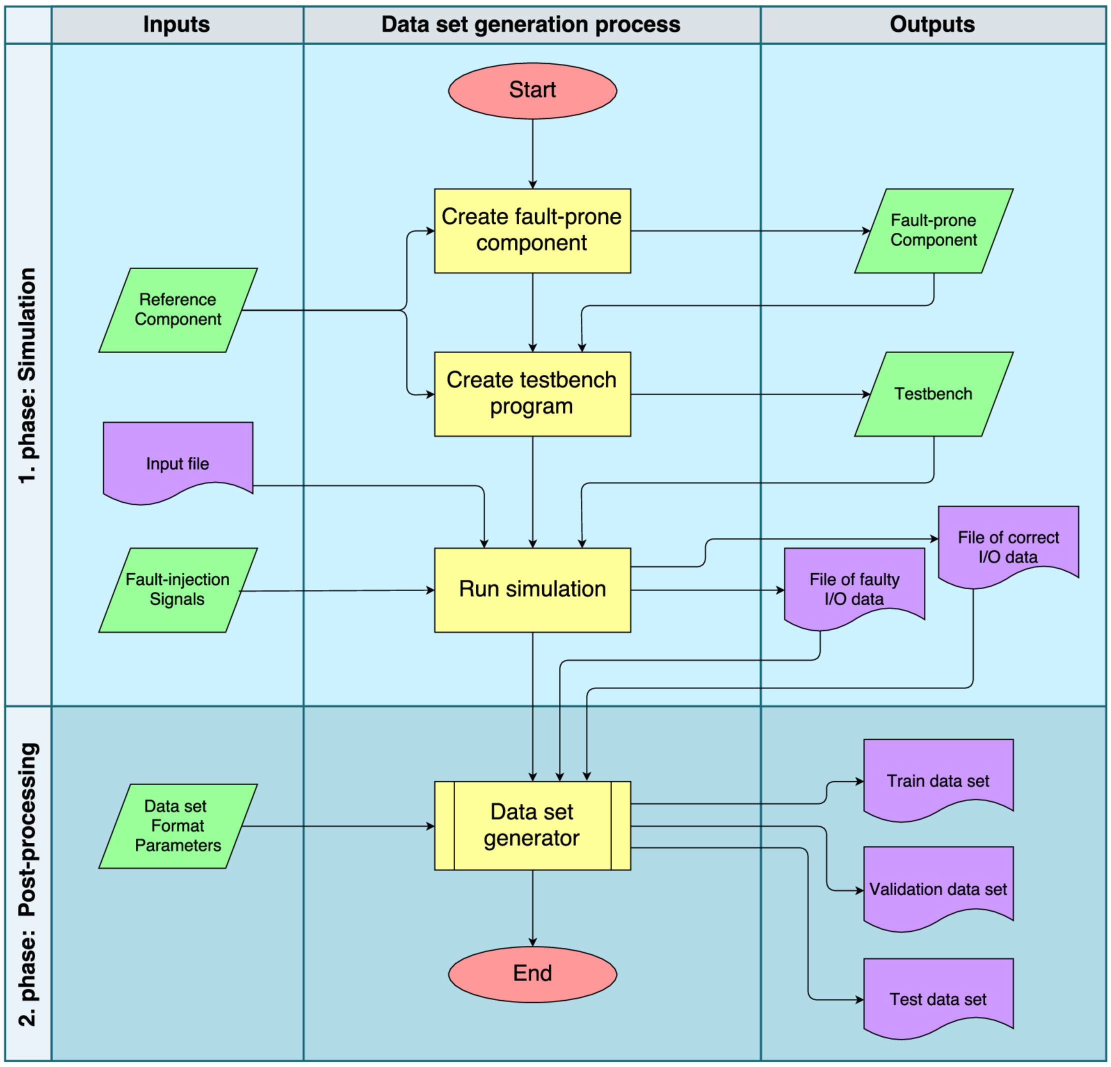


Figure 4.3: Data set generation process

4.3 Used machine learning algorithm implementations

The below subsections describe the origin and the parameters of the specific machine learning algorithm implementations used in this thesis.

4.3.1 HMMWeka library

For running experiments using a hidden Markov model, the HMMWeka library, developed by Marco Gillies [12] was used. It is available as an external package for the Weka machine learning framework [13]. The library offers several parameters to adjust the algorithm's classification accuracy, described on: `<http://www.doc.gold.ac.uk/~mas02mg/software/hmmweka/index.html>`.

The library contains solutions for both *Training* and *Scoring* HMM problems (see 3.2.2.3) which are essential for the classifier's use and its evaluation. The training is implemented using the Baum-Welch algorithm with the expectation-maximization algorithm and forward-backward procedures. The scoring problem is realized using the forward-algorithm.

One of the library's missing part was the solution for the *Matching/Decoding* problem. The implementation of the Viterbi algorithm is part of the thesis. It was added to the HMMWeka library to gain additional statistics about the distribution of the output values between the model's states.

4.3.2 Random Forest

The RandomForest algorithm is used from Weka's `weka.classifiers.trees.RandomForest` class. The parameters of the algorithm are described on documentation page [22].

4.3.3 Multilayer Perceptron

This algorithm is used from Weka's `weka.classifiers.functions.MultilayerPerceptron` class. The parametrization of the classifier is described on page [51].

4.3.4 Transformation methods for non-time series data learners

The methods of this section transform time series data to a reduced form on which standard (non-sequential) learning algorithms (see 3.3) can be applied. Since the experiments which attempted to use non-sequential learners were performed using the Weka machine learning framework [13], the implementation of these transformation methods was also used from this framework.

MIWrapper

MIWrapper is a simple wrapper method for applying standard propositional learners to multi-instance data. It was developed by E. T. Frank and X. Xu [10]. MIWrapper converts the multi-instance data set by separating the nested instances of the relational bag attribute (*Instance 0* to *Instance n* on figure 4.4) to distinct mono-instance data set instances (see figure 4.5). The class of the bag is copied to all the separated instances. The MIWrapper algorithm's base classifier is trained on all the mono-instance instances, gained by transforming the multi-instance instances of the training data set. The below figures illustrate this conversion on a data set which has two attributes (X and Y) in its bag.

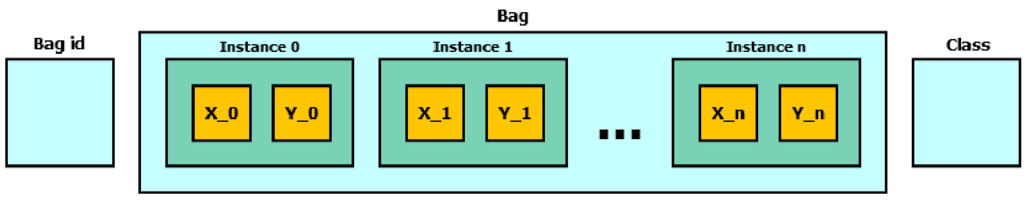


Figure 4.4: Multi-instance data set instance

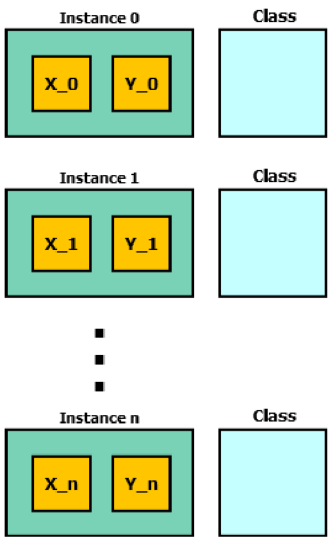


Figure 4.5: Converted multi-instance data set instance to mono-instance data set instances

Testing of the trained base classifier starts with computing the class-probability distribution for each nested instances $(0 \ldots n)$ that the given bag contains. The final class-probability distribution for the original multi-instance data set instance is computed then by one of the following approaches:

- **arithmetic averaging:** from the class-probabilities of the nested instances an arithmetic average is calculated
- **geometric averaging:** from the class-probabilities of the nested instances a geometric average is calculated
- **maximal probability selection:** the highest probability of the nested instances' class-probabilities is selected

SimpleMI

The SimpleMI wrapper algorithm transforms the original multi-instance (time series) data to mono-instance data before both training and testing. After the transformation the base algorithm can work on the simplified mono-instance data which has no longer sequential character. The reduction of the time series data can be performed by one of the following methods:

- mean value selection: selects the mean value from the bag instances' individual attributes. Based on figure 4.4 the new data set instance would look like:

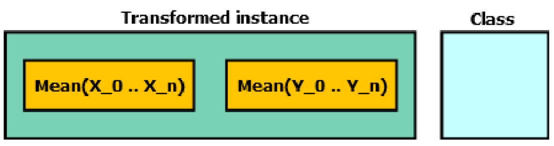


Figure 4.6: Transformed multi-instance data set instance using mean value selection

- averaged minimax: selects the minimal and maximal values from the bag instances' individual attributes and computes their average:

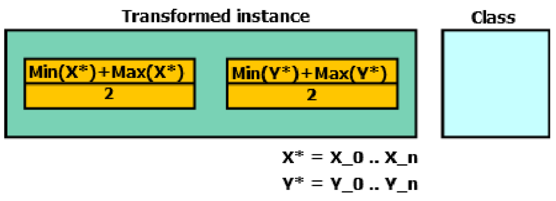


Figure 4.7: Transformed multi-instance data set instance using minimax averaging

- merged minimax: after selecting the minimal and maximal values (same as in previous case), the selected instances are merged together:

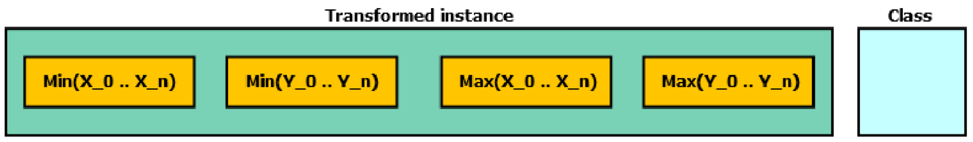


Figure 4.8: Transformed multi-instance data set instance using minimax merging

Chapter 5

Experiments on a finite impulse response filter

This chapter is a collection of experiments which were performed in order to determine which machine learning algorithms can be used for generating checking circuits. This thesis aimed to find algorithms suitable for checking hardware components with sequential logic. For this purpose a finite impulse response (FIR) filter was chosen, which is described in section 5.1.

The goal of the machine learning algorithms of these experiments is to detect the occurring faults in the FIR filter. Using the most accurate algorithms it could be possible to create an online checking circuit which would be able to detect faults in real-time and thus build a fault-tolerant FIR filter. Since FIR filter is used mainly in signal processing, this experiment utilized it for processing audio (`.wav`) files, which is illustrated on the below figure (see 5.1).

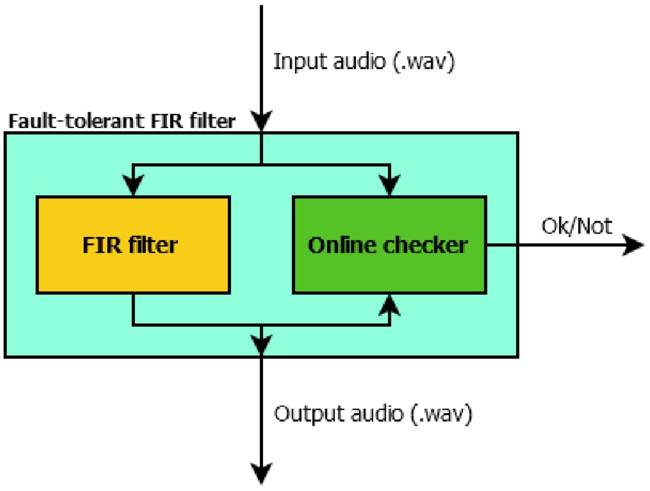


Figure 5.1: Fault-tolerant FIR filter

5.1 FIR filter design

FIR filter is one of the most frequently used filters in signal processing. Its impulse response is finite because it settles to zero in a finite number of sample intervals. FIR filters can be distinguished based on different aspects. They can work either in discrete or continuous time and can be either digital or analogue. This experiment deals with digital discrete-time FIR filters, thus the following descriptions will be restricted to this filter type.

A general discrete-time FIR filter consists of 3 basic components (see figure 5.2): multiplier ($\triangledown$), adder ($\sum$), delay unit (Z^{-1}) and for its implementation it also needs to have some memory to store the filter's coefficients ($h_{0 \ldots N}$). The coefficients influence the filter's behaviour, for example whether it will pass low or high frequencies only.

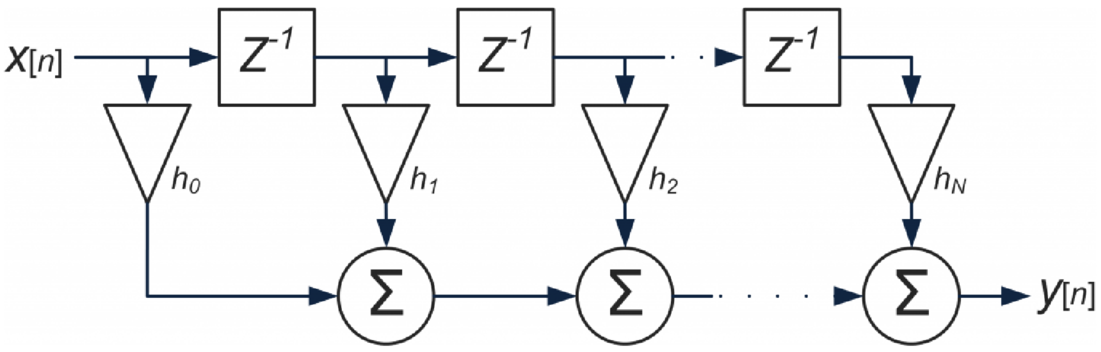


Figure 5.2: General discrete-time FIR filter [21]

The output of this discrete-time FIR filter is defined by the next formula:

$$y[n] = h_0 x[n] + h_1 x[n-1] + \cdots + h_N x[n-N] = \sum_{i=0}^{N} h_i x[n-i]$$

where:

- $x[n]$ is the input signal
- $y[n]$ is the output signal
- N is the filter order
- h_i is the i-th coefficient of the filter

For the purpose of this experiment was chosen an already implemented parametrizable FIR filter, developed by D. Pardo [35]. It has internal fixed-point implementation (12 bit decimal part resolution) designed as a low-pass filter with 15 symmetric coefficients ($h_0 \ldots h_{14}$). The FIR filter's configuration for this experiment is shown on the figure 5.3.

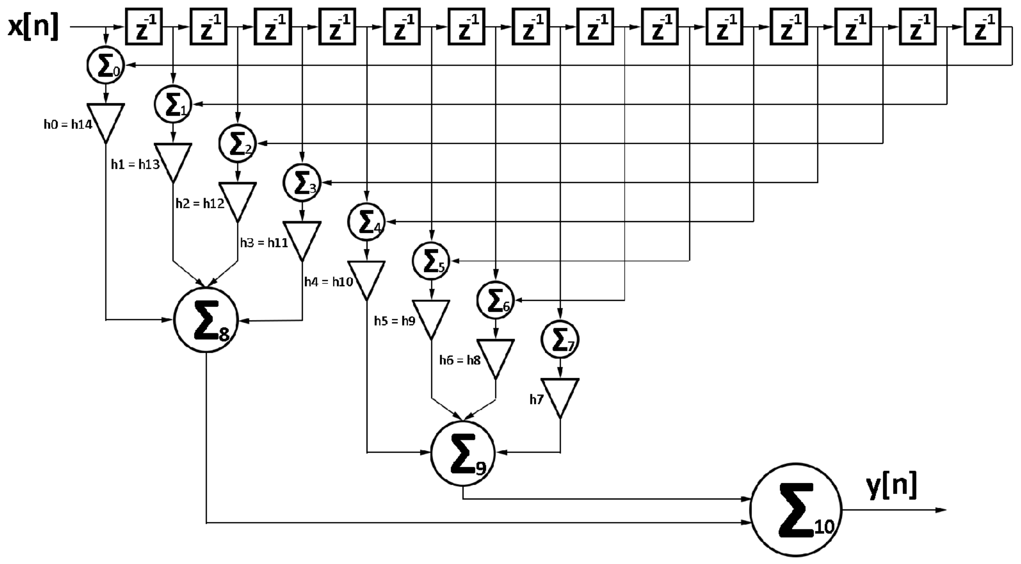


Figure 5.3: FIR filter with symmetric coefficients

The output for this symmetric FIR filter configuration can be defined in the following way:

$$y[n] = h_0(x[n]+x[n-14])+h_1(x[n-1]+x[n-13])+\cdots+h_6(x[n-6]+x[n-8])+h_7x[n-7]$$

where the coefficients $h_0 = h_{14}, h_1 = h_{13} \ldots, h_6 = h_8$ are equal, except the middle coefficient h_7.

5.2 Comparing audio files

In the described FIR filter faults can occur with different severities which is manifested in the resulting audio file. The more serious the occurred fault is, the more damaged the resulting audio output will be. Since it is more important to detect faults which are more serious, it is necessary to distinguish them based on this property. This can be either done by subjective listening tests or the PEAQ algorithm. Both methods require two signals for their operation, a test signal and a reference signal, which are compared to each other. In case of this experiment the reference signal is a product of a faultless FIR filter, generated for the same FIR input signal for which the test signal was generated.

5.2.1 Subjective listening tests

Subjective listening tests involve classification of reference and test signals based on the ITU Radiocommunication Sector's (ITU-R)[1] 5-anchor grade (treated as continuous, see figure 5.4). ITU-R is one of the three sectors of the International Telecommunication Union (ITU), next to the Telecommunication Standardization Sector (ITU-T) and Development Sector (ITU-D). It is mainly responsible for managing the international radio-frequency spectrum and satellite orbit resources.

[1] <http://www.itu.int/en/ITU-R/information/Pages/default.aspx>

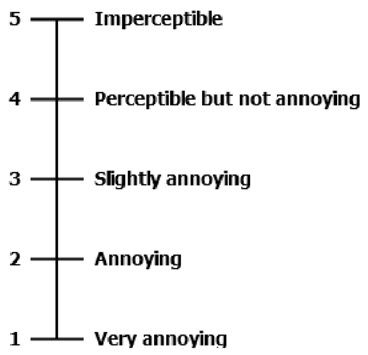


Figure 5.4: The ITU-R five-grade impairment scale [17]

Generally, the final output of subjective listening tests is the Subjective Difference Grade (SDG), defined as:

$$SDG = Grade_{TestSignal} - Grade_{Reference_Signal}$$

where $Grade_{TestSignal}$ and $Grade_{Reference_Signal}$ are from the above scale (see 5.4). Therefore, the SDG values should be from interval 0 to -4 where 0 corresponds to an imperceptible difference and -4 to an impairment considered as very annoying.

Comparing audio files by subjective listening tests is both time consuming and expensive because without an expert who is able to judge the audio quality, the results are not reliable. Therefore, a method for objective measurements of the perceived audio quality (Perceptual Evaluation of Audio Quality, PEAQ) was used, recommended by ITU [17].

5.2.2 PEAQ algorithm

The basic concept of the PEAQ algorithm is to make objective measurements on the reference and test audio file, and try to predict the outcome of subjective listening tests. The output of the PEAQ algorithm is the Objective Difference Grade (ODG) with resolution limited to one decimal. Since the ODG values should be as close as possible to the SDG values obtained for the same input signals, their grading is identical, as shown on the below figure 5.5. According to [17] the general difference between two ODG values of a tenth of the grade is not significant.

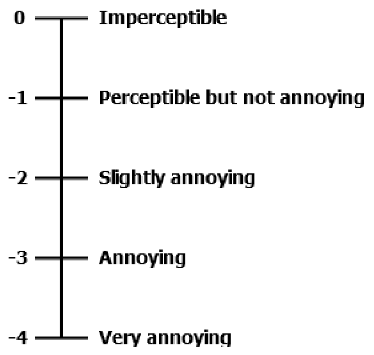


Figure 5.5: Objective/Subjective Difference Grade

The steps of the PEAQ algorithm are summarized in the below figure (see 5.6). The algorithm will not be described in detail in this thesis, for more information refer to [17].

The ear model is based on the Fast Fourier Transformation (FFT) and decomposes the input signals in order to apply weighting processes and to send the signals to other preprocessing methods (e.g. loudness, mask and modulation calculation). After the preprocessing phase, based on the calculated patterns the algorithm calculates various module output variables (MOV), which capture different aspects of the the given signals. Finally, the MOVs are passed to an artificial neural network which outputs the resulting ODG value.

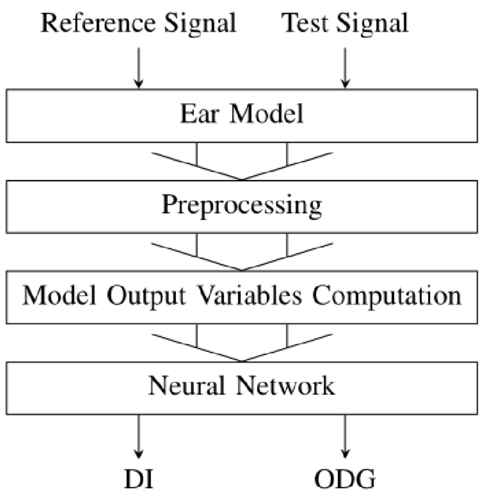


Figure 5.6: Steps of the PEAQ algorithm [26]

The PEAQ algorithm has more implementations, but most of them does not meet the conformance requirements recommended by ITU, declared in [17]. For the purpose of this thesis an open-source implementation was chosen which does not meet the conformance requirements neither, but its obtained results closely resemble the ones of listening tests. It is called GstPEAQ [26] and is available as a GStreamer[2] plug-in.

5.3 Configuration of the injected faults

The faults which were injected to the FIR filter were produced based on the following parameters:

- ***err_sum***: Faults were injected to the result of the FIR filter's individual adders, therefore this parameter specifies the index of the affected adder: from 1 to 10 (see $\sum$-s on figure 5.3).
- ***err_inv_bit***: This parameter specifies which bit of the given adder's result is inverted. For adders $\sum_1 \dots \sum_7$ which have 10-bit result values it can have 0 to 9 values. For adders $\sum_8$, $\sum_9$ and $\sum_{10}$ which have 20-bit result values it can have 0 to 19 values.
- ***err_sample_len***: The duration of the given fault, given in number of samples when the given fault is present in the filter. In the below described experiments this parameter can have one of the following values: $\{3, 6, 12, 24, 30\}$.

[2] <gstreamer.freedesktop.org>

Detecting faults which cause significant (audible) differences in the resulting output signal is more important than detecting almost imperceptible faults. Since each fault-configuration can have different impact on the resulting audio quality, it was necessary to measure which configuration produces faulty audio signals with an adequate ODG value. The ODG value was measured using the previously described GstPEAQ [26] tool, by comparing the FIR filter's faultless output with the ones which include one of the fault types in themselves. The results of these measurements revealed that on adders $\sum_0$ to $\sum_4$ no significant fault can be induced, only adders $\sum_5 \ldots \sum_{10}$ show more significant faults which could be worthy to detect. The below graphs (see 5.7 and 5.8) show the measured ODG values on some of the adders, the rest of the graphs are presented in Appendix B.1. Each graph shows all the possible fault-types on the given adder: the x-axis has information about the inverted bit's position with 5 bars describing the length of the given fault as a number of fault-injected samples. On the y-axis are presented the ODG values which were computed as an average from 10 measurements. The range of y-axis was chosen to be the same on all below presented graphs in order to facilitate the comparison of results on the individual adders. Each ODG value on the graphs was measured by injecting only one fault configuration (*err_sum*, *err_inv_bit*, *err_sample_len* combination) to the FIR filter. The given fault was injected for a given time-interval: specified by a randomly generated starting sample and by the length of the injected fault (*err_sample_len*). The starting sample was different for each of the 10 measurements from which the average ODG was computed.

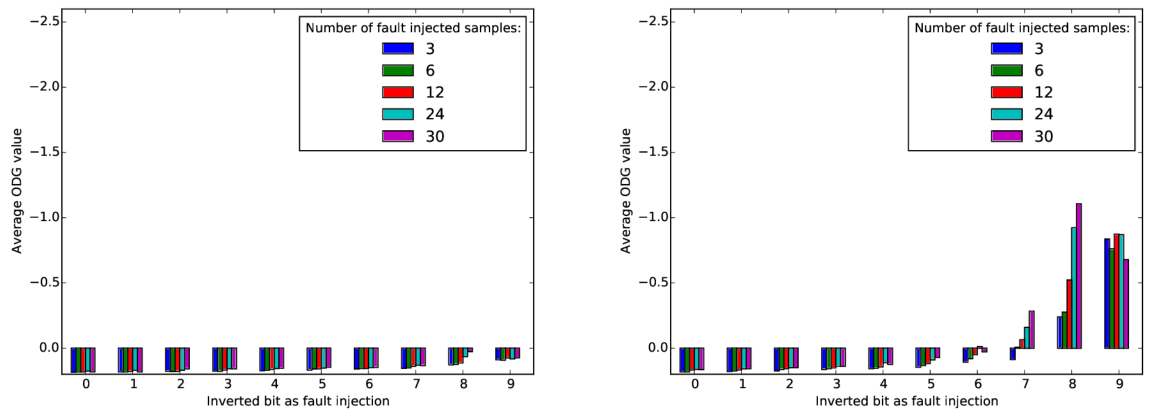


Figure 5.7: ODG values on $\sum_0$ (left) and $\sum_4$(right)

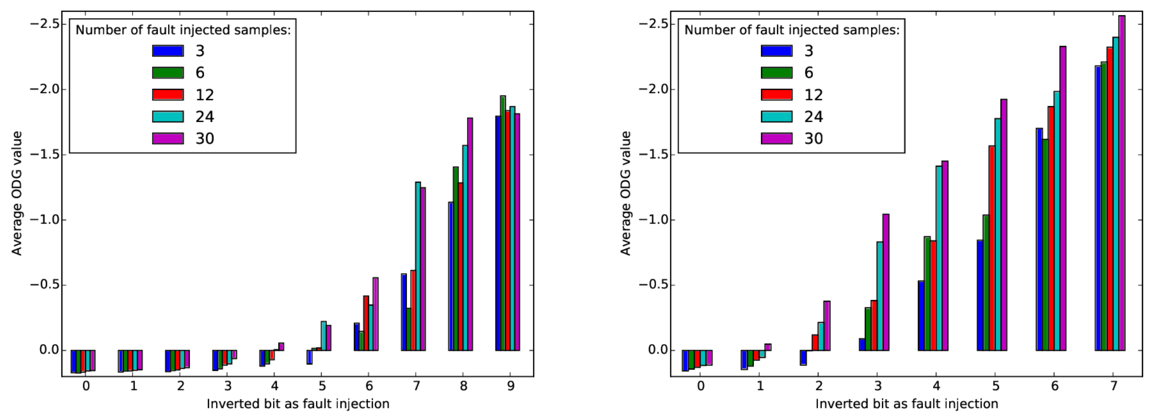


Figure 5.8: ODG values on $\sum_5$ (left) and $\sum_{10}$(right)

5.4 Data sets

The FIR filter's sequential logic requires the use of data set formats which are able to describe time series data. Therefore, the data sets of this experiment all use the Weka machine learning framework's multi-instance format (described in 4.2.1).

The experiments were performed using two types of data sets. The first data set type is called *fir_30_io_vals*. Its bag attribute consists of 30 subsequent FIR input-output values:

$$(in_0, out_0); (in_1, out_1); \ldots ; (in_{29}, out_{29})$$

The second data set type is called *fir_diff_io_vals*. Its bag contains the differences of the subsequent FIR input and output values:

$$(|in_0 - in_1|, |out_0 - out_1|); (|in_1 - in_2|, |out_1 - out_2|); \ldots ; (|in_{28} - in_{29}|, |out_{28} - out_{29}|)$$

In both cases the individual values have decimal representation. The class of the data set instances can have two possible values: „*1*“ if the bag's input-output values were generated by the reference FIR filter, „*0*“ if they were generated by the fault-injected FIR filter. The data sets are generated as described in 4.2.2. The exact content and size of data sets differs in the following experiments (see 5.5 and 5.6).

5.5 Parameter settings for algorithms

The experiments described in this section were performed in order to determine which parameter settings of the individual machine learning classifiers yield better classification results. The source of the data sets were 3 different .wav files which were mixed together to get a reliable data set for learning. The train data set contained 5000 instances, the validation set 2500 instances and the test set 1500 instances, all with cca 50% distribution of the classes (0 or 1). After training the individual classifiers on the training data set, 5 separate validation data sets were used. The classifiers were evaluated on each of the validation sets, and from the measured classification accuracies an average value was calculated. The below experiments use this average classification accuracy value without explicit mention. The final (best) parameters for the individual machine learning classifiers were determined based on their achieved results on the training and testing data set. Subsequently, the classifiers were evaluated on the testing data set to measure their final classification accuracy.

In the experiments of this section, the calculated ODG values were not taken into account when injecting faults to the FIR filter. Therefore, the faults were generated randomly from the previously described fault configurations (see 5.3).

The below results are all presented on the *fir_diff_io_vals* data set (see 5.4) because the classification on the *fir_30_io_vals* data set was less successful (see Appendix B.2).

5.5.1 MIWrapper

Initially, it was planned to use the MIWrapper algorithm in the experiments on the FIR filter, however, it was diagnosed that for this use-case it is not suitable. The reason lies in MIWrapper's conversion of multi-instance data instances to distinct mono-instance data instances, which is described in section 4.3.4. Let's consider a FIR I/O value sequence, where the first 20 I/O values are correct, and the last 10 were generated after a fault has been injected to the filter. For the sake of simplicity, the below example is presented on a general FIR I/O data sequence, but the problem is the same for both *fir_30_io_vals* and *fir_diff_io_vals* data set types.

```
<OK_0>,<OK_1> , . . . , <OK_19>,<ERR_20> , . . . , <ERR_29>,<ERR_class>
```

After the conversion of the above presented multi-instance data instance we get the following mono-instance data instances:

```
<OK_0>,<ERR_class>
<OK_1>,<ERR_class>
 . . .
<OK_19>,<ERR_class>
<ERR_20>,<ERR_class>
 . . .
<ERR_29>,<ERR_class>
```

The problem is that MIWrapper copies the class of the original multi-instance data instance to all mono-instance data instances. Since the first 20 mono-instance data instances are not faulty, the given base classifier would be mislead during the training. Therefore, this method is rather not used in this thesis.

5.5.2 Hidden Markov model

Hidden Markov model (HMM) was used with the following settings: full-matrix untied covariance, ergodic transition model, non-random state initialization (see 3.2.2.3 and 4.3.1). The number of states needed for reaching the classifier's best accuracy was measured on the training and validation data set. The results (see graph 5.24) show that 6 states are ideal for the model's desired functioning.

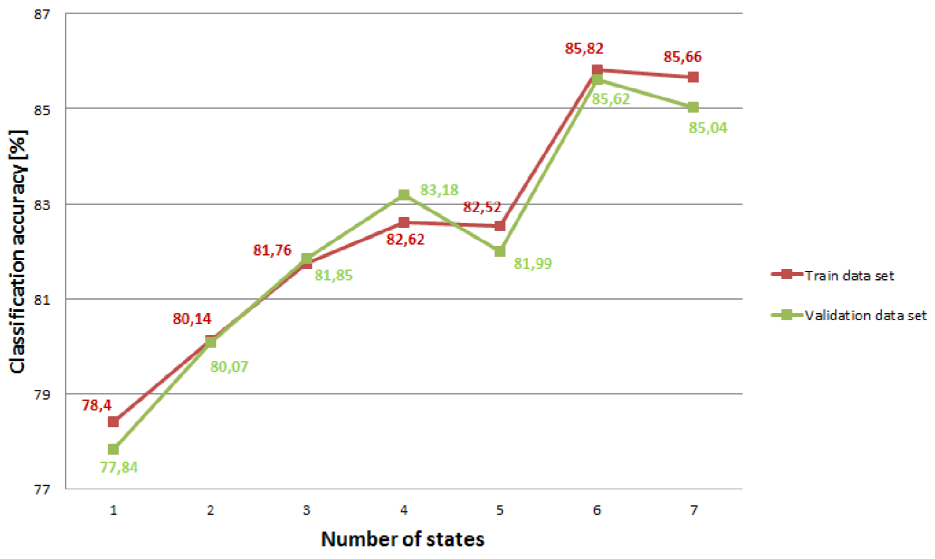


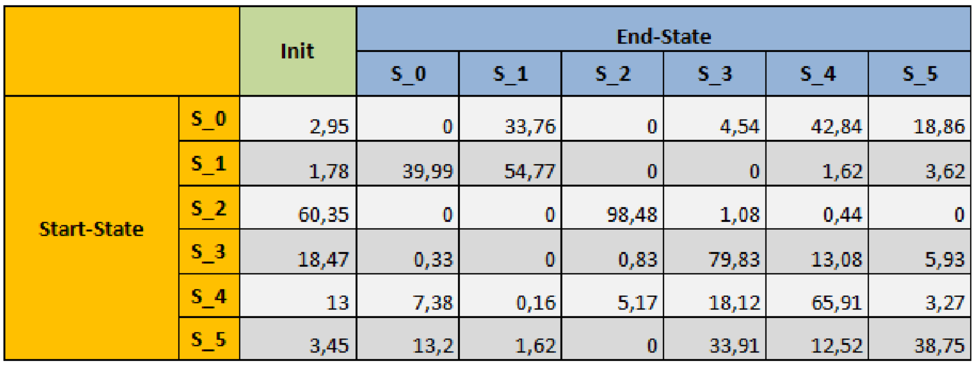
Figure 5.9: HMM classification results by number of states

The subsequent evaluation of the classifier on the testing data set showed **86.4%** accuracy.

Except determining the number of states required for the HMM's proper operation, there are two other HMM parameters which can be evaluated: *transition probability distribution* and *observation probability distribution*. Both of them are described in the theoretical part of the thesis (see 3.2.2.3). The remaining part of this subsection analyses these two parameters of the above evaluated HMM (with 6 states). The aim of this analysis is to reveal whether the chosen number of states is reasonable. For example, using unnecessarily many states should be avoided.

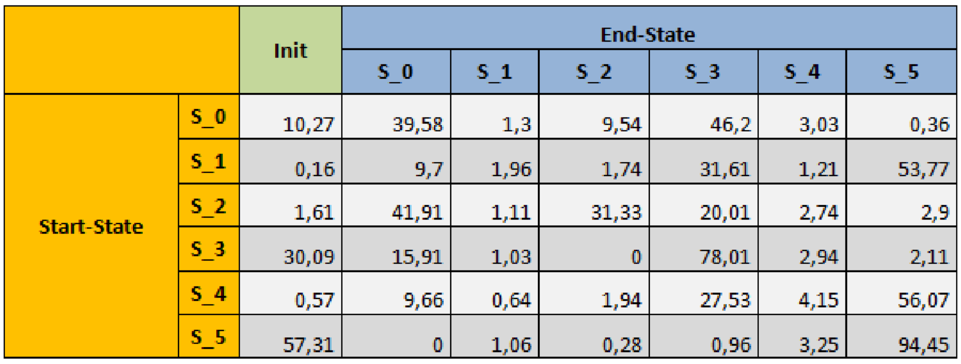
To perform classification using hidden Markov models for each class a separate model is trained. The training and the subsequent classification (scoring) is described in section 3.2.2.3. In this experiment two HMMs were trained: one on correct time series data (*HMM_corr*), another one on fault-injected time series data (*HMM_err*). The below graphical representations allow the comparison and better comprehension of these models.

Transition probability distribution can be either expressed using a state transition diagram or a table containing the transition probabilities between the individual states of the model. Since the evaluated model has 6 states, it was more transparent to tabulate the transition probabilities (see 5.10 and 5.11). The state transition diagrams were also prepared, but because of their size they were attached only to the appendix of this thesis (see B.3).



		Init	End-State					
			S_0	S_1	S_2	S_3	S_4	S_5
Start-State	S_0	2,95	0	33,76	0	4,54	42,84	18,86
	S_1	1,78	39,99	54,77	0	0	1,62	3,62
	S_2	60,35	0	0	98,48	1,08	0,44	0
	S_3	18,47	0,33	0	0,83	79,83	13,08	5,93
	S_4	13	7,38	0,16	5,17	18,12	65,91	3,27
	S_5	3,45	13,2	1,62	0	33,91	12,52	38,75

Figure 5.10: Transition probabilities between the states of *HMM_corr*



		Init	End-State					
			S_0	S_1	S_2	S_3	S_4	S_5
Start-State	S_0	10,27	39,58	1,3	9,54	46,2	3,03	0,36
	S_1	0,16	9,7	1,96	1,74	31,61	1,21	53,77
	S_2	1,61	41,91	1,11	31,33	20,01	2,74	2,9
	S_3	30,09	15,91	1,03	0	78,01	2,94	2,11
	S_4	0,57	9,66	0,64	1,94	27,53	4,15	56,07
	S_5	57,31	0	1,06	0,28	0,96	3,25	94,45

Figure 5.11: Transition probabilities between the states of *HMM_err*

Observation probability distribution is demonstrated on the observation sequences of the testing data set. For this purpose two diagrams were prepared, which are presented on figures 5.12 and 5.13. They show how the individual items of the observation sequences are distributed between the states of the given HMM. Since the data sets of this experiment have the *fir_diff_io_vals* format (see 5.4), the observation sequences consist of FIR I/O value differences. Hence, the items of the observation sequences are (in_{diff}, out_{diff}) value-pairs, where in_{diff} is the difference between two subsequent FIR input values and out_{diff} between two subsequent FIR output values. Each observation sequence of the test data set was decoded to its optimal state-sequence using the given model (*HMM_corr* or *HMM_err*). Subsequently, the items of the individual observation sequences were assigned to their decoded state. In case a given item was observed in multiple states, it was assigned to the most often decoded state. The diagrams show the in_{diff} (x-axis) and out_{diff} (y-axis) value-pairs in a 2-dimensional space. The color of the data points indicates the state of the given HMM model in which the given value-pair was observed.

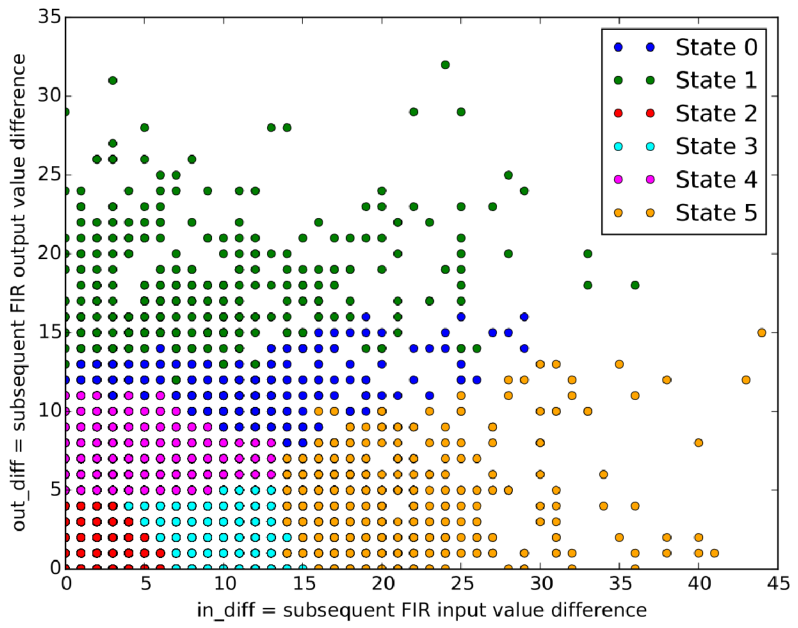


Figure 5.12: Distribution of observation sequence items between the states of *HMM_corr*

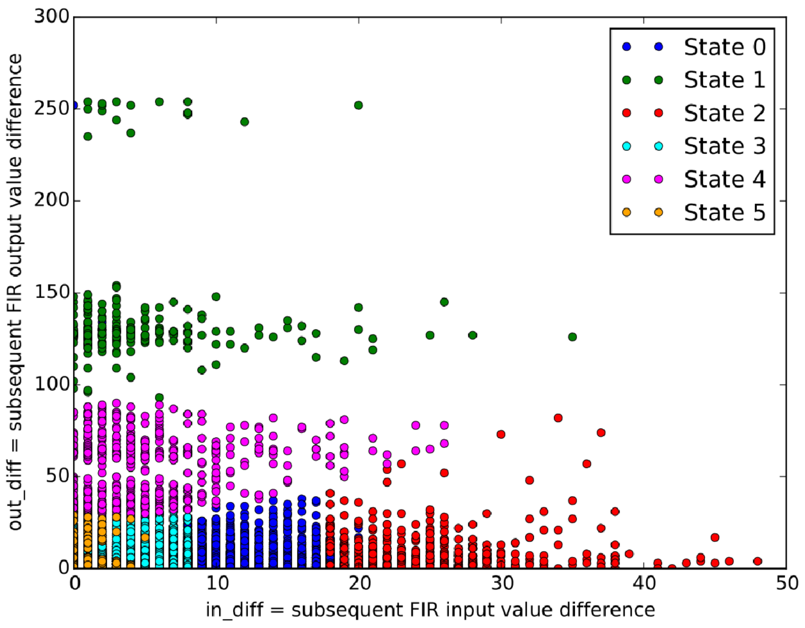


Figure 5.13: Distribution of observation sequence items between the states of *HMM_err*

The above presented diagrams show that the individual states of the trained hidden Markov models (both *HMM_corr* and *HMM_err*) each correspond to a certain range of (in_{diff}, out_{diff}) value-pairs. One of the visible differences between the two models is in the range of the y-axis (out_{diff}) values: for *HMM_corr* it is between 0 and 33 (see 5.12), while for *HMM_err* it is between 0 and 254 (see 5.13). Since model *HMM_err* was trained on fault-injected data sequences, the possible difference between two subsequent FIR output values (out_{diff}) is much higher than in case of the *HMM_corr* model.

The second difference is in the state transition probabilities of the models. Based on diagrams 5.10 and 5.12, in case of model *HMM_corr*, it can be seen that transitions between two states of the model have low probability if the data points observable in the start-state are relatively distant from the data points observable in the end-state. For example, on diagram 5.12 the observed data points of state 2, 3, 4 and 5 are all distant from the observed data points of state 1, therefore transition from states 2, 3, 4 or 5 to state 1 have all low probability (see transition probability table 5.10). The same applies to the opposite direction: from state 1 to state 2, 3, 4 or 5. However, model *HMM_err* does not work this way. On the contrary, in certain cases the transition probability between states with distant observed data points is high. For example, on diagram 5.13 we can see that data points observed in state 1 are distant from data points observed in states 3 and 5, but the state transition table 5.11 show high probabilities from state 1 to state 3 or 5. The difference between the data points observed in the two subsequent states of the transition is mainly in the out_{diff} value (on the y-axis). The subsequent observation of two very different out_{diff} values usually indicates a transition between the correct and faulty behaviour of the FIR filter. Model *HMM_err* contains such transition probabilities, because it was trained on fault-injected data. The reason why model *HMM_corr* does not contain the above described transition probabilities, is because it was trained on purely correct data sequences.

5.5.3 MultilayerPerceptron with SimpleMI

MultilayerPerceptron (MLP) was used with a single hidden layer, where the number of required neurons was assessed by the below presented experiments. The other parameters of the MLP were set to Weka's default values: 0.3 learning rate, 0.2 momentum, the activation function of the MLP was a logistic sigmoid (see 4.3.3).

SimpleMI was tested with each of its transformation methods, which are described in section 4.3.4. Diagram 5.14 shows the results gained by using the *mean value selection* transformation method. It is followed by graph 5.15, which contains the results yield by the *averaged minimax* transformation method. Finally, the classification results measured with the *merged minimax* transformation method are demonstrated on graph 5.16.

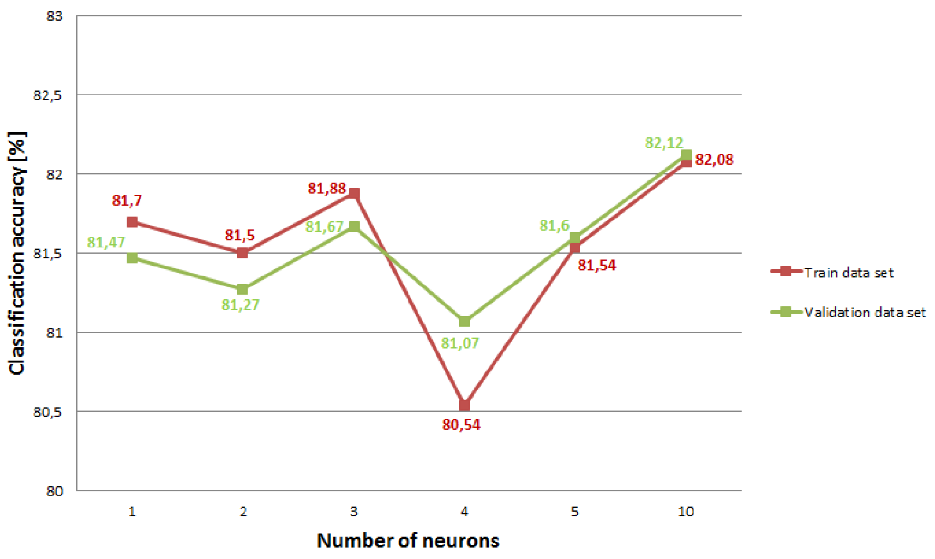


Figure 5.14: SimpleMI with mean value selection using MLP as base classifier

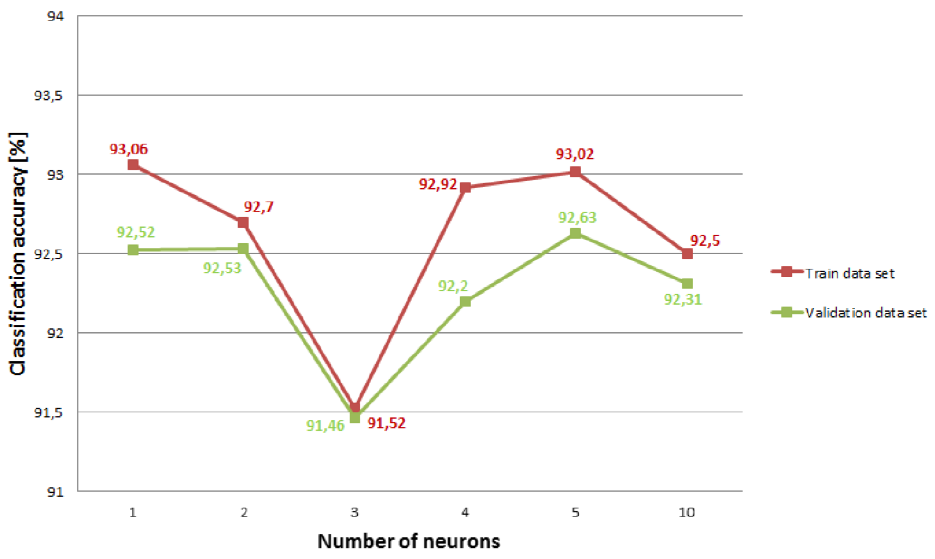


Figure 5.15: SimpleMI with averaged minimax using MLP as base classifier

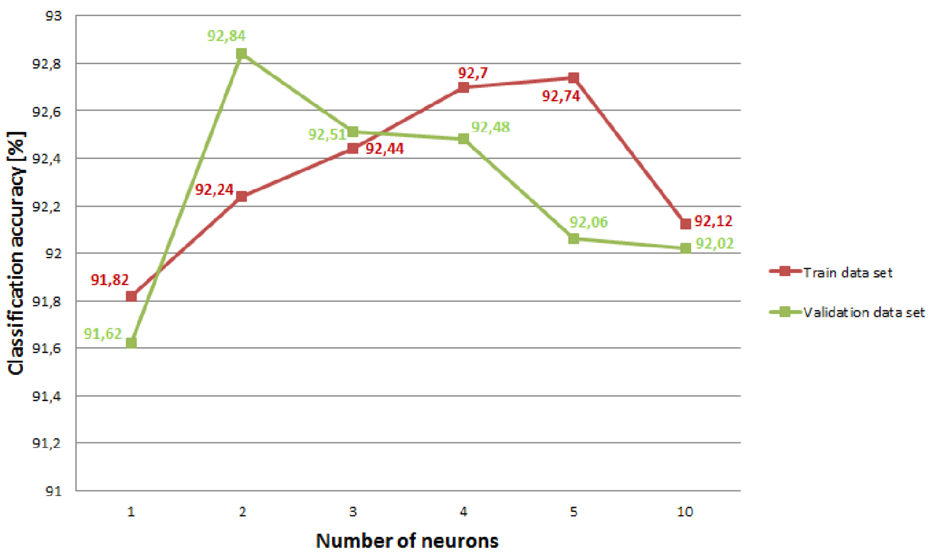


Figure 5.16: SimpleMI with merged minimax using MLP as base classifier

The best results were produced by SimpleMI with the *averaged minimax* transformation method, hence it was chosen for further experiments. Graph 5.15 shows that it is enough to have one single neuron in the MLP's hidden layer to maximize the neural network's classification accuracy. Its evaluation on the testing data set showed **92.73%** accuracy.

On the below figure 5.17 are described the settings of the trained MLP classifier. The differences between the subsequent FIR input values are denoted as vector d_in:

$$(|in_0 - in_1|, |in_1 - in_2|, \dots, |in_{28} - in_{29}|)$$

Similarly, d_out is a vector of subsequent FIR output value differences:

$$(|out_0 - out_1|, |out_1 - out_2|, \dots, |out_{28} - out_{29}|)$$

The inputs of the neural network are set accordingly to the SimpleMI wrapper method with *averaged minimax* transformation. There are two output neurons, one for each class, where *Class 0* is the class representing a faulty FIR I/O sequence and *Class 1* represents a correct FIR I/O sequence. The weights between the neurons are expressed using values of w. The threshold in the individual nodes are denoted as values of t.

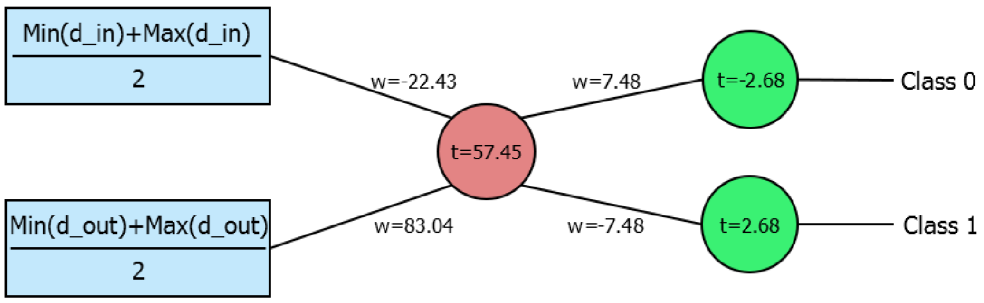


Figure 5.17: Multilayer perceptron after applying SimpleMI with averaged minimax

Since the neural network uses sigmoid activation function ($f(x) = \frac{1}{1+e^{-x}}$), the output value of the hidden neuron will be:

$$h_{out} = \frac{1}{1 + e^{-(-22.43in\_avg+83.04out\_avg+57.45)}}$$

where $in\_avg = \frac{Min(d\_in)+Max(d\_in)}{2}$ and $out\_avg = \frac{Min(d\_out)+Max(d\_out)}{2}$. The output values of the two output neurons are the following:

$$out\_class_0 = \frac{1}{1 + e^{-(7.48h_{out}-2.68)}}$$

$$out\_class_1 = \frac{1}{1 + e^{-(-7.48h_{out}+2.68)}}$$

The input values of the network (in_avg and out_avg) are classified based on which output value (out_class_0 or out_class_1) has greater value. If $out\_class_1 > out\_class_0$ then the inputs are classified as correct (class 1), otherwise they are classified as faulty (class 0). The distribution of the (in_avg, out_avg) value pairs between the two classes (class 0 and 1) are demonstrated on diagram 5.18.

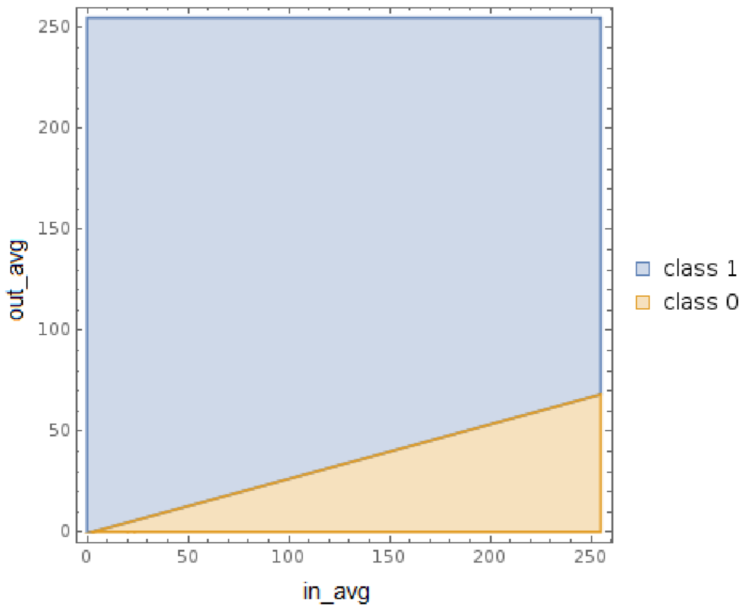


Figure 5.18: Distribution of the neural network's (in_avg, out_avg) value pairs between classes 0 and 1

For comparison, below are presented the structures of the other two, less successful neural networks. On figure 5.19 we can see the MLP trained after applying SimpleMI with *mean value selection* (see results on graph 5.14). Similarly to 5.17, it also has only a single neuron in its hidden layer. Figure 5.20 shows the MLP with two neurons in its hidden layer, trained on data set which was transformed by SimpleMI's *merged minimax* method (see results on diagram 5.16).

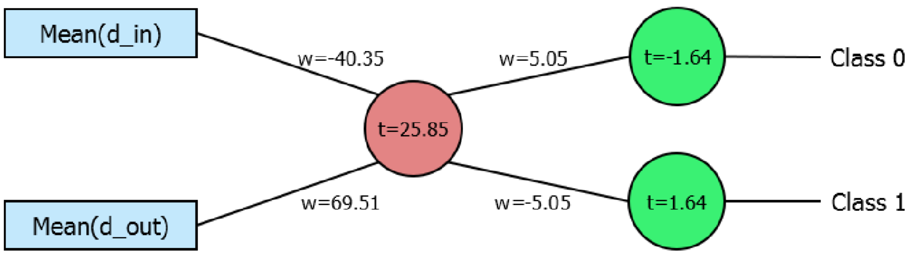


Figure 5.19: Multilayer perceptron after applying SimpleMI with mean value selection

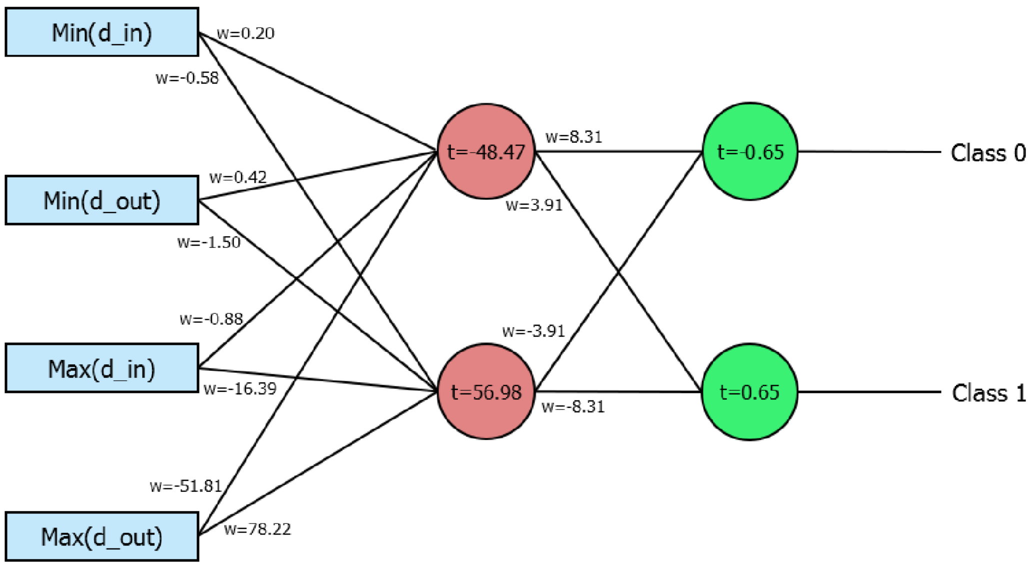


Figure 5.20: Multilayer perceptron after applying SimpleMI with merged minimax

5.5.4 RandomForest with SimpleMI

Similarly to MLP, RandomForest (RF) was also used with SimpleMI to measure the required number of trees for its best functioning. The depth of the trees in the RandomForest were not limited. SimpleMI was used with each of its transformation methods, as in the previously described MLP experiment (see 5.5.3).

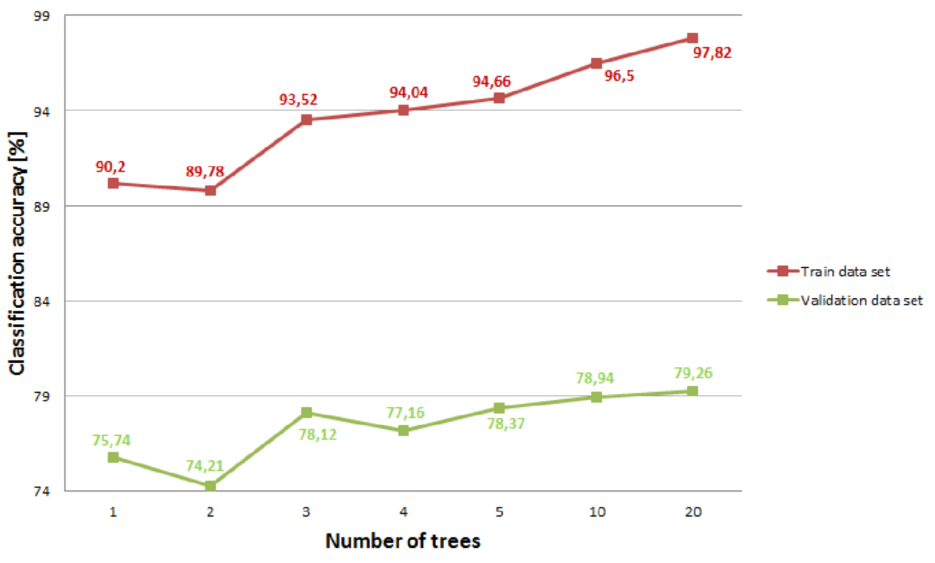


Figure 5.21: SimpleMI with mean value selection using RF as base classifier

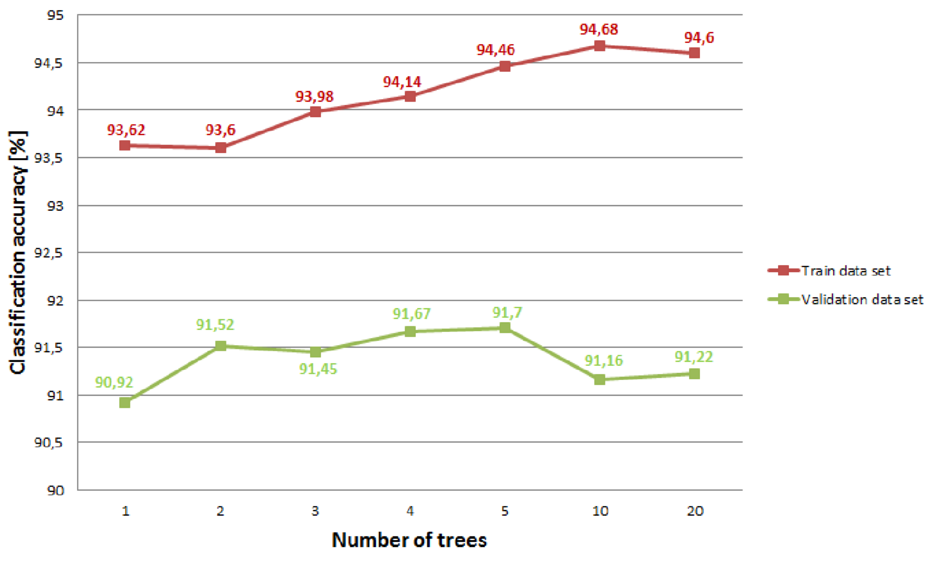


Figure 5.22: SimpleMI with averaged minimax using RF as base classifier

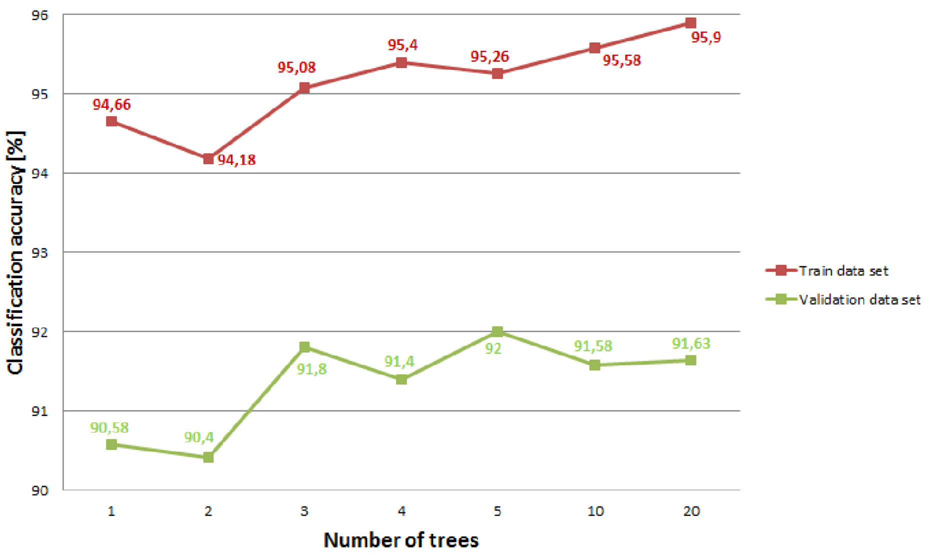


Figure 5.23: SimpleMI with merged minimax using RF as base classifier

The best classification accuracy was measured again with SimpleMI's *averaged minimax* transformation method. Graph 5.22 shows the best classification results with 2 trees of the RandomForest classifier. The out of bag error with 2 trees is 9.25% (equals to 90.75% accuracy) which is similar to the accuracy measured on the validation sets (91.52%). The size of the trees is: 543 and 553 nodes. The classifier's evaluation on the test data set resulted in **91.46%** classification accuracy.

5.6 Evaluation based on the significance of the injected faults

Since it is more important to detect faults which cause an audible dysfunction than detecting those which are merely audible, this experiment shows how the measured classification accuracy depends on the significance of the injected faults. Considering the previous experiment's results on the *fir_diff_io_vals* data set type (described in 5.4), this experiment was also performed on it. The train data set of this experiment consisted of 5000 instances in which the injected faults were not limited based on their ODG value (could also contain almost imperceptible faults). The classifiers were trained using the previously chosen parameters (see 5.5).

For each fault-type a separate test data set was generated based on the previously described parameters of fault injection: *err_sum*, *err_inv_bit*, *err_sample_len* (see 5.3). The individual types of faults were gained by combining the possible values of these parameters: Each resulting test data set consisted of 500 instances, containing only one of the fault-types. The classification accuracy was computed for each testing data set separately.

The individual fault-types (test data sets) were distributed to 5 intervals based on their induced ODG value: $< 0.0, -0.5)$, $< -0.5, -1.0)$, $< -1.0, -1.5)$, $< -1.5, -2.0)$ and $< -2.0, -4.0 >$. These intervals are shown on the x-axis of the below diagrams (see 5.24, 5.25 and 5.26). The classification results of the individual intervals are described by three bars:

- **Minimum:** The lowest classification accuracy of those which were computed for the given interval.

- **Median:** The median classification accuracy from the classification accuracies of the given interval.
- **Maximum:** The highest classification accuracy of those which were computed for the given interval.

Each bar has an information about the given fault-type injected to the test data set for which the bar's classification accuracy was computed. It is described in the top-part of the bars as a triplet: *err_sum*;*err_inv_bit*;*err_sample_len*.

In this experiment it was diagnosed that the faults injected with parameter value *err_sample_len* = 30, cannot be used for learning on the used data set type *fir_diff_io_vals* (see 5.4). Since the *fir_diff_io_vals* data set contains only 30 subsequent FIR input-output values in its bag attribute, parameter *err_sample_len* = 30 would mean generating all these I/O values with an injected fault when *class* = 0 is generated. However, the used algorithms of this experiment were not able to detect faults without the transition from the correct to faulty behaviour, therefore this parameter setting was excluded from the below results.

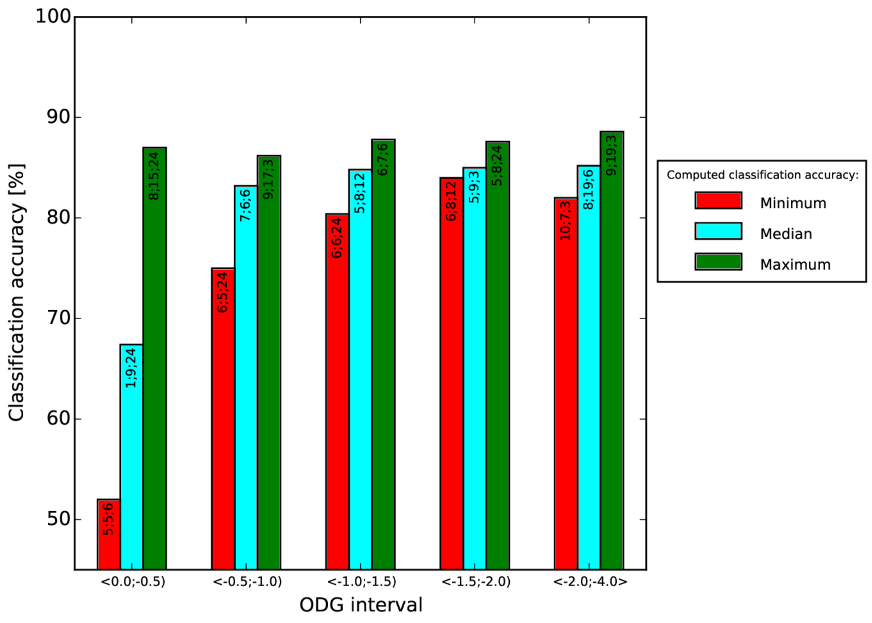


Figure 5.24: HMM classification accuracy by ODG intervals

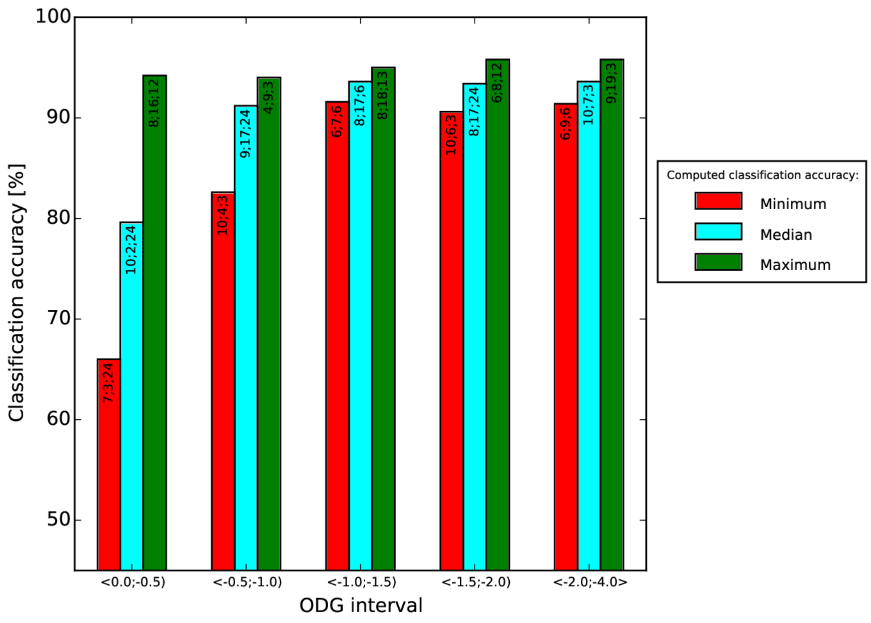


Figure 5.25: RF with SimpleMI classification accuracy by ODG intervals

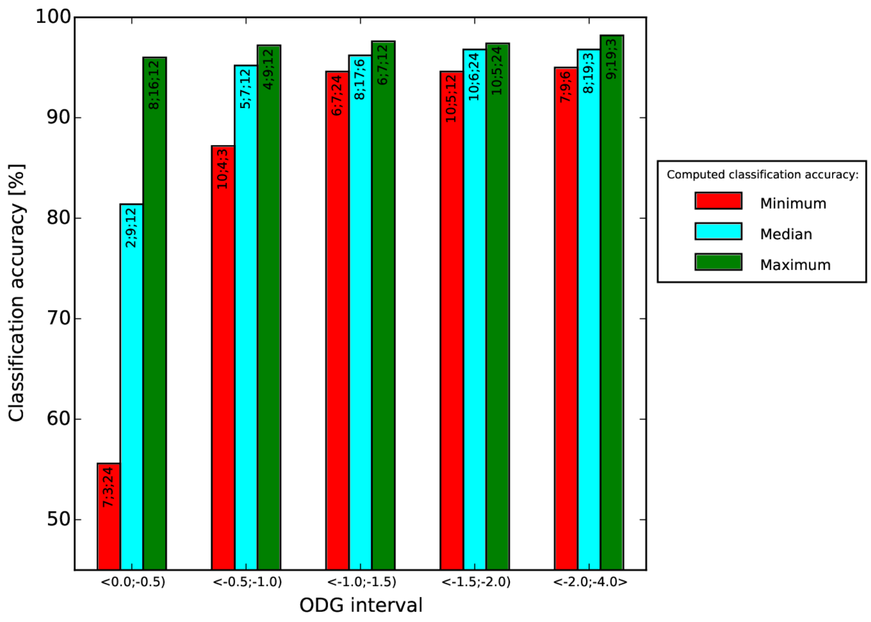


Figure 5.26: MLP with SimpleMI classification accuracy by ODG intervals

Despite that Hidden Markov Model is designed for learning from time series data, both RandomForest and MultilayerPerceptron classifiers produced better classification results using the SimpleMI transformation method. It can be seen that below the -1.0 ODG value of the injected faults the classifiers give more reliable results. The minimal classification accuracy in these cases is **above 90%** for both RF and MLP classifiers. Since the faults injected between ODG values 0.0 and -1.0 are almost imperceptible, the results on the lower ODG value intervals can be assessed positively.

A final experiment with the above evaluated classifiers was performed, now only with faults types which have less than -1.0 ODG value. The results show significant improvement compared to the previous ones (see 5.5), where fault types were not limited based on their ODG value (including fault types with ODG values from interval 0.0 to -1.0). The below results were measured on the testing data set (1500 instances). Hidden Markov model with 6 states was accurate in **92.8%** of the cases (before this 86.4%). SimpleMI combined with MLP (1 neuron in hidden layer) achieved **98.6%** classification results (previously 92.73%). SimpleMI with RF (2 trees) showed results with **97%** classification accuracy (previously 91.46%).

Chapter 6

Conclusion

The aim of the thesis was to propose a solution for increasing the dependability of digital systems. For this purpose fault tolerance with different types of redundancy was studied. The fundamental types of redundancy applicable in fault-tolerant systems were presented. The reader was acquainted with the basic principles of checking circuits applicable in fault-tolerant systems. As target platform for checking circuit development, the FPGA was proposed. The properties of existing checking circuit implementations available for FPGA were discussed.

The thesis focused on creating checking circuits with machine learning algorithms for hardware components with sequential logic. The theoretical background and the individual types of machine learning were presented. Hardware components with sequential logic required special types of machine learning algorithms which are able to process time series. The thesis distinguishes two types of machine learning algorithms for handling time series. The first type was presented as a group of machine learning algorithms capable of processing time series data directly. The second type included non-time series learning algorithms which can work with time series data only by cooperating with specific types of transformation methods.

The application of the individual machine learning algorithms was attained by employing the Weka data mining framework [13]. Besides the machine learning algorithms, Weka also has a specific file format for machine learning data sets. The reader is informed about both the utilized machine learning classifier implementations and the individual properties of the used data set types. The data sets are generated in two steps: simulation of the observed hardware component and post-processing of the simulation-generated files. These are described including the fault injection techniques applied during the simulation.

The experiments of the thesis were performed on a low-pass FIR filter, implemented by D. Pardo [35]. The main goal of the experiments was to determine which machine learning algorithms would be possible to use in checking circuits. For this purpose various machine learning classifiers were chosen and their classification accuracy was measured. In the first section of the experiments, the reader can be acquainted with the design of the used FIR filter. During the experiments the FIR filter was used for processing audio (`.wav`) files. After injecting faults to the FIR filter, the resulting audio file was usually damaged, but the severity of the damage was unknown. It was proposed to measure the difference between the damaged audio signals to be able to distinguish the severity of the injected faults. For this purpose the PEAQ algorithm [26] was used and also explained. The ensuing parts of the experiments describe the configuration of the injected faults and the content of data sets.

The main outcomes of the thesis are experimental results. The first experiments were performed to find the best settings for the tested machine learning classifiers. In this experiment (see section 5.5) 7 combinations of machine learning classifiers were used. The faults injected to the FIR filter were not limited based on the severity of the induced audio-damage. The results of each classifier were evaluated and discussed. The best classification accuracy was achieved by the SimpleMI wrapper algorithm using averaged minimax transformation combined with the MultilayerPerceptron classifier. Its classification was accurate in 92.73% of the cases, measured on the testing data set with 1500 instances. Considering that SimpleMI performs significant reduction of the original time series data (in this case from 60 attribute-values to 2), this result can be evaluated positively. On the other hand it also means that the complexity of the FIR filter does not require a more sophisticated learning algorithm.

The second group of experiments was focused also on the severity of the injected faults (see section 5.6). With this approach it was possible to distinguish different categories of faults, and hence ignore those which are almost imperceptible in the resulting audio signal. The results were evaluated and compared, after which it can be stated that ignoring almost imperceptible faults increases the classification accuracy of the previously evaluated algorithms in a significant way. The above described SimpleMI has 98.6% accuracy with the same settings.

The future work should focus on performing new experiments on more complex hardware components. Hardware components with more complex logic, or with more input and output values could also require the use of new machine learning algorithms. Another possible continuation of this thesis would be to implement one of the most successful classifiers in VHDL to FPGA. Classifiers RandomForest and MultilayerPerceptron with the SimpleMI wrapper algorithm achieved promising results, hence they could be integrated to a checking circuit. For measuring the quality of the online checker on FPGA a separate fault injection platform should be designed.

Bibliography

[1] Alpaydin, E.: *Introduction to Machine Learning*. Massachusetts Institute of Technology. 2010. iSBN 978-0-262-01243-0.

[2] Anderson, P. M.; Coyne, J. W.: A lightweight, high reliability, single battery power system for interplanetary spacecraft. In *Proceedings, IEEE Aerospace Conference*, vol. 5. 2002. pp. 5–2433–5–2444 vol.5.

[3] Avižienis, A.: Fault-Tolerant Systems. *IEEE Trans. Comput.*. vol. 25, no. 12. December 1976. ISSN 0018-9340.

[4] Avižienis, A.; Laprie, J.; Randell, B.: Fundamental Concepts of Dependability. 2001.

[5] Breiman, L.: Random Forests. *Machine Learning*. vol. 45, no. 1. 2001: pp. 5–32.

[6] Carneiro, G.; Chan, A. B.; Moreno, P. J.; et al.: Supervised Learning of Semantic Classes for Image Annotation and Retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. vol. 29, no. 3. March 2007: pp. 394–410. ISSN 0162-8828.

[7] Downes, P. T.: Applications of chaotic time series analysis to signal processing. In *[1992] Conference Record of the Twenty-Sixth Asilomar Conference on Signals, Systems Computers*. Oct 1992. ISSN 1058-6393. pp. 98–104 vol.1.

[8] Dubrova, E.: *Fault-Tolerant Design*. Springer Publishing Company, Incorporated. 2013. ISBN 1461421128, 9781461421122.

[9] Florescu, I.: *Probability and Stochastic Processes*. Wiley. 2014. ISBN 9780470624555.

[10] Frank, E. T.; Xu, X.: Applying propositional learning algorithms to multi-instance data. Technical report. University of Waikato. Department of Computer Science, University of Waikato, Hamilton, NZ. 06 2003.

[11] Fu, H.-C.; Chang, H.-Y.; Xu, Y. Y.; et al.: User adaptive handwriting recognition by self-growing probabilistic decision-based neural networks. *IEEE Transactions on Neural Networks*. vol. 11, no. 6. Nov 2000: pp. 1373–1384. ISSN 1045-9227.

[12] Gillies, M.: HMMWeka. <`http://doc.gold.ac.uk/~mas02mg/software/hmmweka/`>. accessed September 24, 2015.

[13] Hall, M.; Frank, E.; Holmes, G.; et al.: The WEKA Data Mining Software: An Update. *SIGKDD Explorations*. vol. 11, no. 1. 2009.

[14] Han, X.; Yang, B.; Lee, S.: Application of Random Forest Algorithm in Machine Fault Diagnosis. In *Engineering Asset Management*. Springer London. 2006. ISBN 978-1-84628-583-7. pp. 779–784.

[15] Herculano-Houzel, S.: The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in Human Neuroscience*. vol. 3. 2009: page 31. ISSN 1662-5161. Retrieved from:
<http://journal.frontiersin.org/article/10.3389/neuro.09.031.2009>

[16] Hüsken, M.; Stagge, P.: Recurrent neural networks for time series classification. *Neurocomputing*. vol. 50. 2003: pp. 223 – 235. ISSN 0925-2312.

[17] ITU Radiocommunication Assembly: RECOMMENDATION ITU-R BS.1387-1 - Method for objective measurements of perceived audio quality. ITU. 2001.

[18] Jeong, Y.-S.; Jeong, M. K.; Omitaomu, O. A.: Weighted dynamic time warping for time series classification. *Pattern Recognition*. vol. 44, no. 9. 2011: pp. 2231 – 2240. ISSN 0031-3203.

[19] Kaimin, W.; Xiaofei, N.; Yumei, C.; et al.: DTSP-V: A trend-based Top Scoring Pairs method for classification of time series gene expression data. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. Dec 2016. pp. 1787–1794.

[20] Kampouraki, A.; Manis, G.; Nikou, C.: Heartbeat Time Series Classification With Support Vector Machines. *IEEE Transactions on Information Technology in Biomedicine*. vol. 13, no. 4. July 2009: pp. 512–518. ISSN 1089-7771.

[21] Kibbe, H.: Finite Impulse Response Filters Using Apple's Accelerate Framework - Part I. <http://hamiltonkibbe.com/finite-impulse-response-filters-using-apples-accelerate-framework-part-i/>. 2014.

[22] Kirkby, R.: *RandomForest*. University of Waikato. accessed: 2017-04-09.

[23] Koren, I.; Krishna, C. M.: *Fault-Tolerant Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.. 2007. ISBN 0120885255, 9780080492681.

[24] Kröse, B.; van der Smagt, P.: *An Introduction to Neural Networks*. University of Amsterdam. 1996.

[25] Lala, P. K.: Transient and Permanent Fault Injection in VHDL Description of Digital Circuits. 2012.

[26] M. Holters, U. Zölzer: GstPEAQ - an Open Source Implementation of the PEAQ Algorithm. *Proceedings of the 18th International Conference on Digital Audio Effects (DAFx'15)*. 2015. trondheim, Norway.

[27] Matousek, K.: Security and reliability considerations for distributed healthcare systems. In *2008 42nd Annual IEEE International Carnahan Conference on Security Technology*. Oct 2008. ISSN 1071-6572. pp. 346–348.

[28] Matušová, L.; Kaštil, J.; Kotásek, Z.: Automatic Construction of On-line Checking Circuits Based on Finite Automata. In *17th Euromicro Conference on Digital Systems Design*. IEEE Computer Society. 2014. ISBN 978-0-7695-5074-9. pp. 326–332. Retrieved from:
<http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10734>

[29] Mikolov, T.; Karafiát, M.; Burget, L.; et al.: Recurrent neural network based language model. In *Interspeech*, vol. 2. 2010. page 3.

[30] Mishu, S. Z.; Rafiuddin, S. M.: Performance analysis of supervised machine learning algorithms for text classification. In *2016 19th International Conference on Computer and Information Technology (ICCIT)*. Dec 2016. pp. 409–413.

[31] Mitchell, T. M.: *Machine Learning*. McGraw-Hill Science/Engineering/Math. 1997. iSBN 0-07-042807-7.

[32] Murphy, K. P.: *Machine Learning: A Probabilistic Perspective*. Massachusetts Institute of Technology. 2012. iSBN 978-0-262-01802-9.

[33] Nilsson, N. J.: Introduction to Machine Learning: An Early Draft of a Proposed Textbook. `<http://robotics.stanford.edu/people/nilsson/mlbook.html>`. 1998.

[34] Nogales, F. J.; Contreras, J.; Conejo, A. J.; et al.: Forecasting next-day electricity prices by time series models. *IEEE Transactions on Power Systems*. vol. 17, no. 2. May 2002: pp. 342–348. ISSN 0885-8950.

[35] Pardo, D.: VHDL Parametrizable FIR Filter. `<http://opencores.org/project,fir_filter>`. 2013.

[36] Quinlan, J. R.: Induction of Decision Trees. *Machine Learning*. vol. 1, no. 1. March 1986: pp. 81–106. ISSN 0885-6125.

[37] Quinlan, J. R.: *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc.. 1993. ISBN 1-55860-238-0.

[38] Rabiner, L. R.: A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*. vol. 77, no. 2. 1989: pp. 257–286.

[39] Rabiner, L. R.; Juang, B. H.: An introduction to hidden Markov models. *IEEE ASSP Magazine*. January 1986: pp. 4–15.

[40] Riedmiller, P. D. M.: Machine Learning: Multi Layer Perceptrons.

[41] Rokach, L.: Ensemble-based Classifiers. *Artif. Intell. Rev.*. vol. 33, no. 1-2. February 2010: pp. 1–39. ISSN 0269-2821.

[42] Rubino, G.; Sericola, B.: *Markov Chains and Dependability Theory*. EBL-Schweitzer. Cambridge University Press. 2014. ISBN 9781107007574.

[43] Sigaud, O.; Buffet, O.: *Markov Decision Processes in Artificial Intelligence*. Wiley. 2010. ISBN 9781848211674.

[44] Straka, M.: Aplikace hlídacích obvodů v architekturách odolných proti poruchám. In *Počítačové architektury a diagnostika 2008*. Liberec University of Technology. 2008. ISBN 978-80-7372-378-1. pp. 97–102.
Retrieved from: `<http://www.fit.vutbr.cz/research/view_pub.php.cs?id=8698>`

[45] Straka, M.; Kotásek, Z.; Winter, J.: The Design of Hardware Checkers for Verification and Diagnostic Purposes. In *CSE'2008 International Scientific Conference on Computer Science and Engineering*. The University of Technology Košice. 2008. ISBN 978-80-8086-092-9. pp. 320–327.
Retrieved from: `<http://www.fit.vutbr.cz/research/view_pub.php.cz?id=8646>`

[46] Straka, M.; Kotásek, Z.; Winter, J.: Digital Systems Architectures Based on On-line Checkers. In *11th EUROMICRO Conference on Digital System Design DSD 2008*. IEEE Computer Society. 2008. ISBN 978-0-7695-3277-6. pp. 81–87.
Retrieved from: `<http://www.fit.vutbr.cz/research/view_pub.php.cs?id=8621>`

[47] Taylor, J. W.; McSharry, P. E.; Buizza, R.: Wind Power Density Forecasting Using Ensemble Predictions and Time Series Models. *IEEE Transactions on Energy Conversion*. vol. 24, no. 3. Sept 2009: pp. 775–782. ISSN 0885-8969.

[48] Tveter, D. R.: The Backprop Algorithm, Chapter 2. 2001.
Retrieved from:
`<http://www.cim.mcgill.ca/~jer/courses/ai/readings/backprop.pdf>`

[49] Udantha, M.: Markov Models and Hidden Markov Models.
`<http://madhukaudantha.blogspot.cz/2014/05/markov-models-and-hidden-markov-models.html>`. accessed September 24, 2015.

[50] Vyas, T.; Prajapati, P.; Gadhwal, S.: A survey and evaluation of supervised machine learning techniques for spam e-mail filtering. In *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. March 2015. pp. 1–7.

[51] Ware, M.: *MultilayerPerceptron*. University of Waikato. accessed: 2017-04-09.

[52] Yeh, Y.: Dependability of the 777 Primary Flight Control System. In *Dependable Computing for Critcal Applications 5*. IEEE Computer Society Press. 1998. pp. 3–17.

[53] Zhang, D.; Mabu, S.; Wen, F.; et al.: A supervised learning framework for PCA-based face recognition using GNP fuzzy data mining. In *2011 IEEE International Conference on Systems, Man, and Cybernetics*. Oct 2011. ISSN 1062-922X. pp. 516–520.

Appendices

Appendix A

Contents of the attached DVD

Below is presented the simplified directory structure of the attached DVD. In the file User manual.doc is described how the experiments can be executed. It also contains a more accurate description of the directory structure of the experiments.

```
/
├── Experiments ........... Directory containing the source codes of the experiments.
│   ├── Sim_DatasetGen
│   │   ├── CalcODG_of_FaultTypes
│   │   ├── GenDataset_for_each_FaultType
│   │   └── GenDatasets_with_DecreasingODGInterval
│   ├── Weka_Experiments
│   │   ├── configs
│   │   ├── dataset_gens
│   │   ├── graph_gens
│   │   ├── results
│   │   └── weka
│   └── User manual.doc
└── Text ................ Directory containing this technical text in electronic format.
    ├── source...Editable LaTeX documentation sources.
    └── xlelke00.pdf
```

Appendix B

Error detection on a FIR filter

This chapter contains additional graphs and results which were not included in the above experiments chapter (see 5) in order to keep the description of the results transparent.

B.1 Additional fault configuration results

All the injected fault types and their induced ODG value can be seen in this section. The below figures show the fault types on the individual adders of the FIR filter.

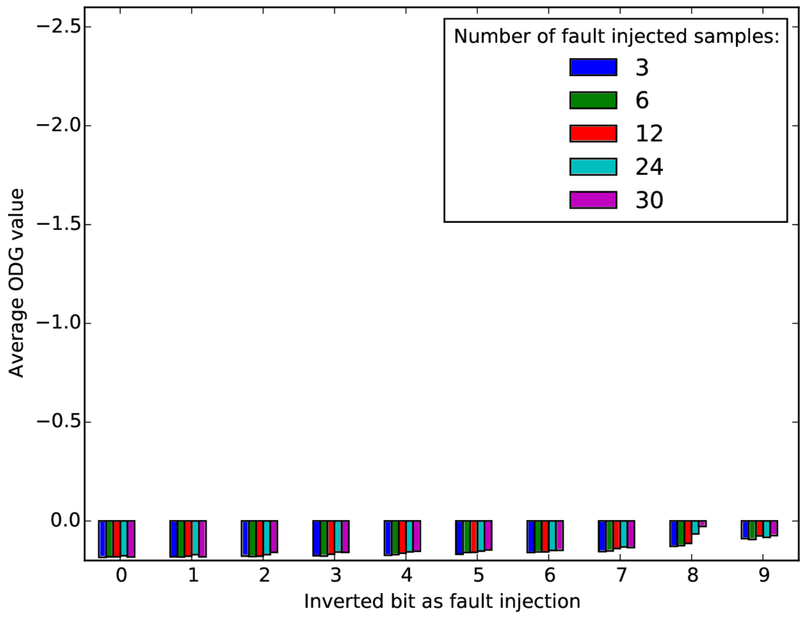


Figure B.1: Faults on adder $\sum_0$

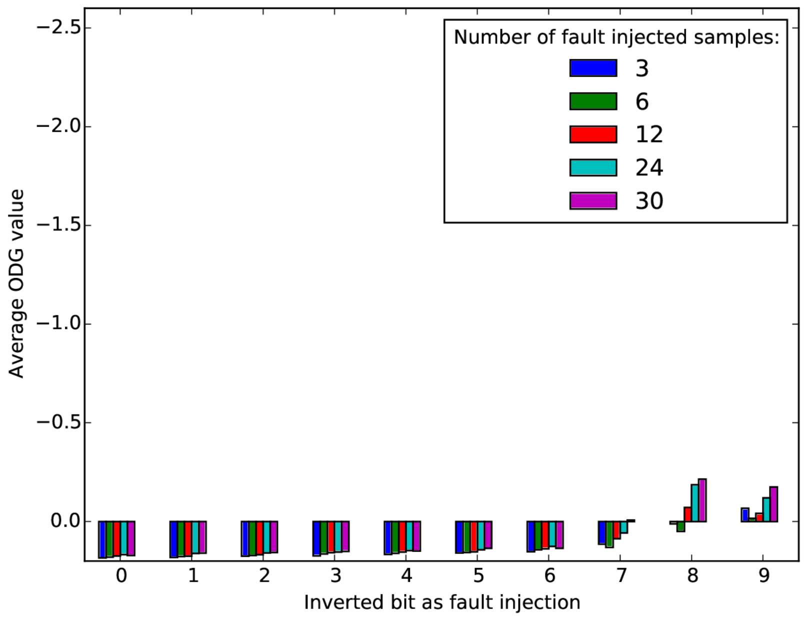


Figure B.2: Faults on adder $\sum_1$

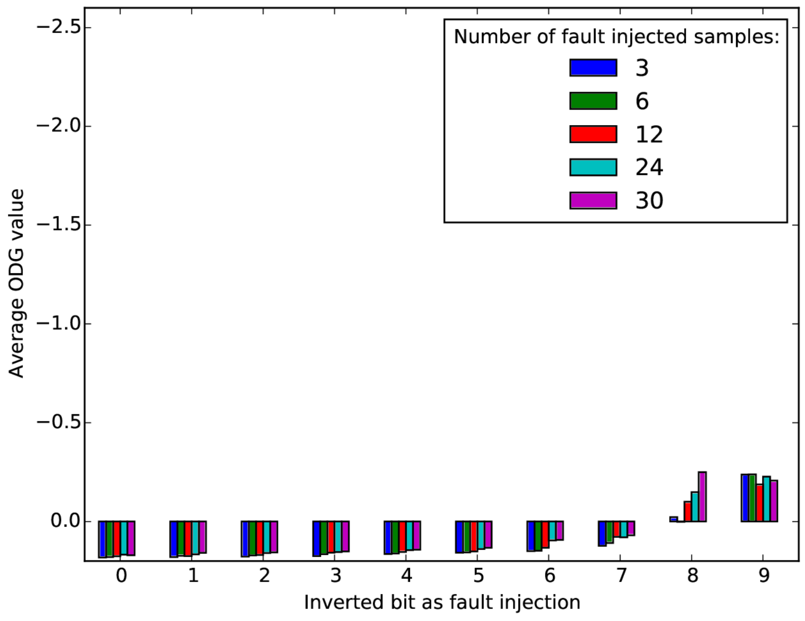


Figure B.3: Faults on adder $\sum_2$

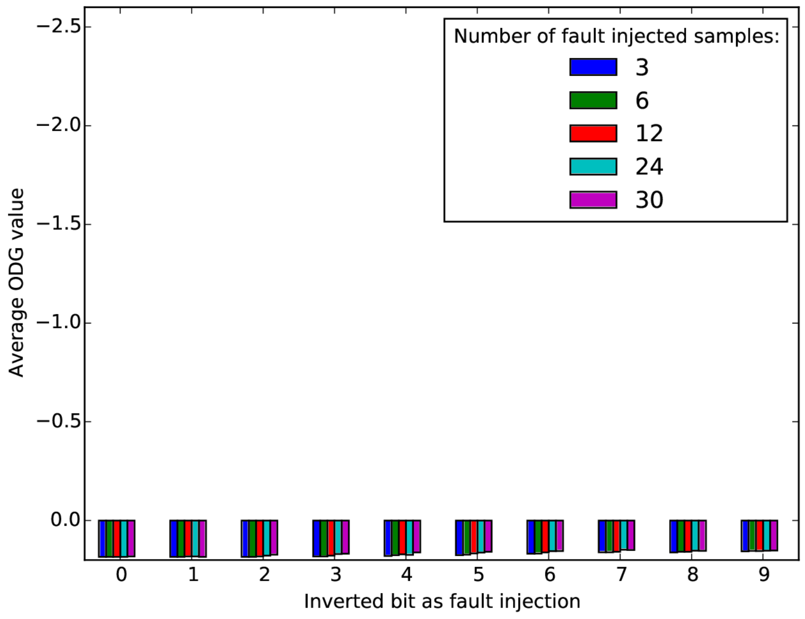


Figure B.4: Faults on adder $\sum_3$

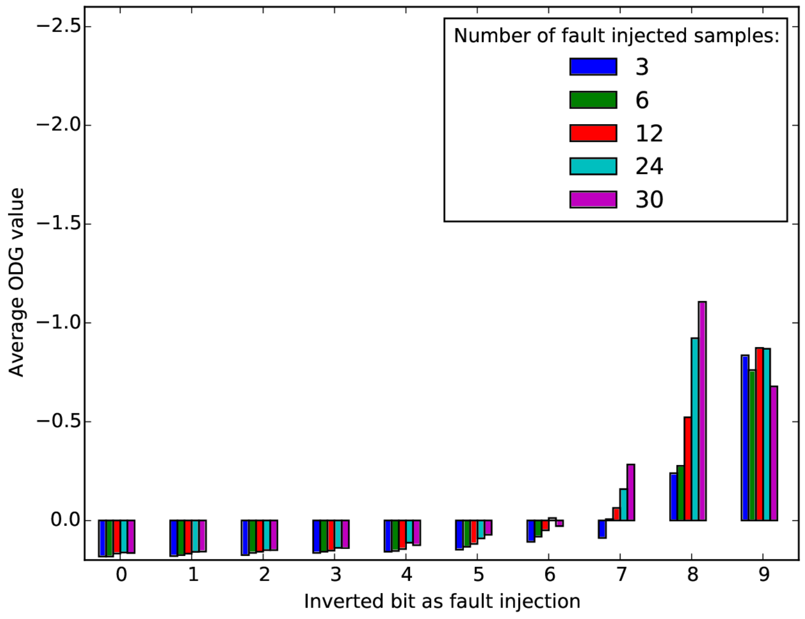


Figure B.5: Faults on adder $\sum_4$

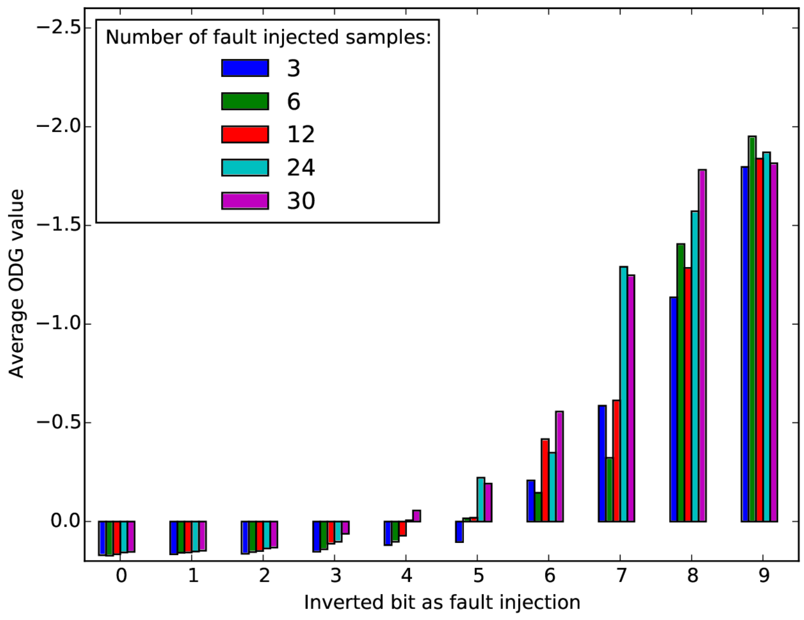


Figure B.6: Faults on adder $\sum_5$

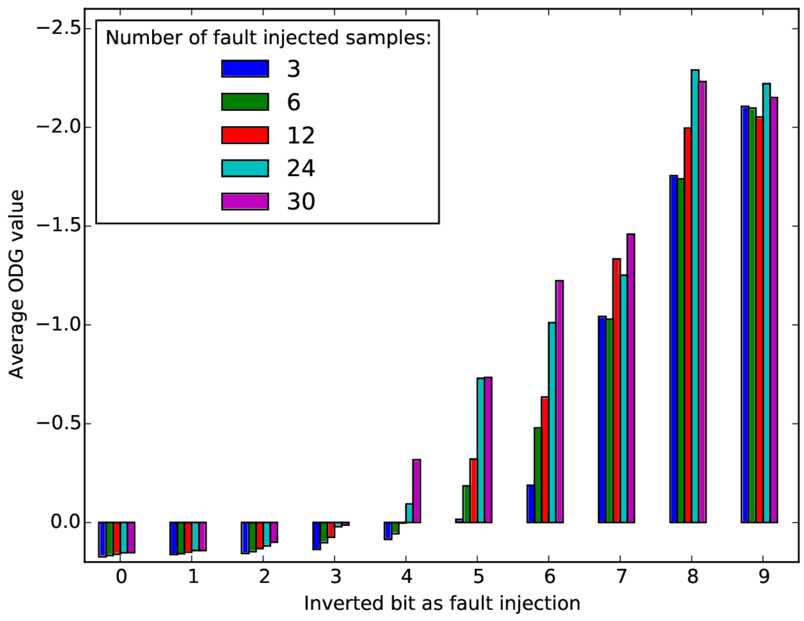


Figure B.7: Faults on adder $\sum_6$

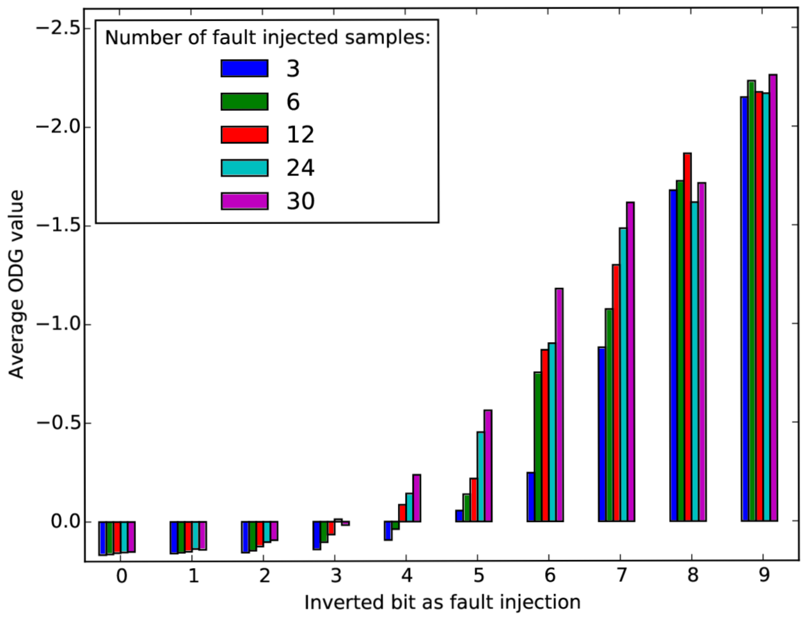


Figure B.8: Faults on adder $\sum_7$

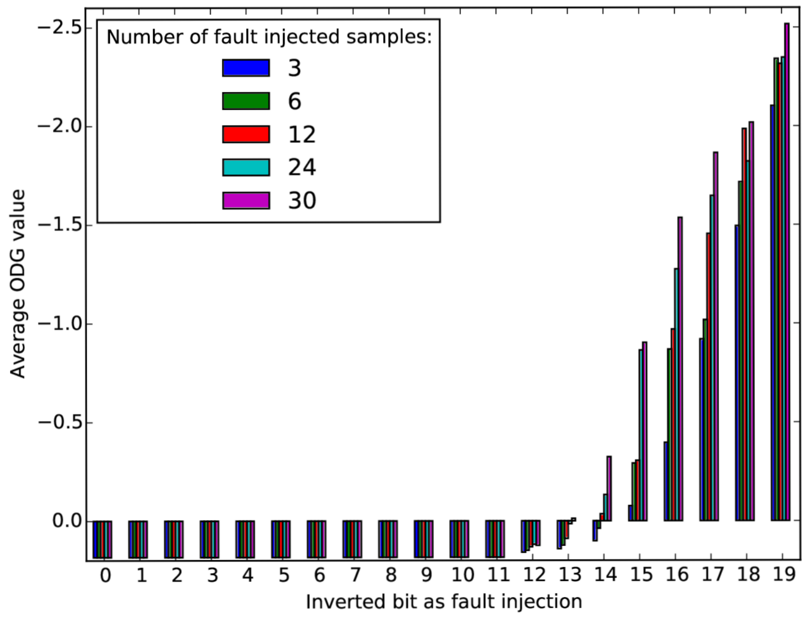


Figure B.9: Faults on adder $\sum_8$

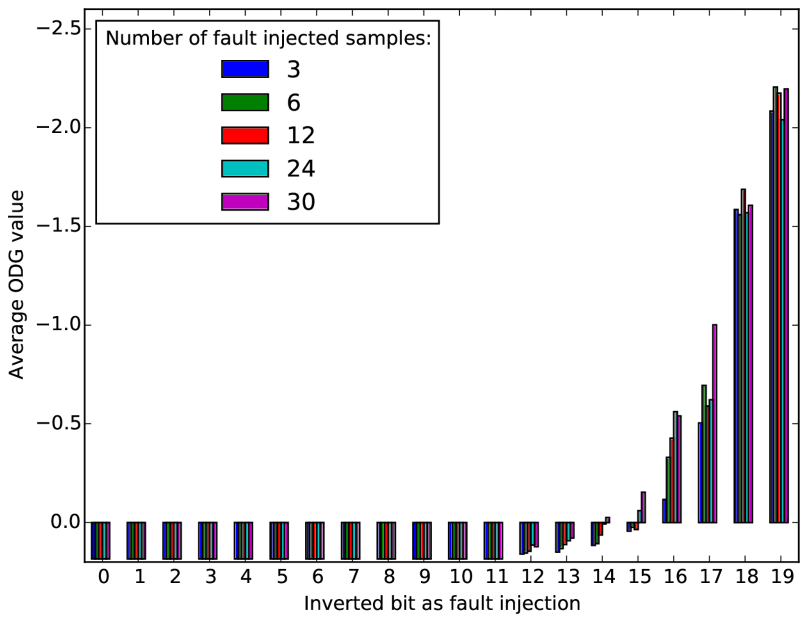


Figure B.10: Faults on adder $\sum_9$

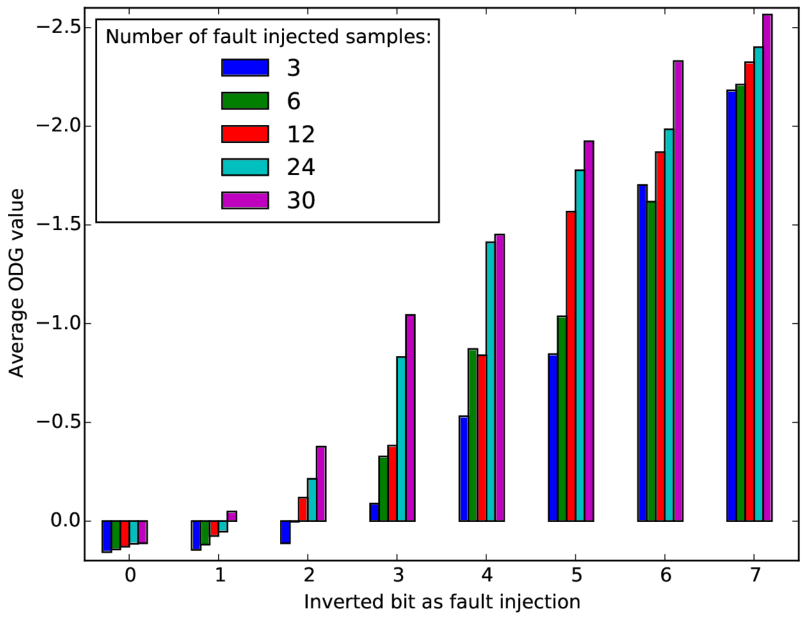


Figure B.11: Faults on adder $\sum_{10}$

B.2 Classification results on the *fir_30_io_vals* data set

The below graphs show the classification results on the *fir_30_io_vals* data set (see 5.4). The rest of the classifiers' parameters are specified in section 5.5.

B.2.1 Hidden Markov model

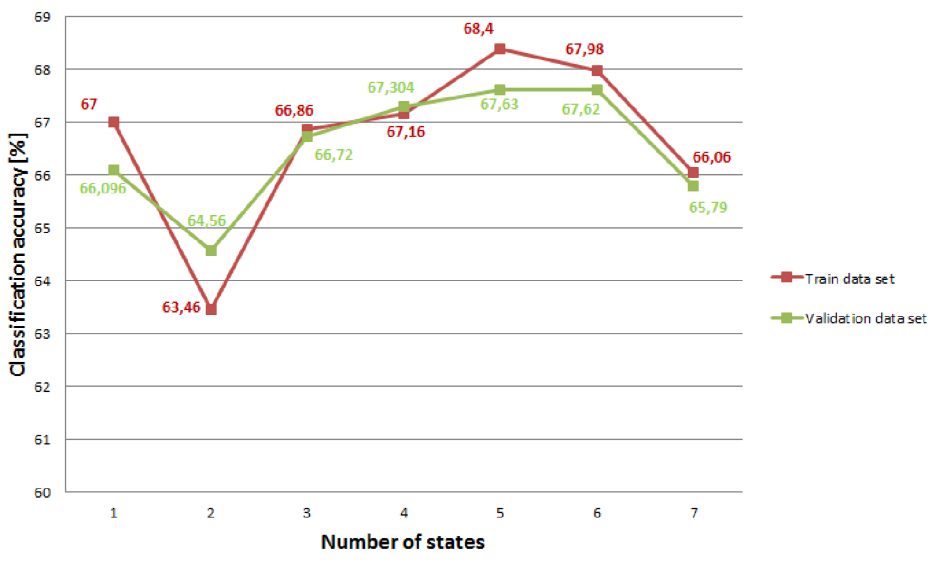


Figure B.12: HMM classification results by the number of states

B.2.2 RandomForest with transformation methods

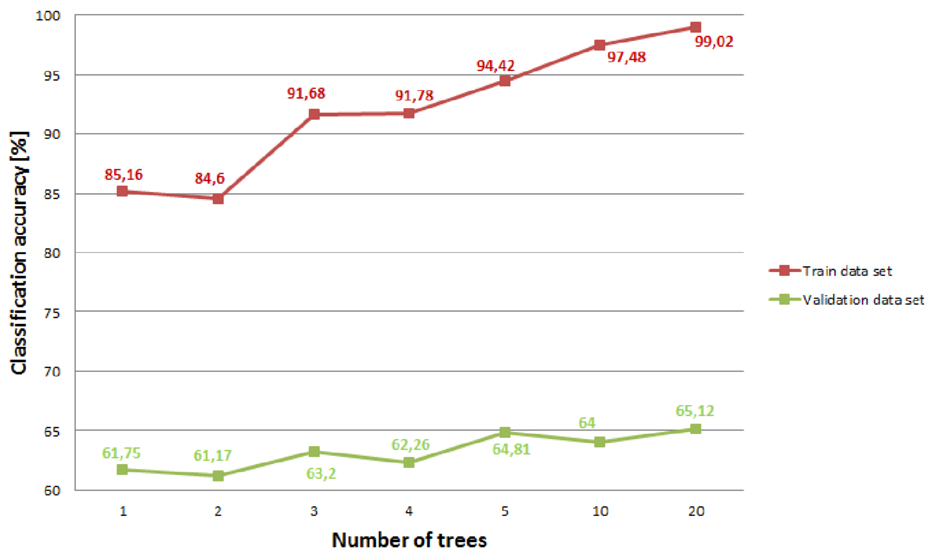


Figure B.13: SimpleMI with mean value selection using RF as base classifier

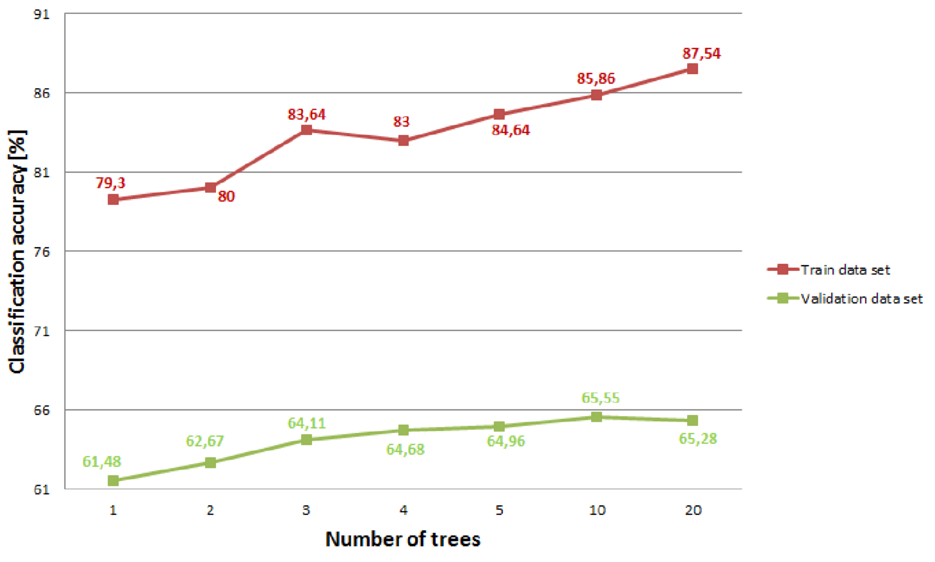


Figure B.14: SimpleMI with averaged minimax using RF as base classifier

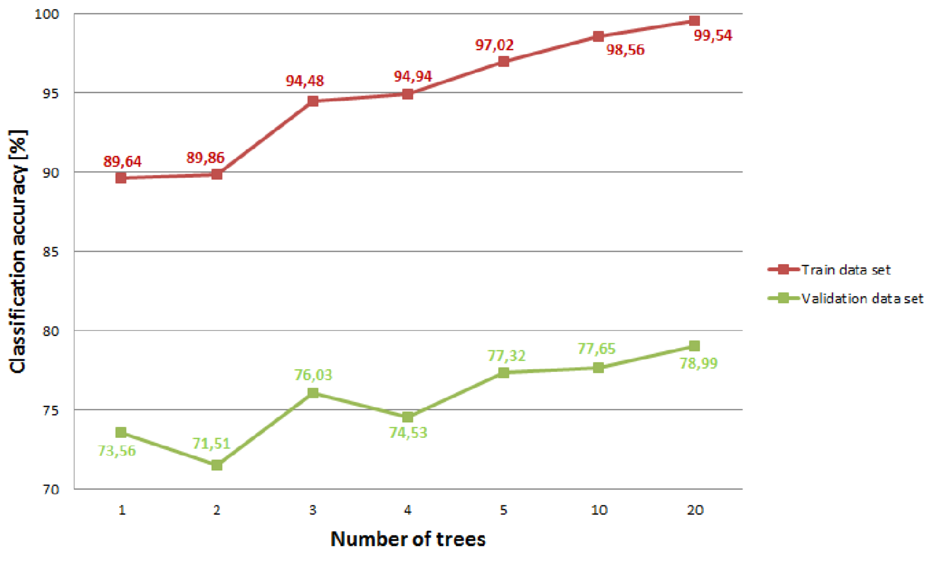


Figure B.15: SimpleMI with merged minimax using RF as base classifier

B.2.3 MultilayerPerceptron with transformation methods

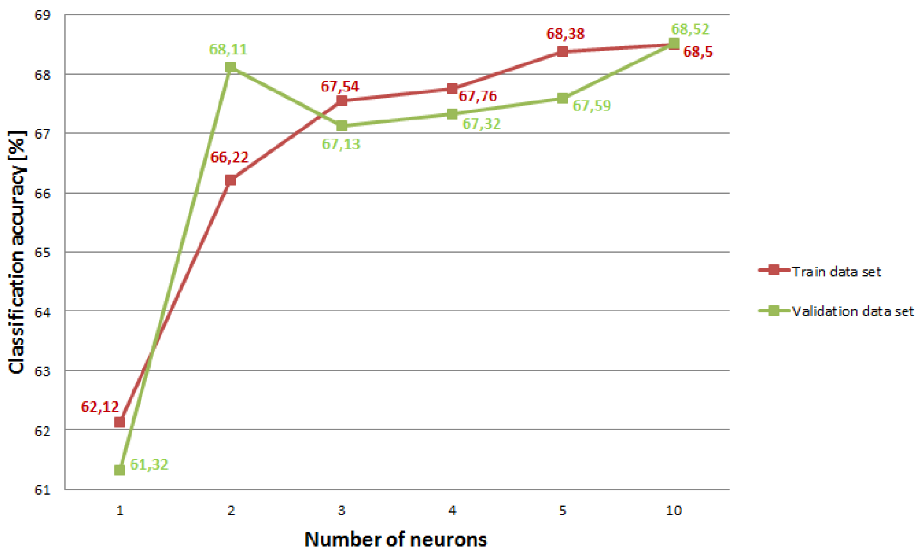


Figure B.16: SimpleMI with mean value selection using MLP as base classifier

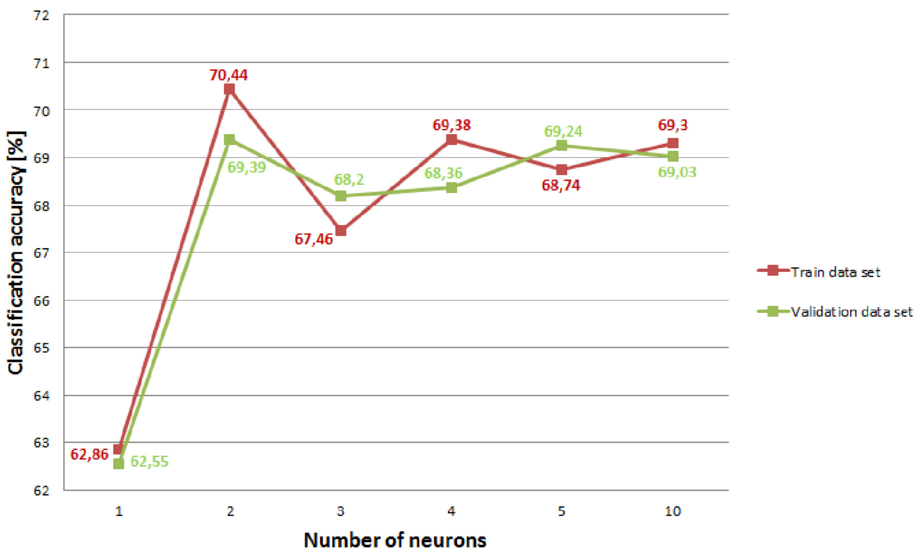


Figure B.17: SimpleMI with averaged minimax using MLP as base classifier

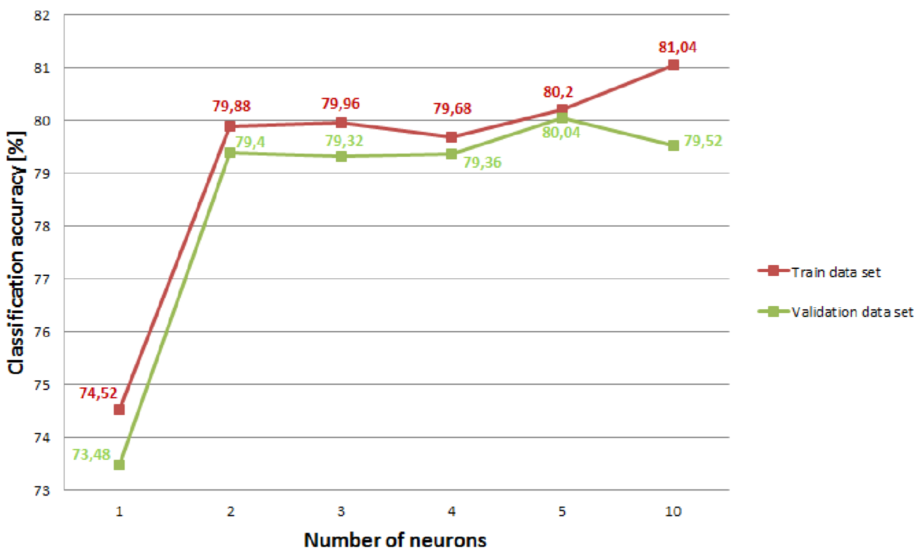


Figure B.18: SimpleMI with merged minimax using MLP as base classifier

B.3 State tranisition diagrams of HMMs trained on the *fir_diff_io_vals* data set

In this appendix section are presented two state transition diagrams gained by training of hidden Markov models on data set format *fir_diff_io_vals* in experiment 5.5. The diagrams were excluded from the experiments section of the thesis because their size is too big, and because presenting them in table form is more transparent. For their generation the *graphviz tool*[1] was used.

[1] <http://www.graphviz.org>

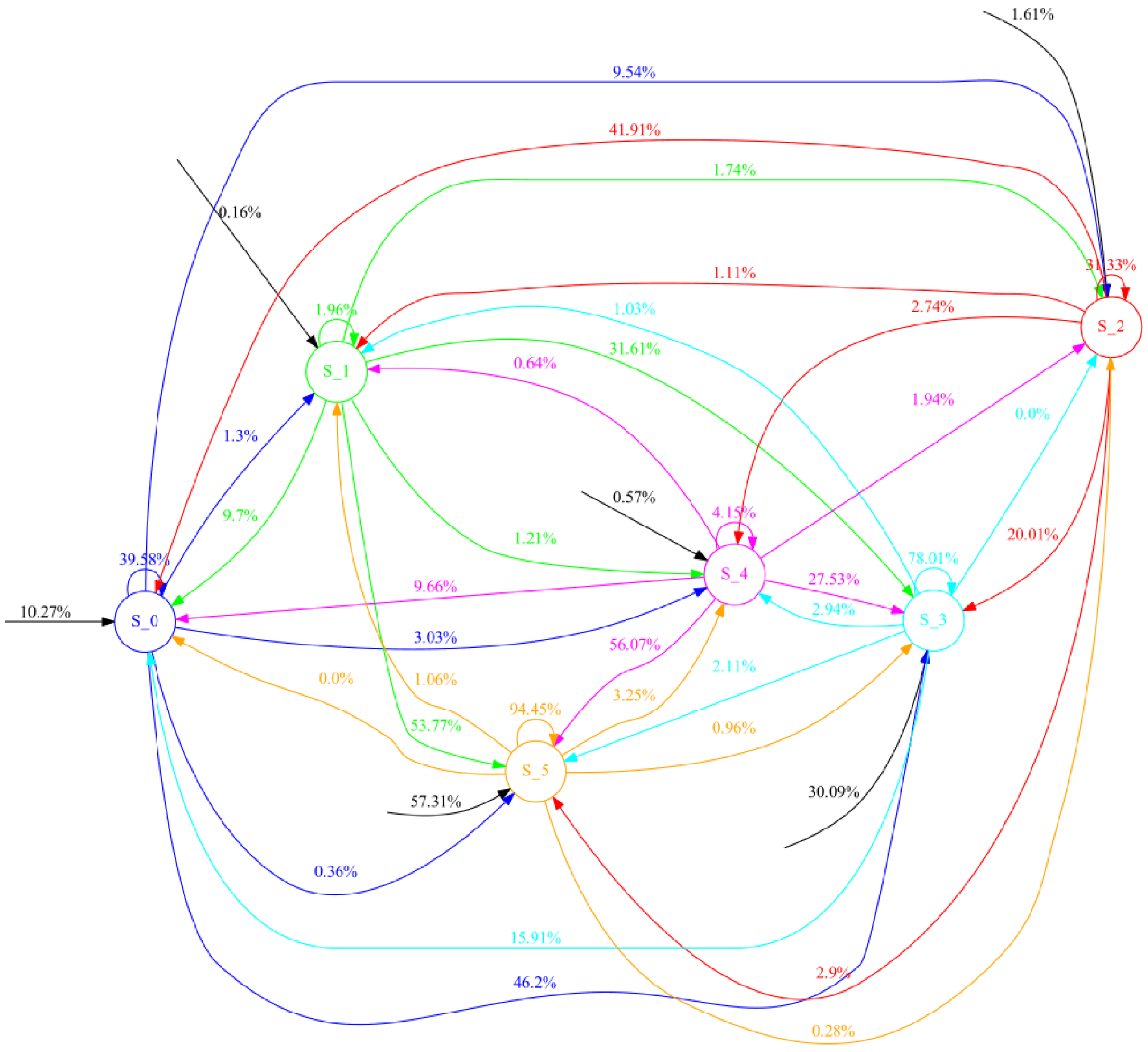


Figure B.19: State transition diagram of HMM with class 0

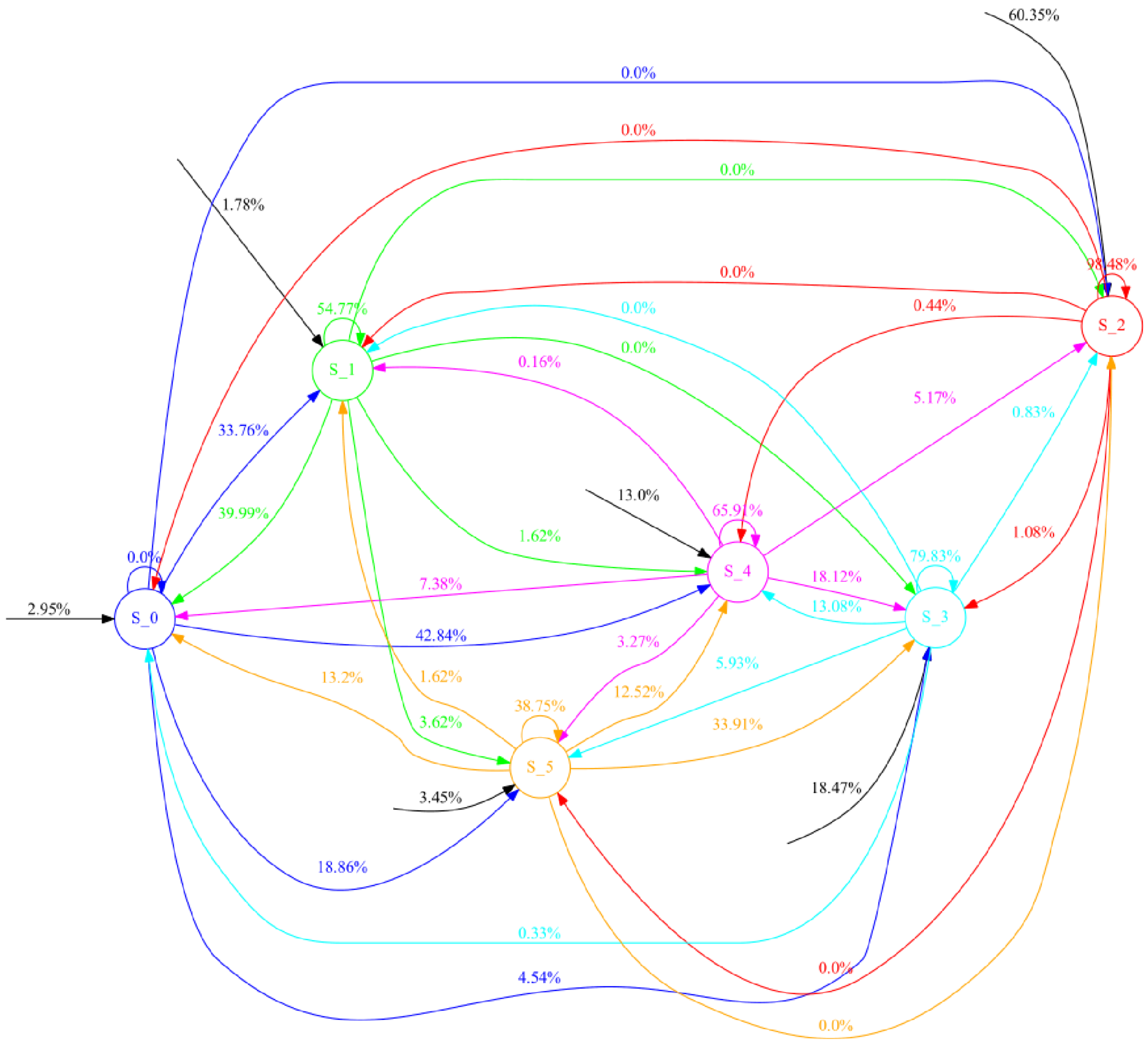


Figure B.20: State transition diagram of HMM with class 1