

# Machine Learning on Microcontrollers

Lorenz Graf



BACHELORARBEIT

Nr. 1510237007

eingereicht am  
Fachhochschul-Bachelorstudiengang

Mobile Computing  
in Hagenberg

im Juni 2017

This thesis was created as part of the course

## Hardwarenahe Programmierung

during

Summer Semester 2017

Advisor:

FH-Prof. DI Stephan Selinger

# Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 23, 2017

Lorenz Graf

# Contents

<b>Declaration</b>	<b>iii</b>
<b>Abstract</b>	<b>vi</b>
<b>Kurzfassung</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem statement . . . . .	1
1.3 Outline . . . . .	1
<b>2 Machine Learning Basics</b>	<b>2</b>
2.1 Classification . . . . .	2
2.2 Iris . . . . .	2
2.3 Anaconda . . . . .	2
<b>3 C++/C Machine Learning Libraries</b>	<b>8</b>
3.1 Shark - Machine Learning Library . . . . .	8
3.2 Alternative Approach . . . . .	9
<b>4 Decision Tree ID3</b>	<b>10</b>
4.1 Functionality . . . . .	10
4.2 Implementation . . . . .	10
4.3 Conclusion . . . . .	10
<b>5 Decision Tree J48</b>	<b>12</b>
5.1 Weka . . . . .	12
5.2 Preprocess Setup . . . . .	12
5.3 Classification . . . . .	12
<b>6 Process Generated Tree</b>	<b>14</b>
6.1 W2C Converter . . . . .	14
6.1.1 Structure . . . . .	14
6.1.2 Node . . . . .	15
6.1.3 Main . . . . .	15
6.2 W2C Converter Usage . . . . .	16
6.3 Function Based C Code . . . . .	17

6.4	Node Based C Code . . . . .	17
6.5	Additional Features . . . . .	18
6.5.1	MSP432 Ready . . . . .	18
6.5.2	Result Table . . . . .	18
<b>7</b>	<b>Classification on MCU</b>	<b>20</b>
7.1	Code Composer Studio . . . . .	20
7.1.1	Project Setup . . . . .	20
7.1.2	Configure Project . . . . .	20
7.1.3	Implementing C Code . . . . .	21
7.1.4	Classification Data . . . . .	21
7.2	Evaluating Results . . . . .	21
<b>8</b>	<b>Conclusion</b>	<b>22</b>
<b>A</b>	<b>Source Code</b>	<b>23</b>
A.1	Python Analytics Code . . . . .	23
A.2	Generated Function Based C Code . . . . .	25
A.3	Generated Node Based C Code . . . . .	26
	<b>References</b>	<b>28</b>
	Literature . . . . .	28
	Films and audio-visual media . . . . .	28
	Online sources . . . . .	28

# Abstract

This thesis deals with the topic of implementing basic machine learning, explains the problems that may occur and presents a solution to fix said problems.

Even though this thesis and its contents are targeted at a specific MCU it should be noted that all of its contents can be translated to any MCU.

One of the biggest problems is the way that current machine learning libraries behave, as they for example utilize dynamic memory allocation. These problems and their origins are also analyzed in detail.

The earlier mentioned problems are prevented by using a java program that was developed as a part of this thesis. It converts textual description of a specific machine learning process and parses it into c code. This code can then be used on any MCU that supports this language. In addition to solving the problems that occur when performing machine learning on MCUs the java software provides two different solutions to solving the given machine learning problem. One of these uses mainly flash memory, the other one ram. By doing so the more restricted resources availability can be maximized.

To prove the concept of the java application it was tested on one of the most common machine learning test sets that is analyzed in detail beforehand. The process is explained step by step so that the reader can reproduce the entire process.

# Kurzfassung

Diese Arbeit befasst sich mit der Implementierung grundlegender Machine Learning Algorithmen, zeigt die Probleme, die dabei auftreten können und stellt eine mögliche Lösung vor, die diesen Probleme vorbeugt.

Obwohl sich diese Arbeit an einer bestimmten MCU orientiert, kann das Konzept auf jegliche MCUs übertragen werden.

Eines der größten Probleme ist die Art der Implementierung, die von Machine Learning Libraries vorgewiesen wird. Dabei stellt vor allem die Benützung von dynamischer Speicherallocierung ein Problem dar. Dieses und ähnliche Probleme werden in der Arbeit genau analysiert.

Den zuvor genannten Problemen wird hier durch die Verwendung eines Java Programms vorgebeugt, das als Teil dieser Arbeit entwickelt wurde. Es konvertiert eine textuelle Beschreibung eines bestimmten Machine Learning Prozesses und wandelt diese in C Code um. Dieser Code kann daraufhin auf sämtlichen MCUs verwendet werden, die diese Programmiersprache unterstützen. Zusätzlich zur Lösung der Probleme, die bei der Implementierung von Machine Learning auf MCUs auftreten können, stellt das Java Programm zwei verschiedene Lösungen zur Verfügung. Eine davon verwendet in erster Linie den statischen Flashspeicher, die andere hauptsächlich den RAM der MCU. Anhand den beiden Implementierungen kann die Verfügbarkeit der am stärksten begrenzten Resource auf der MCU maximiert werden.

Um das Konzept zu belegen, wurde die Java Applikation anhand einer der meist verwendeten Machine Learning Datensätze, der in dieser Arbeit zuvor genau analysiert wird, getestet. Der Prozess wird Schritt für Schritt erklärt, so dass die Ergebnisse der Arbeit reproduziert werden können.

# Chapter 1

## Introduction

### 1.1 Motivation

This bachelor thesis answers the question if basic machine learning algorithms can be run on an MCU within reasonable effort and how much sense it makes to do so. What are the advantages and disadvantages? What problems must be faced when planning to implement machine learning on an MCU?

### 1.2 Problem statement

The characteristics of an MCU are that it is a tiny computer consisting of a single integrated circuit. Even though these SoCs (systems on a chip) are very compact they might still be utilizing multiple different modules, such as multiple cores for example. Hence the resources that such a device can provide are very limited which does more often than not conflict with the very resource heavy process of machine learning.

### 1.3 Outline

This thesis will tackle the possible risks of running machine learning code on such limited resource environments and explains how to still perform such tasks in several different ways. The two main approaches explained in this document will be to limit the code to use nearly explicitly either random access memory or flash storage. By doing so the required resources can be prioritized around the embedding code of the machine learning process.

In detail this thesis is about decision trees, more specifically ID3 (chapter 4) and J48 (chapter 5). The reason for this is that they are simple and easy to understand and to follow their classification process. However the contents of this document can be projected to any machine learning algorithm.

The target MCU that this thesis will focus on is the MSP432P401R[4] in combination with the Integrated Development Environment (IDE) Code Composer Studio[8].



## Chapter 2

# Machine Learning Basics

### 2.1 Classification

The target of this machine learning process is to classify given characteristics to one of multiple types. This includes calculating the weight that the different characteristics have on the outcome of the classification and if a characteristic is even used to classify the object in the first place.

There are many possible ways to perform machine learning and the following process of classification, but this thesis will be concentrated around the ID3 and J48 tree process. These two ways of machine learning and classification are described further down the line (chapter 4 and 5).

### 2.2 Iris

The iris dataset is one of the most used machine learning datasets and therefore a good example set for the purpose of testing machine learning on a micro controller. The set provides information about three different types of flowers (iris setosa, iris versicolour and iris virginica) and their four characteristics (sepal length, sepal width, petal length and petal width). The dataset is analysed in more detail at 2.3. In this case the dataset consists of a total of 150 data lines which are divided equally by three for the classes. Hence every class is described by 50 characteristic data lines.

The iris dataset that is used is provided by the UCI [2] machine learning database and downloaded from the Weka [7] homepage since it is already prepared to be used with the Weka tool which is used in combination with the 2.3 python library to analyse the given dataset.

### 2.3 Anaconda

*Anaconda is the leading open data science platform powered by Python. The open source version of Anaconda is a high performance distribution of Python and R and includes over 100 of the most popular Python, R and Scala packages for data science. [9]*

After downloading and installing anaconda the functionality of all the needed libraries can be tested by running the following code [10].

```

1 # Check the versions of libraries
2 # Python version
3 import sys
4 print('Python: {}'.format(sys.version))
5 # scipy
6 import scipy
7 print('scipy: {}'.format(scipy.__version__))
8 # numpy
9 import numpy
10 print('numpy: {}'.format(numpy.__version__))
11 # matplotlib
12 import matplotlib
13 print('matplotlib: {}'.format(matplotlib.__version__))
14 # pandas
15 import pandas
16 print('pandas: {}'.format(pandas.__version__))
17 # scikit-learn
18 import sklearn
19 print('sklearn: {}'.format(sklearn.__version__))

```

Using the Anaconda python library the analytic results of the python code (appendix A.1) reveal the following information about the iris dataset.

At first lets take a look at some sample data, also called head data since it resembles the first few entries of a dataset.

	sepal-length	sepal-width	petal-length	petal-width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
11	4.8	3.4	1.6	0.2	Iris-setosa
12	4.8	3.0	1.4	0.1	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
17	5.1	3.5	1.4	0.3	Iris-setosa
18	5.7	3.8	1.7	0.3	Iris-setosa
19	5.1	3.8	1.5	0.3	Iris-setosa

**Figure 2.1:** Data head of the iris dataset

In figure 2.1 we can see the characteristics identifier, starting with zero and being incremented with each additional value. Since the first twenty lines of data are shown we

see the identifier zero to nineteen. We also get an overview of what margins the values of the characteristics (sepal-length, sepal-width, petal-length and petal-width) have. The last column represents the class that the characteristics in the line are describing. Since the dataset is sorted by class all of the head data shows characteristic information about the iris setosa class.

	sepal-length	sepal-width	petal-length	petal-width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

**Figure 2.2:** Detailed information about the iris dataset

Figure 2.2, a screenshot of the anaconda dataset analysis, provides an overview of not only the first twenty, but the entire dataset. The information that is shown here contains the following information about every given characteristic but not the classes.

1. count - Count of data lines that contain the given information.
2. mean - The average value of the characteristic calculated over the entire dataset.
3. min - The minimum value of the characteristic in the entire dataset.
4. 25% - First quartile
5. 50% - Second quartile
6. 75% - Third quartile
7. max - The maximum value of the characteristic in the entire dataset.

class	
Iris-setosa	50
Iris-versicolor	50
Iris-virginica	50
dtype: int64	

**Figure 2.3:** Iris classes and their quantities

Figure 2.3 gives an overview of the available classes (Iris-setosa, Iris-versicolor and Iris-virginica) and what amount of classification data is provided in the dataset for each of these classes. This dataset contains 50 data lines for each class which leads to a total of 150 lines of classification data.

To provide some more information about the datasets three figures (2.6, 2.7 and 2.8) were generated that describe the used data.

In figure 2.4 a few machine learning procedures are performed and tested using the given dataset. We can see the accuracy of the different algorithms ranging from 0 (classified none of the given data correctly) to 1 (classified all of the given data correctly).

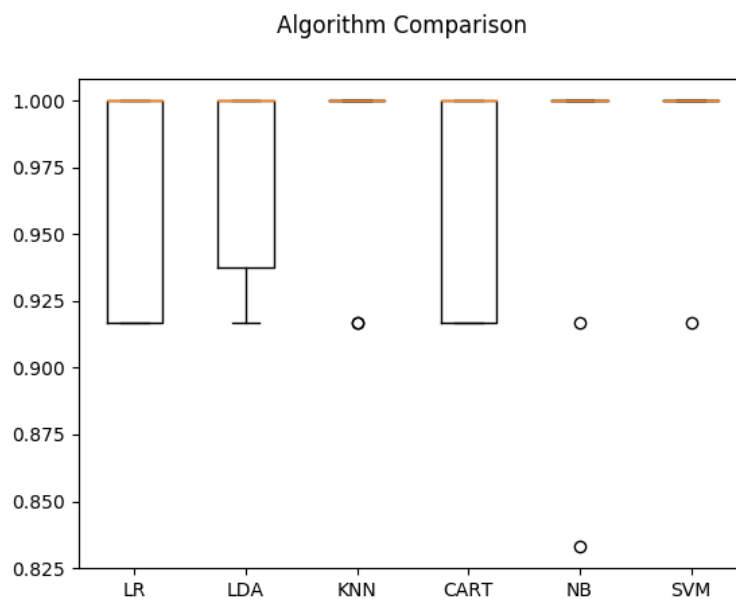
```
LR: 0.966667 (0.040825)
LDA: 0.975000 (0.038188)
KNN: 0.983333 (0.033333)
CART: 0.966667 (0.040825)
NB: 0.975000 (0.053359)
SVM: 0.991667 (0.025000)
```

**Figure 2.4:** Machine learning results for different algorithms using the iris dataset

The algorithms perform well for our use case given the amount of data that was provided to them. The different machine learning procedures used are listed below.

1. LR - Logistic Regression
2. LDA - Linear Discriminant Analysis
3. KNN - KNeighbors Classifier
4. CART - Decision Tree Classifier
5. NB - GaussianNB
6. SVM - Support Vector Machine

To provide some further insight into the performance of the different algorithms a graphical representation in form of a box diagram (figure 2.5) was generated.



**Figure 2.5:** Box diagram showing performance of different machine learning algorithms

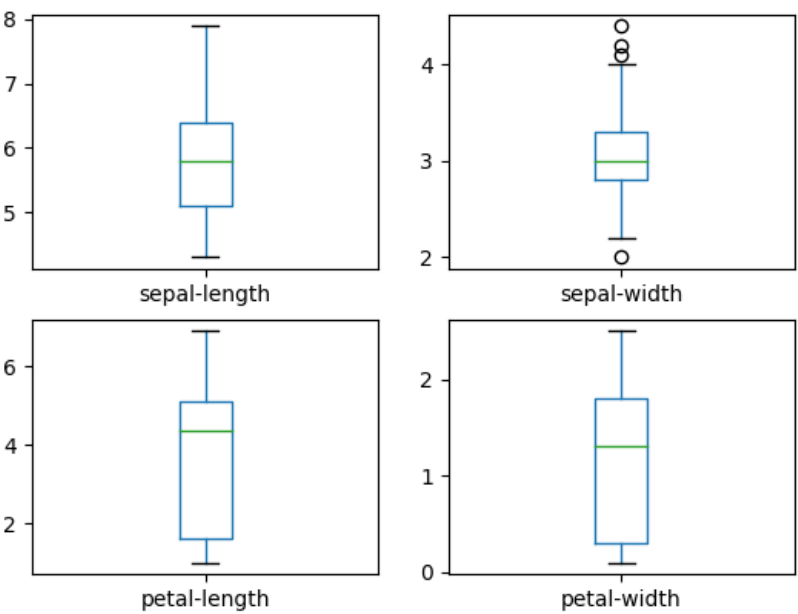


Figure 2.6: Box diagram of the iris data distribution

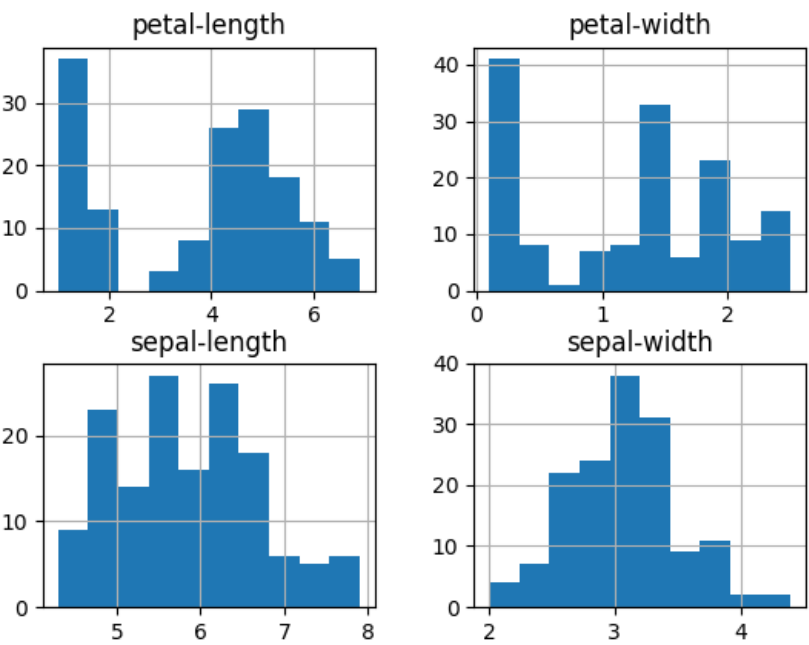
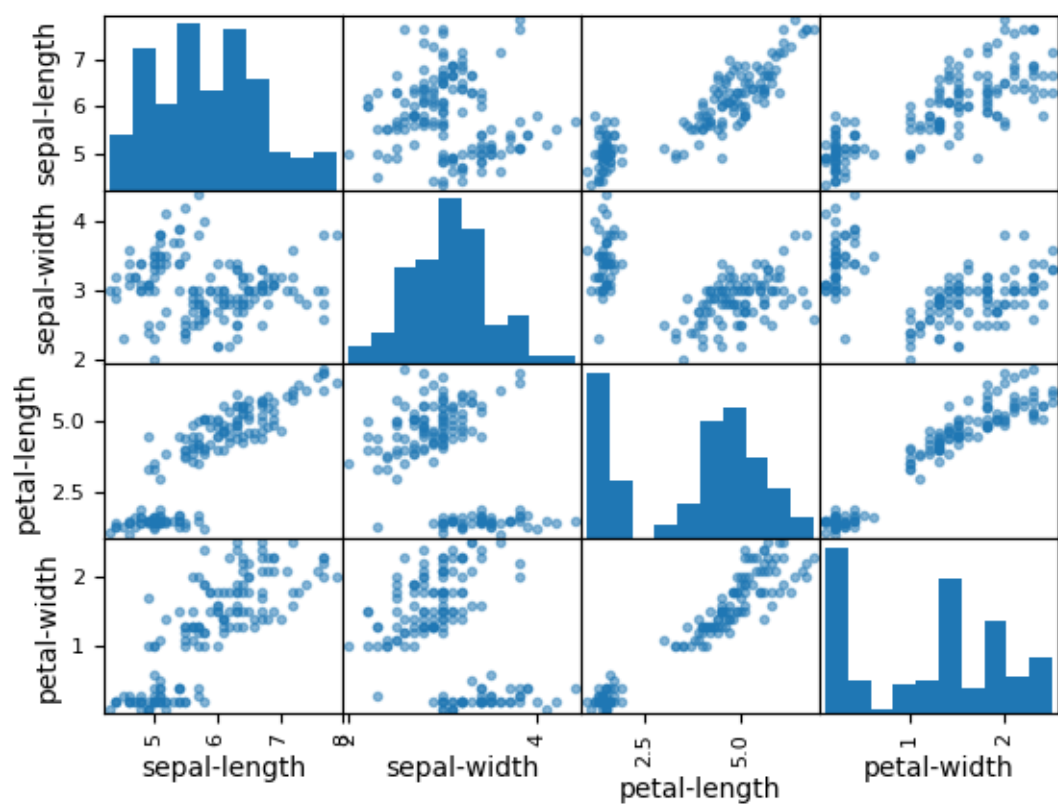


Figure 2.7: Histogram of the iris data distribution



**Figure 2.8:** Scatter diagram of the iris data distribution

## Chapter 3

# C++/C Machine Learning Libraries

There are a lot of machine learning tools out there but only a few of them are programmed to work with C++/C. By using open source collections of massive amounts of machine learning libraries [3] a more precise selection of what libraries fit the purpose and the needs of this project can be formed.



### 3.1 Shark - Machine Learning Library

The top C++/C machine learning library that is used for most projects in these languages is the Shark - Machine Learning [1] library. Its feature list contains a lot of different supervised learning, like LDA and SVM, as well as unsupervised learning, like principal component analysis and hierarchical clustering. Furthermore it supports evolutionary algorithms and basic linear algebra and optimization algorithms.

After selecting a library the setup process starts. The dependencies of *CMake* [6] and the *boost binaries* [5] have to be met before starting the actual installation process. Now that this is figured out it is time to install the actual library. To do so download the files needed from the shark [1] homepage on which the information in Figure 3.1 is shown.

Since the target operating system was set to Microsoft Windows, so that most other students and professors can reproduce the machine learning process, this was a problem. This error was not resolved in the next few days so a previous version of shark had to be used.

After installing all necessary dependencies and the machine learning library itself another problem occurred. The dynamic memory allocation that provides a lot of agility

OS	coverage	status	service
Linux and Mac OS	library and unit tests	 build passing	travis-ci
Windows	library only, no unit tests	 build failing	appveyor

**Figure 3.1:** Shark Windows build failed

on personal computers and similiar machines does not work well on MCUs. The reasons for this are the very limited amount of memory available on such devices and the fragmentation problem that occurs over long time performance on embedded systems. Futhermore systems like the target one is usually programmed to perform one task so there is no use for reallocating and reusing memory. Also memory allocation comes at a high cost of performance in terms of processing speed.

## 3.2 Alternative Approach

Since most machine learning libraries use dynamic memory allocation they are not suitable for the process of performing classification of the iris problem on an MCU. Therefore a different, more static, approach is needed to solve this task. The new target of the thesis is to develop a program that generates very performant C code that is specially designed and generated for a specific machine learning problem, as for example the iris problem. For simplicity the program creates a decision tree since in section 2.3 the results for such trees are acceptable for the selected use case.



## Chapter 4

# Decision Tree ID3

### 4.1 Functionality

The Iterative Dichotomiser 3 (ID3) algorithm generates a decision tree from a data set as for example the iris data set that is used in this case. Its most common domains are machine learning and natural language processing. After being trained with a given set of data it is used to classify future samples. It therefore meets the requirements to solve the iris problem.

### 4.2 Implementation

The source code of the implementation [11] used was originally coded and commented in italian and had to be translated entirely before taking further steps.

### 4.3 Conclusion

After the code was translated and tested the results were pretty accurate. The only problem was that this implementation still uses dynamic memory allocation which hindered perfect performance on the MCU and given certain data sets still tried allocating more memory than the 64K that were available on the MSP432 MCU [4]. Furthermore decision trees such as the ID3 are more likely to face the problem of data over-fitting. When this happens the algorithm splits the data until only pure sets are available. To fix this problem J48 (chapter 5) is used instead of ID3.

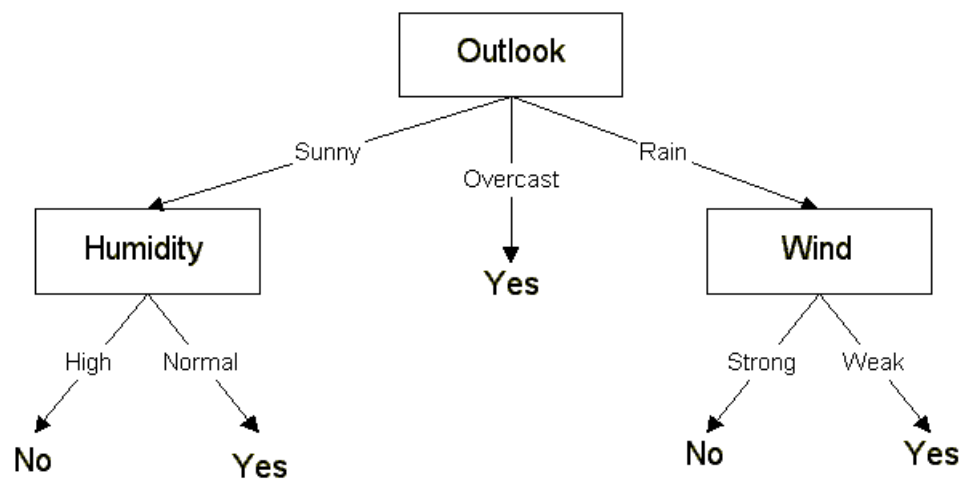


Figure 4.1: Example of an ID3 decision tree [12]

## Chapter 5

# Decision Tree J48

The J48, as well as the ID3, is used to classify data after being trained with a fixed data set. The J48 algorithm is an open source implementation of the C4.5 algorithm from the Weka data mining tool [7]. C4.5 is an extension of the ID3 algorithm mentioned in chapter 4.

### 5.1 Weka

Using Weka a J48 decision tree can quickly be trained and used to classify data inside the data mining tool. The challenge is to extract the information about the decision tree into performant, optimised C code that can run on an MCU like the used MSP432[4].

### 5.2 Preprocess Setup

Starting the Weka application prompts the user with a dialog on which the action "Explorer" is selected. This reveals the actual window that is used to create the J48 decision tree.

In the "Preprocess" tab select "Open file" and select the iris dataset provided by the UCI[2]. The content of the tab now shows the characteristics that are contained in the dataset file and a few informations about the dataset such as minimum, maximum, mean and standard deviation. It also shows a graph that represents the distribution of the characteristics. A visual representation of what the user should see at this moment is provided in figure 5.1.

### 5.3 Classification

To generate a J48 decision tree switch to the "Classify" tab and choose the "trees/J48" classifier. Select the "More options..." button and enable the "Output source code" for the "WekaClassifier" since it will give further insight into how the C code should look like. Furthermore to provide a mental cross link between the generated C code and the Weka generated Java code the unique function identifiers are also used in the C code.

The decision tree is now ready to be trained by the weka tool. Simply hit start and watch the flightless weka bird do a belly flop when the training process is finished.

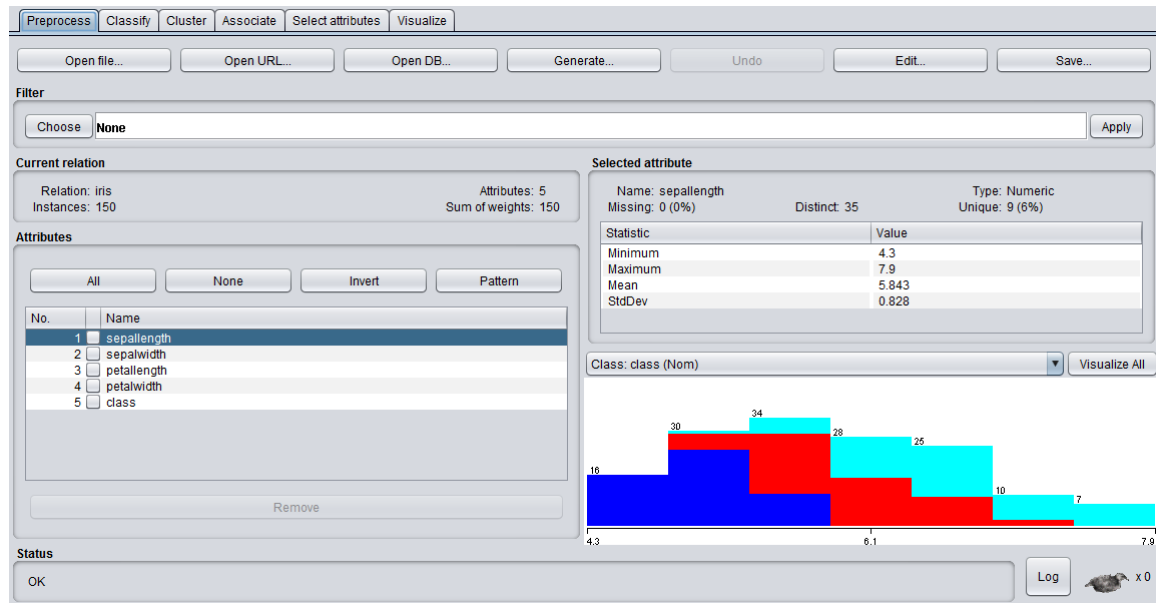


Figure 5.1: Weka Preprocess tab screenshot

A lot of data can now be seen in the "Classifier output" field. Amongst other things it contains the amount of data used to train the tree, the trees structure, its attributes and classes and the previously enabled java code. This textual representation of the decision tree is very informative, but not easy to see at the first glance how data will be processed during classification.

To get a better, visual representation of what the tree looks like, how it works and how it is going to process classification data select the result element in the "Result list" container. Right click the item and choose the "Visualize tree" option. The now generated diagram represents the J48 decision tree in a form that can be understood more easily compared to the textual description from before. Using the iris dataset the generated decision tree visualization should look similar to figure 5.2.

The training process is now finished and the J48 decision tree is ready to be used for classification of data sets.

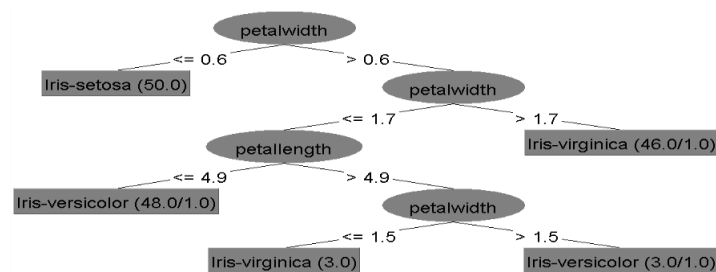


Figure 5.2: Weka result for the J48 decision tree

## Chapter 6

# Process Generated Tree

Having now trained a fully functional J48 decision tree the next task is to process the newly acquired information into generating C code out of it that can be run on the chosen MCU, the MSP432.

To accomplish this target the information that is created by the weka data mining tool has to be read and parsed by another program. This program then generates two C main files that solve the same problem, but use different implementations. The program is called "W2C Converter" (Weka-to-C Converter). A screenshot of this program can be seen in figure 6.3. This screenshot can be used through out this chapter to get a better impression on how the application has to be operated.

### 6.1 W2C Converter

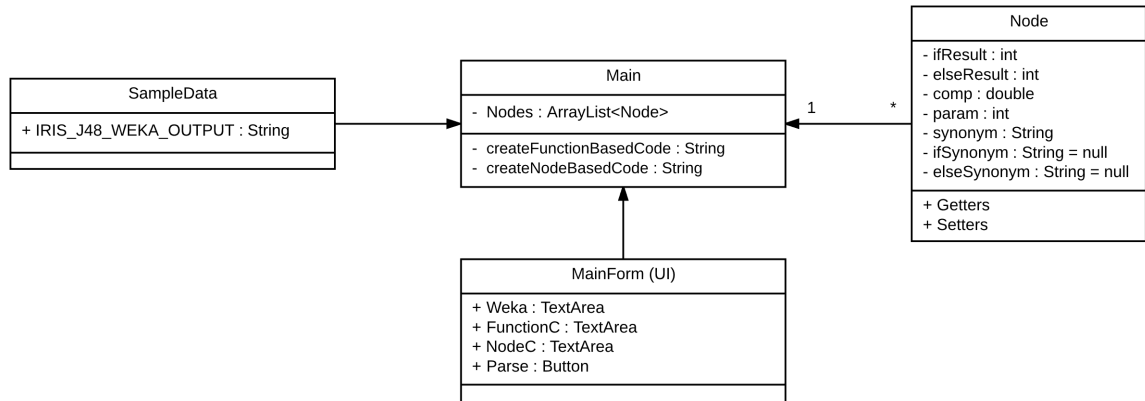
The programming language of choice is java since it provides a lot of versatility and simplicity. There are no additional libraries or other kind of sources needed to compile this program.

To further understand the working process lets split it up into subsections that are each explained individually.

#### 6.1.1 Structure

First of all let us look at the structure that the software is built upon. Its basic components are the Main-, Node- and SampleData classes. These all work together to accomplish the successful conversion of the textual weka J48 tree definition to the function based 6.3 and node based 6.4 c-codes. Figure 6.1 shows the class diagram of the application. In short the functionality of each of those classes is as follows.

- **Node** class is used as a data container.
- **Main** class performs the conversion.
- **SampleData** class provides fast debugging and assists in delimitation of several values. This class will therefore not be described in more detail since it majorly helped in the developing process and is not a critical component of the final software product.



**Figure 6.1:** Class diagram of the W2C Converter

### 6.1.2 Node

The node class is really the critical factor since it has to be modelled to fit result values, forwarding links and synonyms into one data model. To accomplish the functionality of the parser the node therefore contains two result codes (if the condition is met and if it isn't), a compare and param value that defines with which value has to be compared to decide on further forwarding and three synonym strings (own, condition met and not met).

The strings that are referred to as synonyms are used to name either the functions or the structs and therefore fulfill two major requirements. First, the individuality of each node's reference is provided (preventing ambiguous calls) and second, every single step the final c code product takes can be referred back to the textual weka tree definition.

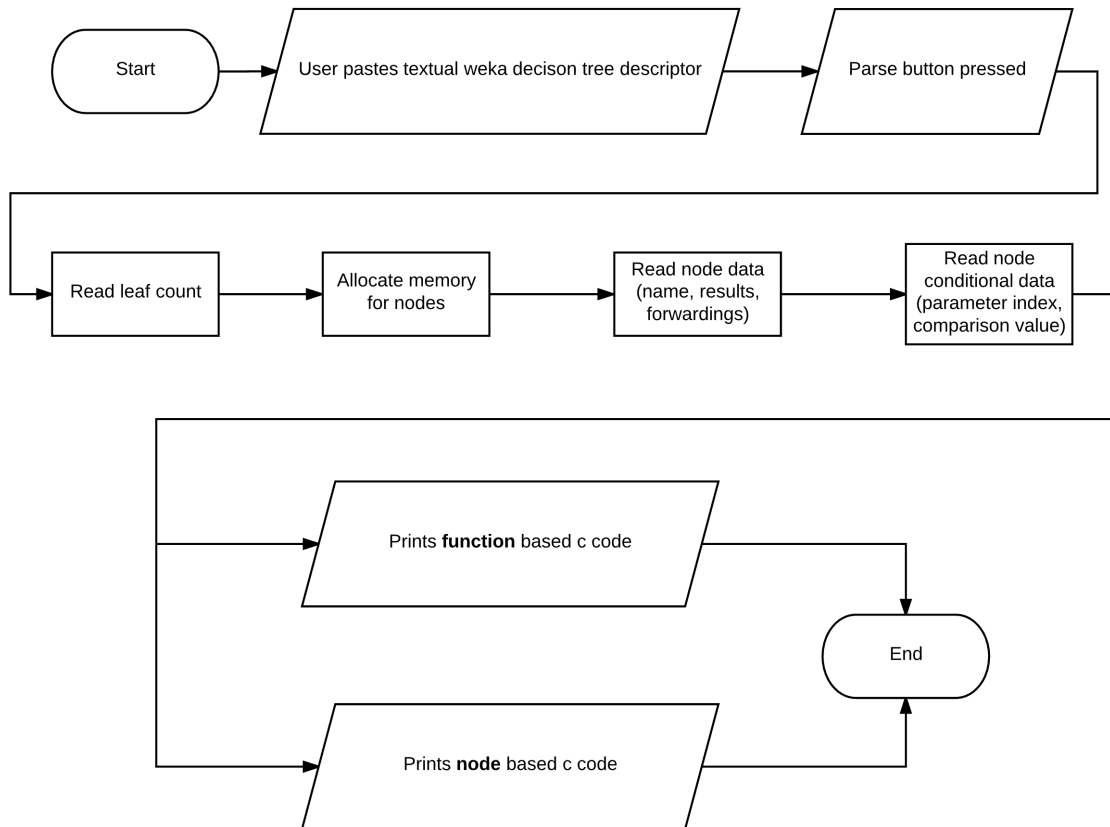
### 6.1.3 Main

Now that the data structure behind the process is clear the data extraction can be explained in further detail. The process is split up and explained step by step in the following enumeration.

1. Reading the **leaf count** provided by the textual weka j48 decision tree description. This variable is used to know the exact amount of iterations that have to be taken until the software detected every node.
2. The converter now iterates over the entire file **allocating memory** for each node found. During the same iteration process all node data is extracted. This behaviour allows the program to use a single iteration for any code, so no exponential scaling will appear on bigger textual weka tree descriptors.
3. When allocating memory the software automatically extracts the **name** of the current node and stores it in the *own synonym* variable.
4. Extracting the **if- and else results** that determine as what the current data line provided currently is classified as. If there is no more forwarding this is the final result the classification tree returns. In the same step, if a **forwarding node** is found a reference to it will be stored in the *forwarding synonym* variables.

5. Now that the enviornmental forwarding is prepared the actual **comparison data** that makes the tree decide which branch to follow next is extracted and stored in the *compare* and *param* values.

Now all critical data is extracted from the textual weka decision tree descriptor. This enables the actual process of printing c code. To get a better overview of the work the W2C Converter performs take a look at figure 6.2.



**Figure 6.2:** Flow diagram of the W2C Converter

## 6.2 W2C Converter Usage

The W2C Converter provides the user with a basic list of instructions that, if followed correctly, lead to the successful generation of two C code main files. Their differences are described in further detail in sections 6.3 and 6.4. If the previous steps from chapter 5 were followed no further steps need to be taken in the weka data mining tool. Simply change focus of the weka explorer to the "Classifier output" text field by left clicking once inside it. Then, using the command "Select everything" (Ctrl+A) we can copy (Ctrl+C) the entire classifier output text and paste (Ctrl+V) it into the W2C Converters "Weka output" field.

### 6.3 Function Based C Code

First lets take a look at the implementation that utilizes functions to solve the problem. It generates a function for each leaf in the decision tree that contains an if-else clause to decide on which function to forward the data to. This C implementation of a decision tree focuses mainly on the usage of the flash memory that an MCU usually has got more of compared to RAM. The only two memory allocations in the generated C main file are for the classification result and the test dataset itself. This code is very comparable to the java code generated by the Weka data mining tool except in C function headers are required to cross access the different function nodes if needed.

An example node that was taken from the iris data test set looks like this (comments were added after code generation for comprehensibility):

```

1 int Nd8409be2(double i[]) // Unique node synonym
2 {
3     int p = -1;           // Init result as -1 in case of error
4     if (i[2] <= 4.9)      // Parameter index and compare value
5     {
6         p = 1;           // Successful classification to class 1 (check result table)
7     }
8     else
9     {
10        p = N5e2faa8d3(i); // Forwarding classification process to the next node
11    }
12    return p;             // Returns classified result code
13 }
```

### 6.4 Node Based C Code

The other implementation of the decision tree uses Node structs that contain the parameters index which they are comparing to the also contained compare value. Based on whether the parameter is higher or lower than the compare value a check function either forwards the classification request to a different node that is processed recursively also in the check function or returns a finished classification value.

If an error occurs during the classification process the classification result will be -1 signaling the user that the classification was not successful and enabling the simple usage of "if(result)" to check for errors.

The node struct that is required for this kind of computation looks like this (comments were added after code generation for comprehensibility):

```

1 struct Node // Struct name
2 {
3     int param;           // Parameter index
4     double comp;         // Compare value
5     int ifResult;        // Successful classification (class value, check result table)
6     int elseResult;
7     struct Node* ifNode; // Forwarding classification process to the next node
8     struct Node* elseNode;
9 };
```

A generated sample node from the iris data test set would therefore by definition look like this:



```
1 struct Node N2a8166703 = { .param = 3, .comp = 1.5, .ifResult = 2, .elseResult = 1,
    .ifNode = 0, .elseNode = 0 };
```

The previously mentioned check function will always look like this:

```
1 int check(double i[], struct Node* node)
2 {
3     if (i[node->param] <= node->comp)
4     {
5         return node->ifResult >= 0 ? node->ifResult : check(i, node->ifNode);
6     }
7     else
8     {
9         return node->elseResult >= 0 ? node->elseResult : check(i, node->elseNode);
10    }
11 }
```

## 6.5 Additional Features

All that has to be done now is to copy any one of the generated C codes and paste them into an MSP432 project as described in 7.

### 6.5.1 MSP432 Ready

Both of the generated codes already include the msp header file and stop the watchdog timer in the main function. The generated codes even contain some iris characteristic sample data that can be uncommented to immediately compile and test the application.

### 6.5.2 Result Table

Furthermore both generated codes provide the a comment section containing all possible classification outcomes and their corresponding values. For the iris dataset this comment section looks as follows:

```
1 /**
2  * RESULT TABLE:
3  * 0 - setosa
4  * 1 - versicolor
5  * 2 - virginica
6  */
```

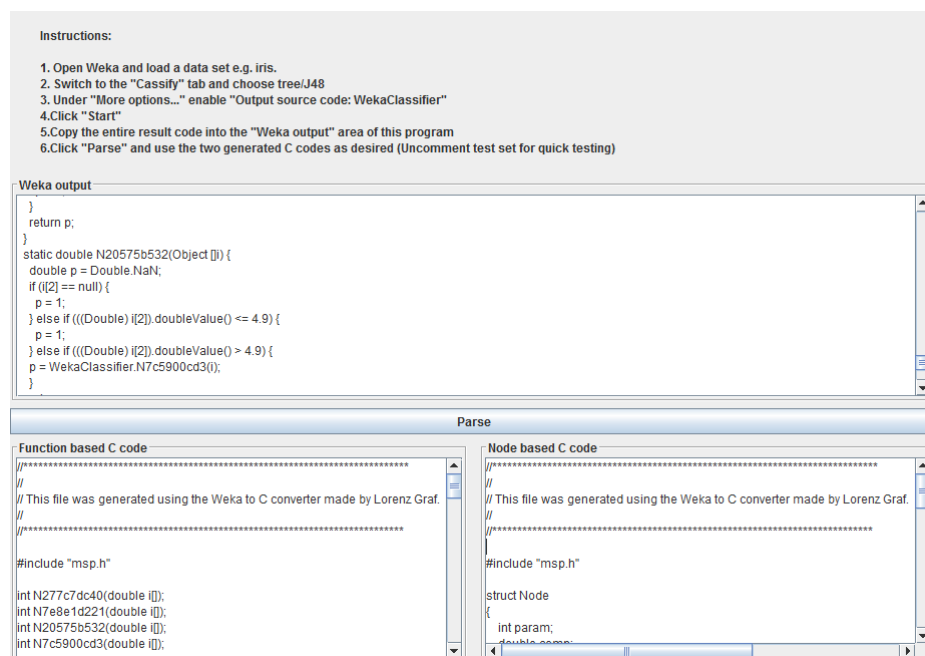


Figure 6.3: Screenshot of the W2C Converter

## Chapter 7

# Classification on MCU

To use the decision tree C code that was generated by the W2C Converter simply copy and paste the corresponding text fields content into a C main file. After saving the file and making sure that all dependencies, such as header files and compilers, are met the file can be compiled to machine code. Make sure to load the program onto the development board or chip that is used.

A different, more simple approach to compiling and loading the decision tree code is explained in section 7.1.

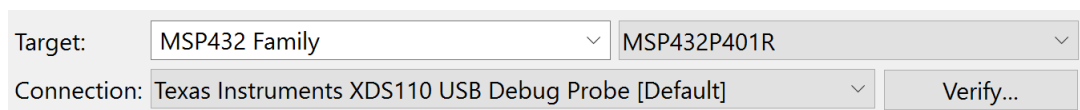
### 7.1 Code Composer Studio

Compiling and loading can be done for example using the Code Composer Studio[8] which integrates the libraries and compilers required to develop an application for the MSP432.

#### 7.1.1 Project Setup

First, create a new project by selecting "File > New > CCS Project".

Configure the Code Composer Studio setup wizard to match the target development board. In this case an MSP432P401R is used and therefore the configuration should be as seen in figure 7.1.



Target:	MSP432 Family	MSP432P401R
Connection:	Texas Instruments XDS110 USB Debug Probe [Default]	Verify...

**Figure 7.1:** Code Composer Studio target setup for MSP432

#### 7.1.2 Configure Project

After making sure a wired usb connection is established between the computer and the target development board check its functionality using the "Verfiy..." button in the connection row of the project setup wizard.

After giving the project a name it is ready to be created and loaded inside the project explorer bar.

### 7.1.3 Implementing C Code

Open the `main.c` file that is located in the root folder of the project and replace its entire content by overwriting it with one of the two generated C codes mentioned in chapter 6 in sections 6.3 and 6.4.

### 7.1.4 Classification Data

If required uncomment the example iris data test values to immediately be able to build the project using the hammer button in the Code Composer Studio IDE as described in section 6.5.

Any given set of data that might be classified by the generated decision tree C code, as well as other functionalities like bluetooth connectivity (etc.) may now be implemented into the code.

After successfully building the project the code may now be run or debugged on the development board.

## 7.2 Evaluating Results

The classification of characteristic, added or modified in section 7.1.4, is either performed by calling the root function for the function based solution or calling the check function using the root node as parameter for the node based solution.

Evaluation can be performed by comparing the result of the classification with the automatically generated result table mentioned in section 6.5.2.

## Chapter 8

# Conclusion

Even though the resources provided by MCUs are not meant to perform machine learning tasks 1.2 it is most certainly possible. If done correctly the machine learning process is very performant and consistent.

Machine learning libraries, such as shark 3.1, are most likely not the best way to perform classification or similar machine learning processes on an MCU. This can be lead back mostly to the usage of dynamic memory allocation 3.1 which causes several problems in the environment of an MCU.

By creating a program that uses information about a previously trained decision tree 5 as input and outputs code that is compatible with the target MCU, such as the W2C Converter 6.1, the effort of using MCUs for classification shrinks to a minimum.

Doing so combines the high consistency of embedded systems with the versatility of general computing machines like a personal computer or a mac.

# Appendix A

## Source Code

### A.1 Python Analytics Code

```
1 # Load libraries
2 import pandas
3 from pandas.tools.plotting import scatter_matrix
4 import matplotlib.pyplot as plt
5 from sklearn import model_selection
6 from sklearn.metrics import classification_report
7 from sklearn.metrics import confusion_matrix
8 from sklearn.metrics import accuracy_score
9 from sklearn.linear_model import LogisticRegression
10 from sklearn.tree import DecisionTreeClassifier
11 from sklearn.neighbors import KNeighborsClassifier
12 from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
13 from sklearn.naive_bayes import GaussianNB
14 from sklearn.svm import SVC
15
16 # Load dataset
17 url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
18 names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
19 dataset = pandas.read_csv(url, names=names)
20
21 # shape
22 print(dataset.shape)
23
24 # head
25 print(dataset.head(20))
26
27 # descriptions
28 print(dataset.describe())
29
30 # class distribution
31 print(dataset.groupby('class').size())
32
33 # box and whisker plots
34 dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
35 plt.show()
36
37 # histograms
```

```
38 dataset.hist()
39 plt.show()
40
41 # scatter plot matrix
42 scatter_matrix(dataset)
43 plt.show()
44
45 # Split-out validation dataset
46 array = dataset.values
47 X = array[:,0:4]
48 Y = array[:,4]
49 validation_size = 0.20
50 seed = 7
51 X_train, X_validation, Y_train, Y_validation = model_selection.train_test_split(X, Y
    , test_size=validation_size, random_state=seed)
52 # Test options and evaluation metric
53 seed = 7
54 scoring = 'accuracy'
55
56 # Spot Check Algorithms
57 models = []
58 models.append(('LR', LogisticRegression()))
59 models.append(('LDA', LinearDiscriminantAnalysis()))
60 models.append(('KNN', KNeighborsClassifier()))
61 models.append(('CART', DecisionTreeClassifier()))
62 models.append(('NB', GaussianNB()))
63 models.append(('SVM', SVC()))
64
65 # evaluate each model in turn
66 results = []
67 names = []
68 for name, model in models:
69     kfold = model_selection.KFold(n_splits=10, random_state=seed)
70     cv_results = model_selection.cross_val_score(model, X_train, Y_train, cv=kfold,
        scoring=scoring)
71     results.append(cv_results)
72     names.append(name)
73     msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
74     print(msg)
75
76 # Compare Algorithms
77 fig = plt.figure()
78 fig.suptitle('Algorithm Comparison')
79 ax = fig.add_subplot(111)
80 plt.boxplot(results)
81 ax.set_xticklabels(names)
82 plt.show()
83
84 # Make predictions on validation dataset
85 NB = GaussianNB()
86 NB.fit(X_train, Y_train)
87 predictions = NB.predict(X_validation)
88 print(accuracy_score(Y_validation, predictions))
89 print(confusion_matrix(Y_validation, predictions))
90 print(classification_report(Y_validation, predictions))
```

## A.2 Generated Function Based C Code

```
1 //*****
2 //
3 // This file was generated using the Weka to C converter made by Lorenz Graf.
4 //
5 //*****
6
7 #include "msp.h"
8
9 int N6635884e0(double i[]);
10 int N4cf3e16a1(double i[]);
11 int N7c0eba942(double i[]);
12 int N2a8166703(double i[]);
13
14 int N6635884e0(double i[])
15 {
16     int p = -1;
17     if (i[3] <= 0.6)
18     {
19         p = 0;
20     }
21     else
22     {
23         p = N4cf3e16a1(i);
24     }
25     return p;
26 }
27
28 int N4cf3e16a1(double i[])
29 {
30     int p = -1;
31     if (i[3] <= 1.7)
32     {
33         p = N7c0eba942(i);
34     }
35     else
36     {
37         p = 2;
38     }
39     return p;
40 }
41
42 int N7c0eba942(double i[])
43 {
44     int p = -1;
45     if (i[2] <= 4.9)
46     {
47         p = 1;
48     }
49     else
50     {
51         p = N2a8166703(i);
52     }
53     return p;
54 }
```



```

55
56 int N2a8166703(double i[])
57 {
58     int p = -1;
59     if (i[3] <= 1.5)
60     {
61         p = 2;
62     }
63     else
64     {
65         p = 1;
66     }
67     return p;
68 }
69
70 // Example test set (IRIS)
71 // double testSet [] = { 5.0,3.6,1.4,0.2 };
72
73 // Results:
74 // 0 - Iris-setosa
75 // 1 - Iris-versicolor
76 // 2 - Iris-virginica
77
78 void main(void)
79 {
80     // Stop watchdog timer
81     WDTCTL = WDTPW | WDTHOLD;
82
83     int result = N4cf3e16a1(testSet);
84 }

```

### A.3 Generated Node Based C Code

```

1 //*****
2 //
3 // This file was generated using the Weka to C converter made by Lorenz Graf.
4 //
5 //*****
6
7 #include "msp.h"
8
9 struct Node
10 {
11     int param;
12     double comp;
13     int ifResult;
14     int elseResult;
15     struct Node* ifNode;
16     struct Node* elseNode;
17 };
18
19 struct Node N2a8166703 = { .param = 3, .comp = 1.5, .ifResult = 2, .elseResult = 1,
    .ifNode = 0, .elseNode = 0 };
20 struct Node N7c0eba942 = { .param = 2, .comp = 4.9, .ifResult = 1, .elseResult = 0,
    .ifNode = 0, .elseNode = &N2a8166703 };
21 struct Node N4cf3e16a1 = { .param = 3, .comp = 1.7, .ifResult = 0, .elseResult = 2,

```

```
        .ifNode = &N7c0eba942, .elseNode = 0 };
22 struct Node N6635884e0 = { .param = 3, .comp = 0.6, .ifResult = 0, .elseResult = 0,
    .ifNode = 0, .elseNode = &N4cf3e16a1 };
23
24 int check(double i[], struct Node* node)
25 {
26     if (i[node->param] <= node->comp)
27     {
28         return node->ifResult >= 0 ? node->ifResult : check(i, node->ifNode);
29     }
30     else
31     {
32         return node->elseResult >= 0 ? node->elseResult : check(i, node->elseNode);
33     }
34 }
35
36 // Example test set (IRIS)
37 // double testSet [] = { 5.0,3.6,1.4,0.2 };
38
39 // Results:
40 // 0 - Iris-setosa
41 // 1 - Iris-versicolor
42 // 2 - Iris-virginica
43
44 void main(void)
45 {
46     // Stop watchdog timer
47     WDTCTL = WDTPW | WDTHOLD;
48
49     int result = check(testSet, &N4cf3e16a1);
50 }
```

# References

## Literature

- [1] Christian Igel, Verena Heidrich-Meisner, and Tobias Glasmachers. “Shark”. *Journal of Machine Learning Research* 9 (2008), pp. 993–996 (cit. on p. 8).
- [2] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: <http://archive.ics.uci.edu/ml> (cit. on pp. 2, 12).
- [3] Joseph Misiti. *Awesome Machine Learning*. <https://github.com/josephmisiti/awesome-machine-learning>. 2017 (cit. on p. 8).
- [4] *MSP432P401R SimpleLink™ Microcontroller LaunchPad™ Development Kit (MSP-EXP432P401R) - Datasheet*. SLAU597C. Revised March 2017. Texas Instruments. Mar. 2015 (cit. on pp. 1, 10, 12).

## Films and audio-visual media

- [5] Boost. *Boost C++ Libraries*. Version 1.64. Mar. 17, 2017. URL: <http://www.boost.org/> (cit. on p. 8).
- [6] CMake. *CMake GUI*. Version 3.8.0. Mar. 17, 2017. URL: <https://cmake.org/> (cit. on p. 8).
- [7] Ian H. Witten Eibe Frank Mark A. Hall. *The WEKA Workbench*. <http://www.cs.waikato.ac.nz/ml/weka/>. Version 3.8. 2017 (cit. on pp. 2, 12).
- [8] Texas Instruments. *Code Composer Studio*. Version 7.1.0. Mar. 17, 2017. URL: <http://www.ti.com/tool/CCSTUDIO> (cit. on pp. 1, 20).

## Online sources

- [9] *Anaconda Software Distribution*. Version 4.3.1. Continuum Analytics. URL: <https://continuum.io> (visited on 05/28/2017) (cit. on p. 2).
- [10] Jason Brownlee. *Your First Machine Learning Project in Python Step-By-Step*. URL: <http://machinelearningmastery.com/machine-learning-in-python-step-by-step/> (visited on 05/28/2017) (cit. on p. 3).
- [11] Daniel Fingers. *ID3 Algorithm Implementation in C*. URL: <http://id3alg.altervista.org/> (visited on 05/28/2017) (cit. on p. 10).

- [12] *The ID3 Algorithm*. Herbert Wertheim College of Engineering. URL: <https://www.cise.ufl.edu/~ddd/cap6635/Fall-97/Short-papers/2.htm> (visited on 05/28/2017) (cit. on p. 11).