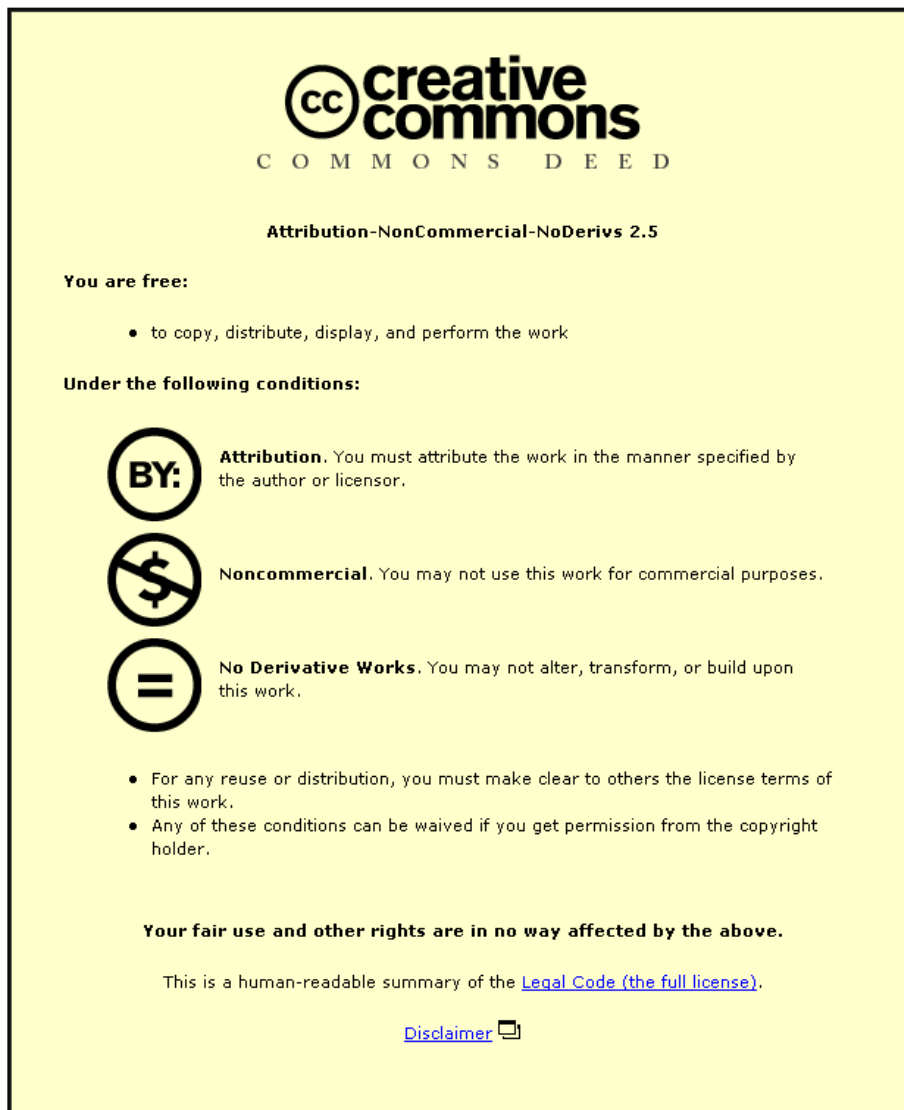


This item was submitted to Loughborough's Institutional Repository (<https://dspace.lboro.ac.uk/>) by the author and is made available under the following Creative Commons Licence conditions.



CC creative commons
COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

BY: **Attribution.** You must attribute the work in the manner specified by the author or licensor.


Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the [Legal Code \(the full license\)](#).

[Disclaimer](#) 

For the full text of this licence, please go to:
<http://creativecommons.org/licenses/by-nc-nd/2.5/>

Implementation of Decision Trees for Embedded Systems

By

Bashar E. A. Badr

A doctoral thesis submitted in partial fulfilment of the
requirements for the degree of

Doctor of Philosophy

School of Electronic, Electrical and Systems Engineering

Loughborough University, United Kingdom

May 2014

© Bashar E. A. Badr, 2014

ABSTRACT

This research work develops real-time incremental learning decision tree solutions suitable for real-time embedded systems by virtue of having both a defined memory requirement and an upper bound on the computation time per training vector. In addition, the work provides embedded systems with the capabilities of rapid processing and training of streamed data problems, and adopts electronic hardware solutions to improve the performance of the developed algorithm.

Two novel decision tree approaches, namely the Multi-Dimensional Frequency Table (MDFT) and the Hashed Frequency Table Decision Tree (HFTDT) represent the core of this research work. Both methods successfully incorporate a frequency table technique to produce a complete decision tree.

The MDFT and HFTDT learning methods were designed with the ability to generate application specific code for both training and classification purposes according to the requirements of the targeted application. The MDFT allows the memory architecture to be specified statically before learning takes place within a deterministic execution time.

The HFTDT method is a development of the MDFT where a reduction in the memory requirements is achieved within a deterministic execution time. The HFTDT achieved low memory usage when compared to existing decision tree methods and hardware acceleration improved the performance by up to 10 times in terms of the execution time.

ACKNOWLEDGEMENT

First and foremost, I praise Allah the Almighty, the Most Gracious, the Most Merciful who blessed me with strength, health, patience to finish this work.

I would like to express my sincere and deepest gratitude to my supervisor Dr David J. Mulvaney for his continuous support, patience, motivation and valuable advice throughout this research work. His guidance, assistance, effort and critical comments were essential to complete this research work and the writing of the thesis.

I would like to thank my second supervisor Dr Vassilios A. Chouliaras for his positive energy, continuous support and valuable advice that helped me throughout the time of my research.

I would like to extend my warmest thanks to all staff members of the school of Electronic, Electrical and Systems Engineering at Loughborough University especially the IT department for their kind help and support.

My appreciation also goes to my research colleagues and my friends in Loughborough for their unlimited support that made my stay in Loughborough so memorable. I am also grateful to my friends in Jordan for their continual encouragement.

Last but not least, I would like to express my deepest gratitude and thanks to my parents and my brothers for their endless care and support throughout my study in Loughborough. Without the praying, blessing and the encouragement of my parents I couldn't achieve my goals.

DEDICATION

*I dedicate this work to my parents for
their unconditional love, enormous
support, encouragement and guidance
throughout my life to date.*

RESEARCH PUBLICATIONS

1. Bashar E. Badr, and David J. Mulvaney, **Hardware Implementation of Decision Trees for Robotic Applications**, Research Conference 2010, RSSE, Loughborough University, May 2010
2. Bashar E. Badr, David J. Mulvaney, and Vassilios A. Chouliaras, **Implementation of Decision Trees in a Configurable Multiprocessor Architecture** in *VLSI-SOC 2011: Proceedings of the 19th IFIP/IEEE International Conference on Very Large Scale Integration 2011*, Kowloon, Hong Kong, China, Oct. 3-5, 2011.
3. Bashar E. Badr, David J. Mulvaney, and Vassilios A. Chouliaras, **Implementation of Decision Trees for Embedded Applications**. PhD Conference 2012, EESE, Loughborough University, May 2012.

ABBREVIATIONS

AI	: Artificial Intelligence
ANN	: Artificial Neural Networks
ASIC	: Application-Specific Integrated Circuit
DT	: Decision Tree
FPGA	: Field Programmable Gate Array
FT	: Frequency Table
FTDT	: Frequency Table Decision Tree
GA	: Genetic Algorithm
HDL	: Hardware description language
HFT	: Hashed Frequency Table
HFTDT	: Hashed Frequency Table Decision Tree
HLS	: High-level Synthesis
ID3	: Induction Dichotomizer – version 3
ID4	: Induction Dichotomizer – version 4
ID5	: Induction Dichotomizer – version 5
ID5R	: Modified Version of Induction Dichotomizer – version 5
IDP	: Incremental Decision Path
IG	: Information Gain
IP	: Intellectual Property
ITI	: Incremental Tree Inducer
LUT	: Look up table
MDFT	: Multi-Dimensional Frequency Table
ML	: Machine Learning
RL	: Reinforcement Learning
RTL	: Register Transfer Level
SVM	: Support Vector Machines

TABLE OF CONTENTS

ABSTRACT	II
ACKNOWLEDGEMENT	III
RESEARCH PUBLICATIONS	V
ABBREVIATIONS	VI
TABLE OF CONTENTS	VII
LIST OF FIGURES	XI
LIST OF TABLES	XIII
1 INTRODUCTION	1
1.1 MOTIVATION.....	2
1.2 AIM AND OBJECTIVES.....	2
1.3 CONTRIBUTIONS TO KNOWLEDGE	3
1.4 THESIS ORGANIZATION	4
1.5 SUMMARY.....	5
2 BACKGROUND	6
2.1 MACHINE LEARNING SYSTEMS.....	6
2.1.1 <i>REAL-TIME EMBEDDED SYSTEMS</i>	8
2.1.2 <i>REAL-TIME INCREMENTAL LEARNING SYSTEMS</i>	8
2.1.3 <i>METHODS USED FOR MACHINE LEARNING</i>	9
2.2 DECISION TREES	10
2.2.1 <i>NON-INCREMENTAL AND INCREMENTAL LEARNING DT ALGORITHMS</i>	11
2.2.2 <i>FTDT AND IDP INCREMENTAL LEARNING METHODS</i>	11
2.2.3 <i>ENTROPY, INFORMATION GAIN AND GAIN RATIO</i>	14
2.2.4 <i>DECISION TREE PRUNING</i>	16
2.2.5 <i>HARDWARE IMPLEMENTATIONS OF DECISION TREES</i>	18
2.3 CONCLUSION.....	21
3 EXPERIMENTAL PROCEDURE	22
3.1 INTRODUCTION.....	22
3.2 MDFT AND HFTDT STRUCTURE	23

3.2.1	<i>GENERATE DECISION TREE CODE</i>	24
3.2.2	<i>DECISION TREE TRAINING</i>	26
3.2.3	<i>CLASSIFICATION USING THE DECISION TREE</i>	29
3.3	SUMMARY.....	30
4	MULTI-DIMENSIONAL FREQUENCY TABLE DECISION TREE.....	31
4.1	INTRODUCTION.....	31
4.2	FREQUENCY TABLE FOR DECISION TREES.....	32
4.3	NOVELTY OF MDFT.....	33
4.4	DECISION TREE CALCULATIONS.....	34
4.5	MAXIMUM NUMBER OF CALCULATIONS (WORST CASE SCENARIO).....	36
4.6	CALCULATING THE MEMORY USAGE OF THE MDFT METHOD	38
4.7	GENERATING A DT AND RULES USING THE MDFT METHOD	41
4.8	AN ILLUSTRATIVE EXAMPLE OF THE MDFT METHOD	42
4.9	EVALUATION OF THE MDFT METHOD	73
4.10	SUMMARY.....	77
5	HASHED FREQUENCY TABLE DECISION TREE.....	79
5.1	INTRODUCTION.....	79
5.2	THE HASHED FREQUENCY TABLE.....	80
5.3	TECHNIQUES FOR THE INDEX AND REVERSE INDEX FUNCTIONS.....	81
5.3.1	<i>INDEX FUNCTION</i>	81
5.3.2	<i>REVERSE INDEX FUNCTION</i>	82
5.3.3	<i>AN ILLUSTRATIVE EXAMPLE</i>	83
5.3.4	<i>ALTERNATIVE INDEX FUNCTION TECHNIQUE</i>	84
5.3.5	<i>ALTERNATIVE REVERSE INDEX FUNCTION TECHNIQUE</i>	85
5.4	HFTDT METHOD CALCULATIONS.....	86
5.4.1	<i>MAXIMUM NUMBER OF CALCULATIONS (WORST CASE SCENARIO)</i>	86
5.4.2	<i>CALCULATING MEMORY USAGE</i>	86
5.5	GENERATING THE DT USING HFTDT	87
5.6	AN ILLUSTRATIVE EXAMPLE FOR HFTDT METHOD	89
5.7	COMPARISON BETWEEN HFTDT AND MDFT	92
5.8	SUMMARY.....	93
6	EXPERIMENTS TO GENERATE DECISION TREES USING HFTDT	94
6.1	INTRODUCTION.....	94
6.2	CLASSIFIERS USED IN THE EXPERIMENT	95

6.3	TEST DATASETS	95
6.4	TEST RESULTS	97
6.4.1	NUMBER OF NODES.....	98
6.4.2	CALCULATION TIME	100
6.4.3	MEMORY USAGE	103
6.4.4	CLASSIFICATION ACCURACY.....	105
6.5	CONCLUSION.....	108
7	HARDWARE IMPLEMENTATION OF HFTDT	109
7.1	INTRODUCTION.....	109
7.2	HFTDT CODE PROFILING.....	110
7.2.1	SOURCE CODE RECOMPILATION.....	110
7.2.2	RESULT OBTAINED USING THE FLAT PROFILE.....	110
7.2.3	RESULTS OBTAINED USING THE CALL GRAPH.....	114
7.3	HARDWARE DESIGN FOR THE MOST TIME-CONSUMING FUNCTION	117
7.3.1	HIGH-LEVEL SYNTHESIS TOOLS	117
7.3.2	CHOOSING AN HLS TOOL	118
7.3.3	HARDWARE DESIGN USING THE HLS TOOL	119
7.3.4	DESIGN SYNTHESIS AND CO-SIMULATION.....	119
7.3.5	TARGETED FPGAs.....	121
7.3.6	DATASET USED IN THE HARDWARE DESIGN.....	122
7.4	SIMULATION RESULTS.....	122
7.4.1	NURSERY DATASET	124
7.4.2	RESULTS FOR THE AGARICUS-LEPIOTA DATASET.....	125
7.4.3	RESULTS FOR THE CHESS DATASET	127
7.4.4	RESULTS SUMMARY.....	128
7.5	EXTENSION OF THE PARALLEL IMPLEMENTATION	131
7.6	SUMMARY.....	133
8	CONCLUSIONS AND FURTHER WORK.....	134
8.1	CONCLUSIONS.....	134
8.1.1	MULTI-DIMENSIONAL FREQUENCY TABLE METHOD (MDFT).....	134
8.1.2	HASHED FREQUENCY TABLE DECISION TREE METHOD (hftdt).....	135
8.2	FURTHER WORK	136
	REFERENCES:.....	138
	APPENDIX A AN ILLUSTRATIVE EXAMPLE FOR THE HFTDT METHOD	146

APPENDIX B	SOURCE CODE FUNCTIONS	171
B.1	HFTDT ALGORITHM SOURCE CODE FILES	172
B.2	SOURCE CODE FUNCTIONS LIST	173
APPENDIX C	TABLES RESULTS	176
C.1	DATA SETS SIMULATION RESULTS	177
APPENDIX D	RESOURCES INCLUDED ON THE DVD	180

LIST OF FIGURES

FIGURE 2.1: MACHINE LEARNING SYSTEM.....	7
FIGURE 2.2: FTDT ALGORITHM [11]	13
FIGURE 2.3: IDP ALGORITHM [11].....	14
FIGURE 3.1: THE PROGRAMS INVOLVED IN THE IMPLEMENTATION OF THE MDFT AND HFTDT APPROACHES.....	23
FIGURE 3.2: THE GENERATE PROGRAM.....	24
FIGURE 3.3: THE PROGRAM TO TRAIN THE DECISION TREE	28
FIGURE 3.4: TWO DIFFERENT APPROACHES USED FOR STORING THE TRAINING DATA IN A FREQUENCY TABLE.....	29
FIGURE 3.5: CLASSIFICATION USING THE DECISION TREE.....	30
FIGURE 4.1: A BASIC EXAMPLE OF A THREE DIMENSIONAL FREQUENCY TABLE	33
FIGURE 4.2: MDFT NODE STRUCTURE	39
FIGURE 4.3 (A): EXAMPLE OF A CONVENTIONALLY LINKED DECISION TREE STRUCTURE	39
FIGURE 4.3 (B): DECISION TREE STRUCTURE IN WHICH THE NODES HAVE FIXED MEMORY USAGE BY EMPLOYING CHILD AND SIBLING POINTER.....	40
FIGURE 4.4: DESCRIPTION OF THE MDFT ALGORITHM	41
FIGURE 4.5: ATTRIBUTE OUTLOOK (A_1) IS CHOSEN FOR THE ROOT NODE.....	51
FIGURE 4.6: ATTRIBUTE TEMPERATURE (A_2) IS SELECTED FOR NODE 2.....	56
FIGURE 4.7: NODE 3 IS A LEAF NODE.	57
FIGURE 4.8: ATTRIBUTE WIND (A_4) IS SELECTED FOR NODE 4.....	62
FIGURE 4.9: NODE 5 IS A LEAF NODE.	63
FIGURE 4.10: ATTRIBUTE 'HUMIDITY' (A_3) IS SELECTED FOR NODE 6.	66
FIGURE 4.11: NODE 7 IS A LEAF NODE.	67
FIGURE 4.12: NODE 8 IS A LEAF NODE.	69
FIGURE 4.13: NODE 9 IS A LEAF NODE.	70
FIGURE 4.14: ATTRIBUTE WIND (A_4) IS SELECTED FOR NODE 10.	70
FIGURE 4.15: NODE 11 IS A LEAF NODE.	71
FIGURE 4.16: THE COMPLETED DT, WHERE NODE 12 IS A LEAF NODE.	72
FIGURE 4.17: DECISION TREE CALCULATION TIME OF THE EXAMPLE FOUR DIMENSIONAL MDFTs, EACH BAR SHOWING THE TOTAL NUMBER OF ATTRIBUTE VALUES IN EACH EXAMPLE.....	74
FIGURE 4.18: DECISION TREE CALCULATION TIME OF THE EXAMPLE EIGHT DIMENSIONAL MDFTs, EACH BAR SHOWING THE TOTAL NUMBER OF ATTRIBUTE VALUES IN EACH EXAMPLE.....	74
FIGURE 4.19: NUMBER OF ADDITIONS NEEDED AT THE ROOT NODE OF THE EXAMPLE FOUR DIMENSIONAL MDFT.....	76
FIGURE 4.20: NUMBER OF ADDITIONS NEEDED AT THE ROOT NODE OF THE EXAMPLE EIGHT DIMENSIONAL MDFT	76

FIGURE 4.21: MEMORY REQUIREMENTS OF THE MDFT FOR THE FOUR AND EIGHT DIMENSIONAL EXAMPLES	77
FIGURE 5.1: HFT STRUCTURE	80
FIGURE 5.2: DESCRIPTION OF THE HFTDT ALGORITHM	88
FIGURE 6.1: NUMBER OF NODES FOR THE NURSERY DATASET	99
FIGURE 6.2: NUMBER OF NODES FOR THE AGARICUS-LEPIOTA DATASET	99
FIGURE 6.3: NUMBER OF NODES FOR THE CHESS DATASET	100
FIGURE 6.4: EXECUTION TIME FOR THE NURSERY DATASET	101
FIGURE 6.5: EXECUTION TIME FOR THE AGARICUS-LEPIOTA DATASET	102
FIGURE 6.6: EXECUTION TIME FOR THE CHESS DATASET	102
FIGURE 6.7: MEMORY USAGE FOR THE NURSERY DATASET.....	104
FIGURE 6.8: MEMORY USAGE FOR THE AGARICUS-LEPIOTA DATASET	104
FIGURE 6.9: MEMORY USAGE FOR THE CHESS DATASET	105
FIGURE 6.10: CLASSIFICATION ERROR FOR THE NURSERY DATASET	106
FIGURE 6.11: CLASSIFICATION ERROR FOR THE AGARICUS-LEPIOTA DATASET	107
FIGURE 6.12: CLASSIFICATION ERROR FOR THE CHESS DATASET	107
FIGURE 7.1: EXAMPLE OF INFORMATION PROVIDED BY A FLAT PROFILE GENERATED BY GPROF FOR THE NURSERY DATASET	111
FIGURE 7.2: FLAT PROFILE RESULTS FOR THE NURSERY DATASET	112
FIGURE 7.3: FLAT PROFILE RESULTS FOR THE AGARICUS-LEPIOTA DATASET	113
FIGURE 7.4: FLAT PROFILE RESULTS FOR THE CHESS DATASET	113
FIGURE 7.5: EXAMPLE OF A CALL GRAPH GENERATED BY GPROF FOR THE NURSERY DATASET.....	115
FIGURE 7.6: CALL GRAPH FOR HFTDT SHOWING THE CALL SEQUENCE AND THE EXECUTION TIMES OF THE FUNCTIONS	116
FIGURE 7.7: PROCESS OF HIGH-LEVEL LANGUAGE TRANSLATION TO HDL USING HLS TOOLS.....	117
FIGURE 7.8 OVERVIEW OF THE VIVADO HIGH-LEVEL SYNTHESIS DESIGN PROCESS	119
FIGURE 7.9 CONCEPT OF C VERIFICATION	120
FIGURE 7.10: XILINX 7-SERIES FPGAs FAMILIES [109]	122
FIGURE 7.11: CODE FOR THE ACCELERATED FUNCTION FOR THE AGARICUS-LEPIOTA DATASET	123
FIGURE 7.12: HARDWARE ACCELERATION RESULTS OF TARGETING A RANGE OF FPGAs FOR THE NURSERY DATASET	124
FIGURE 7.13: HARDWARE ACCELERATION RESULTS OF TARGETING A RANGE OF FPGAs FOR THE AGARICUS-LEPIOTA DATASET.....	126
FIGURE 7.14: HARDWARE ACCELERATION RESULTS OF TARGETING A RANGE OF FPGAs FOR THE CHESS DATASET.	127
FIGURE 7.15: HARDWARE ACCELERATION RESULT COMPARED WITH C4.5 AND ITI FOR THE NURSERY DATASET.....	130
FIGURE 7.16: HARDWARE ACCELERATION RESULT COMPARED WITH C4.5 AND ITI FOR THE AGARICUS-LEPIOTA DATASET	130
FIGURE 7.17: HARDWARE ACCELERATION RESULT COMPARED WITH C4.5 AND ITI FOR THE CHESS DATASET.....	131
FIGURE 7.18: IMPLEMENTATION OF <i>REV_INDEX_FUNC</i> TO SUPPLY DATA TO THE OPEN NODES IN A DT.....	132

LIST OF TABLES

TABLE 2.1: SUMMARY OF MACHINE LEARNING METHOD SUITABILITY FOR REAL-TIME INCREMENTAL EMBEDDED LEARNING [11]	10
TABLE 3.1: THE APPLICATION PARAMETERS AVAILABLE FROM THE NAMES FILE	24
TABLE 3.2 AN EXAMPLE TRAINING SET, QUINLAN [69]	27
TABLE 3.3: (A) THE ATTRIBUTES AND CLASSES SHOWN IN THE ORIGINAL EXAMPLE AND (B) THE .NUMERICAL NOTATION AFTER CONVERSION.....	27
TABLE 3.4: THE VECTORS OF TABLE 3.2 AFTER CONVERSION TO NUMERICAL FORM	28
TABLE 4.1: THE WEATHER PROBLEM DATASET CONTAINS 16 INPUT VECTORS WITH 15 UNIQUE ENTRIES.....	42
TABLE 4.2: THE ATTRIBUTES AND CLASSES SHOWN IN THE WEATHER EXAMPLE WITH THE CORRESPONDING NOTATION.....	43
TABLE 4.3: MDFT SHOWS A UNIQUE NOTATION FOR EACH CELL ACCORDING TO ITS ATTRIBUTES AND CLASS VALUES.....	44
TABLE 4.4: MDFT FOR THE WEATHER PROBLEM, WHERE THE CELLS ARE UPDATED ACCORDING TO THE INPUT DATASET.....	44
TABLE 4.5: CLASSIFICATION (IF-THEN) RULES MODEL OBTAINED BY THE MDFT METHOD FOR THE COMPLETED DT SHOWN IN FIGURE 4.16.	72
TABLE 5.1: A MAP OF THE KEYS OF THE WEATHER PROBLEM	89
TABLE 5.2: HASHED FREQUENCY TABLE OF THE WEATHER PROBLEM	90
TABLE 5.3: COMPARISON BETWEEN THE HFTDT AND MDFT METHODS	92
TABLE 6.1: SUMMARY OF DATASET CHARACTERISTICS.....	97
TABLE 7.1: TARGETED FPGAs FOR THE HARDWARE DESIGN	121
TABLE 7.2: HARDWARE DESIGN REPORT FOR THE NURSERY DATASET	125
TABLE 7.3: HARDWARE DESIGN REPORT USING AGARICUS-LEPIOTA DATASET.....	126
TABLE 7.4: HARDWARE DESIGN REPORT USING CHESS DATASET	128
TABLE 7.5: HARDWARE SIMULATION RESULTS SUMMARY.....	128
TABLE A.1 CLASSIFICATION (IF-THEN) RULES MODEL OBTAINED BY THE HFTDT METHOD FOR THE COMPLETED DT SHOWN IN FIGURE 4.16.	170
TABLE C.1: HARDWARE SIMULATION RESULTS FOR NURSERY DATASET COMPARED WITH SOFTWARE RESULTS.	177
TABLE C.2: HARDWARE SIMULATION RESULTS FOR AGARICUS-LEPIOTA DATASET COMPARED WITH SOFTWARE RESULTS.	178
TABLE C.3: HARDWARE SIMULATION RESULTS FOR CHESS DATASET COMPARED WITH SOFTWARE RESULTS.	179

1 INTRODUCTION

Embedded systems are extensively used in many application domains, such as avionics, automotive systems, health monitoring and mobile communications. To maintain a competitive advantage, embedded system developers need to continually add new features, thereby increasing the computational demands on such systems. The market is one of substantial growth, forecast to expand at an annual rate of 6.8% between years 2012 and 2018 [1].

Embedded systems are increasingly being applied in applications that need to interact with the surrounding environment, requiring the ability to take correct and rapid decisions. Intelligent interaction with human counterparts requires an ability to learn in real-time (where the knowledge is accumulated and stored in an accessible form during the interaction) and incrementally (to augment existing stored knowledge). Where the targeted environment includes humans, safety issues will be paramount.

Systems that interact with humans are expected to behave intelligently and 'human like'. Such intelligent systems need to be able to adopt interaction modes that are natural to their human users as well as to understand their requirements. For intelligent systems to be able to interact with humans in all their environments implies that machine learning needs to become achievable on embedded platforms; a consideration still very rarely taken into account in the design of learning algorithms.

1.1 MOTIVATION

At present, machine learning systems are not available that are able to interact with human users in their normal living and working environments. The motivation of the work presented in this thesis is to make an initial contribution to providing such a system. The current work will design and implement a machine learning system that is capable of learning incrementally, in real-time and which is suited to embedded implementation. Decision trees (DTs) are adopted as a machine learning method, due to their simplicity, reliability and classification performance. Incremental DT learning methods have been considered by previous researchers, but the real-time embedded aspects have not been fully solved.

1.2 AIM AND OBJECTIVES

The aim in this research work is the design and implementation of novel machine learning algorithms suitable for embedded systems and to maximize its responsiveness and usefulness by adopting a real-time, incremental learning approach. This requires that the learning method is able to produce its outputs and knowledge updates within an acceptable time interval.

It is important to investigate the range of available candidate classifiers that can be used to implement a real-time embedded system. The work will show that an improved decision tree algorithm can give a substantial reduction in memory usage and calculation time can be achieved. In addition, the utilization of flexible hardware such as field-programmable gate arrays allows further acceleration of processing and permits larger problems to be addressed.

The following are the objectives to achieve the aim.

- Investigate the literature for existing learning methods that have potential to apply real-time incremental learning in embedded systems.
- Develop solutions for incremental learning in software simulation.

- Improve the solutions for real-time applications by providing a deterministic time solutions.
- Improve the memory management of the solutions to target embedded systems by maintaining a fixed memory usage.
- Evaluate the performance of the solutions with respect to other learning methods using a range of test databases.
- Implement a hardware solution for the most time-consuming part of the algorithm.

1.3 CONTRIBUTIONS TO KNOWLEDGE

In this research work, decision tree algorithms were developed that are suitable for real-time incremental learning systems. Below is a brief summary of the algorithms where a more detailed discussion can be found in chapters 3, 4 and 5.

Frequency table in multi-dimensional form

The multi-dimensional frequency table introduced in this thesis can hold in a compact form all the relationships between the attributes of the vectors in the training datasets. The frequency table has a number of dimensions equal to the number of attributes and is able to record the frequency of occurrence of each input vector.

Multi-dimensional frequency table algorithm (MDFT) as a real-time incremental learning method

The MDFT is a novel real-time incremental learning decision tree algorithm that employs a multi-dimensional frequency table and provides the ability to build a decision tree suitable for embedded systems. The MDFT algorithm can meet embedded system constraints by having a defined calculation time and memory usage.

Hashed frequency table decision tree algorithm (HFTDT) as a real-time incremental learning method

The HFTDT algorithm is a development of the MDFT algorithm. It also adopts a multi-dimensional Frequency table technique but stores the table values in a more compact form, allowing the building of decision trees for datasets larger than these that can be handled by MDFT. The algorithm uses a hash table to store the multi-dimensional frequency table, providing a unique key for each vector stored. The new HFTDT algorithm is able to produce similar classification performance to the MDFT method, but with reduced memory requirement.

1.4 THESIS ORGANIZATION

Chapter 2 presents background material, including an overview of machine learning systems, real-time embedded systems and real-time incremental learning systems. The chapter concentrates on decision tree approaches, including non-incremental and incremental learning algorithms as well as their implementation in software and hardware.

Chapter 3 presents the experimental procedure of the research work used for the generation of application-specific software and hardware to implement the MDFT and HFTDT algorithms.

Chapter 4 concentrates on the description, implementation and testing of the MDFT method. The chapter demonstrates the real-time, incremental and embedded nature of the MDFT approach.

Chapter 5 introduces the HFTDT method and demonstrates its implementation and testing. It is demonstrated to be a real-time incremental learning decision tree method, using a hashed frequency table to store incoming input data vectors.

Chapter 6 discusses the results of testing the HFTDT method by means of a series of examples and compares its performance with other learning methods, namely C4.5, kNN and ITI.

Chapter 7 describes the hardware implementation of HFTDT, where a custom hardware design of the most time consuming function is implemented to improve performance.

Chapter 8 presents the main findings of the thesis and gives suggestions for future research.

1.5 SUMMARY

An important development of modern electronic devices would be their ability to respond intelligently to their owners and on a time scale that is natural. A suitable embedded platform for such a device would be a real-time incremental machine learning system. This thesis investigates the most promising machine learning algorithm among those already existing and works towards its adoption for use in embedded systems.

2 BACKGROUND

This chapter introduces a range of machine learning methods that are described in the literature and which are relevant to the current work. The chapter first investigates machine learning in general, but concentrates on the real-time incremental learning systems that are the most relevant to the current work. As a result of this general investigation, decision tree methods are identified as the most relevant for real-time incremental learning in embedded system. Consequently, the second part of this chapter concentrates on existing techniques used for decision tree implementation, including both incremental and non-incremental approaches, the splitting criteria used for tree generation, pruning techniques and a review of the implementations of decision trees in electronic hardware.

2.1 MACHINE LEARNING SYSTEMS

Several methods and algorithms have been developed by researchers in the field of machine learning with the aim of building intelligent systems that can complete certain tasks within a pre-defined time period. Machine learning plays a significant role in Artificial Intelligence (AI) as those systems that are capable of self change as further information is acquired [2].

Several authors (such as [3], [4]) consider learning to be the gain of knowledge or the refinement of skills. The Oxford English Dictionary [4] defines machine learning as “*Computing the capacity of a computer to learn from experience*”, so including any self modification that can occur as a result of obtaining the newly acquired information. Mitchell [5] considered machine learning to be the ability of computers to program themselves and to take advantage of the data flow rather than just performing its processing. Köpf [6] considered machine learning to be a field of concern for the study of algorithms that can improve automatically from experience. Generally, such changes do not modify the algorithm itself, but rather the changes are expected to be to the database.

The majority of machine learning methods require that the data from which to learn are provided in a structured format. These data are typically represented by a vector whose elements describe attributes specific to the learning problem being tackled. During training, the vectors are used to modify the algorithm's internal representation of the problem. Often, a second set of vectors is maintained for testing purposes to assess the performance of the learning approach.

Figure 2.1 shows the architecture generally adopted by machine learning systems. The system behaviour that transforms inputs to actions is the application of rules. It is important to emphasise that a machine learning algorithm uses the knowledge acquired to generate rules that are then updated as new knowledge is acquired. It may be insightful at this stage to explain that in the decision tree implementations described later in this thesis, the accumulated knowledge is represented as a frequency table and the machine learning algorithm generates a decision tree that has an equivalent representation as a set of rules.

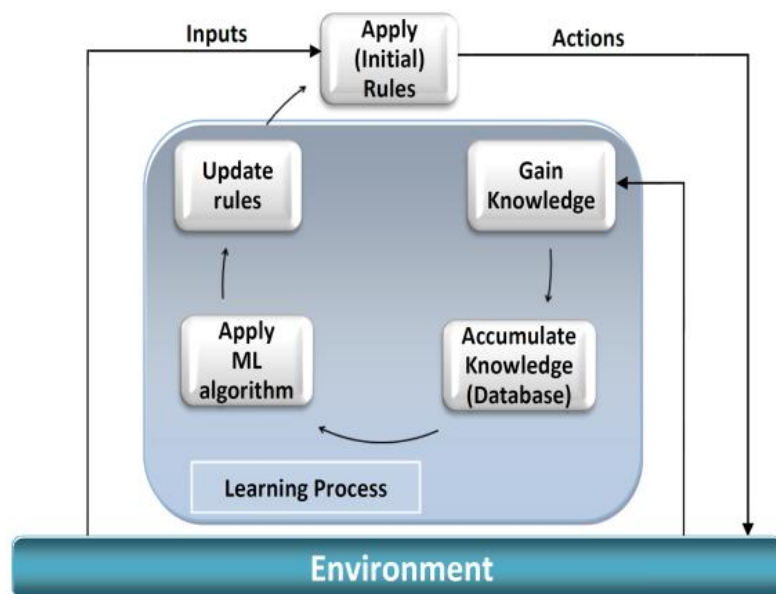


Figure 2.1: Machine learning system

2.1.1 REAL-TIME EMBEDDED SYSTEMS

Embedded systems are those dedicated to perform a specific task and are generally executed on a platform running an operating system that controls the microprocessor interface [7]. Real-time tasks are those characterized by a deadline, this being the maximum time needed by the system to complete execution [8].

Real-time systems are often further divided into those dealing 'hard' and 'soft' tasks [9]. Hard real-time tasks occur in highly critical applications, where missing one deadline may cause a catastrophic system failure, for example in automatic anti-lock braking systems or in air traffic control. In soft real-time systems, the missing of a deadline is normally possible and, although adversely affecting its performance, does not cause a major system failure, for example in an automated teller machine or a domestic appliance controller.

2.1.2 REAL-TIME INCREMENTAL LEARNING SYSTEMS

Although many embedded systems will need to operate in real-time to some extent, few can be categorised as incrementally learning. Non-incremental learning systems need to have a complete set of input data vectors available before learning can proceed. Incremental learning systems [10] have the ability to carry out continuous learning, in the sense that learning can start even if only one or a small number of vectors is available, but the rules can continue to be updated as more data are acquired [11]. A task can be considered as incremental if the learning system needs to be able to generate outputs before all inputs have been provided or assimilated [12]. As an example, incremental learning gives an intelligent robot the ability to navigate not only in fixed environments, but also in changing environments where existing objects may move or new objects be introduced. It is important to note that such navigation is likely to require that the learning system is able to operate in real-time as well as incrementally, since objects need to be avoided while the system continues to learn.

2.1.3 METHODS USED FOR MACHINE LEARNING

Learning methods can be supervised [13], unsupervised [14] or semi-supervised [15] algorithms. In supervised learning it is assumed that a teacher or supervisor trains the system, such that the instance examples are given with the desired categories. In unsupervised learning, the desired categories aren't provided as part of the training and the machine needs to be capable of generating and maintaining its own categories. In semi-supervised learning the system learns from a mixture of categorised and uncategorised training vectors.

In the design of an embedded system that performs real-time incremental learning, a number of constraints need to be met. To operate in an embedded environment, the algorithm needs to be capable of using a specified memory capacity that is known *a priori*. To be able to operate in real-time, the number and types of calculations that are to be performed during learning needs to be known, as well as the characteristics of the computing platform on which the algorithm is to be run. These learning operations will include integrating a new vector into the knowledge base and updating the rule set. The ability to operate incrementally depends on the learning algorithm itself, and specifically whether it has been designed to require the entire dataset is made available before learning can begin, or whether learning can progress with partial data. Table 2.1 compares the ability of the most popular categories of machine learning methods to operate in an embedded real-time incremental manner.

In instance-based learning, the learning process requires the storage of all training vectors as well as information relating to classification performance: the process is time consuming and substantial memory is required. For artificial neural networks and support vector machines, a large sample size is required to achieve the best prediction accuracy [13], thereby increasing the demand on memory and computation time and often making the methods unsuitable for real-time applications. In reinforcement learning and Bayesian learning, the memory requirement and long computation times limit their suitability for real-time applications. A small number of genetic algorithm approaches permit incremental learning, but all require an initially unknown computational time. Inductive

learning, which includes decision trees, is shown in the table as being potentially suitable for real-time incremental learning. However, as discussed in the following section, this suitability is highly dependent on the specific decision tree approach adopted and many have shortcomings in at least one aspect of being suited to real-time or incremental learning. The focus of the current work is to develop a decision tree method that is well suited to both real-time and incremental implementation in embedded systems.

Table 2.1: Summary of machine learning method suitability for real-time incremental embedded learning [11]

Method	Capable of operating in embedded devices	Capable of performing real time learning	Capable of performing incremental learning
Instance based learning [16]	Yes	No	No
Artificial neural networks [17], [18]	Yes	Yes	Yes
Support vector machines [19]	Yes	No	No
Reinforcement learning [20]	Yes	No	No
Bayesian learning [11], [21]	Yes	No	Yes
Genetic algorithm [22], [23]	Yes	No	Yes
Inductive learning DTs [11], [24]	Some	Partially	Some

2.2 DECISION TREES

Decision trees (DTs) are a hierarchical model generated under supervised learning which consists of internal decision nodes and terminal leaves identified by a sequence of recursive splits [2]. DT algorithms have been successfully exploited in classification domains, including pattern recognition, decision support systems, expert systems [11], [25] and applied in speech and character recognition, remote sensing and medical diagnosis [26]. The advantages include simplicity, relatively low computation complexity, accuracy, and they can produce a representation of what has been learned that is easy for their users to understand.

2.2.1 NON-INCREMENTAL AND INCREMENTAL LEARNING DT ALGORITHMS

Several DT algorithms have been developed, and can be divided into two categories, namely non-incremental learning algorithms (ID3 [24], C4.5 [27], C5 [28], CART[29], SPRINT[30] and ID6NB [31]) and incremental learning algorithms (ID4 [32], ID5 [33], ID5R [34], ITI [35], FTDT and IDP [11]).

The non-incremental C4.5 algorithm introduced by Quinlan [27] is an evolution of ID3 and uses dynamic memory allocation, a type of storage normally avoided in embedded applications due to the potential to exceed available storage capacity [36]. ID4, ID5 and ID5R produce binary trees, and, in spite of the fact that ID5 and its successor ID5R were developed to overcome the relatively poor classification performance of ID4, both produce results that are of similar quality to ID3 [11], [31], [37]. ID5 and ID5R require considerably more memory than their non-incremental counterparts, making these algorithms poorly suited for embedded targets.

ITI (Incremental Tree Inducer) [35] exhibits performance often comparable with C4.5 and better than ID4, ID5 and ID5R [11]. The memory usage and computation time of ITI increase with the number of training vectors, making this algorithm an unlikely candidate for embedded systems [38].

2.2.2 FTDT AND IDP INCREMENTAL LEARNING METHODS

The shortcomings of the DT approaches found in the literature were identified by Swere [11]. Initial work was carried out in order to identify suitable DT enhancements that would meet the time and memory performance constraints for real-time incremental learning, yet be able to provide similar classification performance to C4.5. The FTDT (frequency table decision tree) method and the IDP (incremental decision path) were both based on a novel frequency table approach that was originally developed to allowing a fixed memory and known DT calculation time to be defined; features not apparent in ID5, ID5R and ITI.

In generating DTS, both FTDT and IDP perform information gain (IG) [24] and entropy [39] calculations similar to those found in C4.5. As all the information needed to perform DT calculations use data found in the frequency table (FT) for a given application, an upper bound on the number of calculations can be determined in advance. As the dimensions of the FT are also known for a given problem, the memory requirement can also be determined *a priori*.

A common drawback in incremental learning algorithms arises when more memory is needed to store the additional data obtained as time progresses [35], an issue overcome in FTDT and IDP [37]. Swere *et al.* [38] found that the calculation time needed to generate the nodes using the frequency table was typically an order of magnitude less than that required by ITI. However, the major drawbacks of the FTDT and IDP methods are that they are only able to generate partial decision trees and consequently are unable to solve only problems with a small number of attributes or, for larger problems, generate solutions whose classification performance is significantly inferior to that achievable by C4.5.

Figure 2.2 shows the FTDT algorithm. By recording in a FT the relationship of an attribute with all other attributes, limited information which is only sufficient to build a partially decision tree is available. In most DTs, additional information is required to build accurately trees of depth greater than two, as the correlation of more than two attributes is needed.

Input: Frequency table, a training vector	
Outputs: A new DT and commensurate rules	
Start	
1.	Update the frequency table following the arrival of a new training vector
2.	IF all the entries in the frequency table are of the same class THEN produce a DT containing a single terminal node of the class, go to step 7 ELSE initialize the DT to include a single root node
3.	Select a non-terminal node by following a path through each of the attribute values of a previous node
4.	IF all the frequency table entries for this node are of the same class THEN label the node as a terminal node of that class ELSE compute the entropy value and the IG for all attributes, select the attribute of the largest IG and use this attribute for the current node
5.	Add edges from the node for each of its attribute values
6.	If there remains nodes in the DT that are not terminal nodes, then go to step 2
7.	Generate rules from the DT
8.	Store the DT and the DT rules
9.	Use the DT as required and store new training vector
End	

Figure 2.2: FTDT algorithm [11]

The IDP algorithm shown in Figure 2.3 is a development of the FTDT algorithm which aims to reduce the time needed to classify vectors. IDP generates only that single branch of the DT that is needed to classify a given test vector. This particular approach is best suited to incremental learning applications where both training and testing need to be carried out concurrently.

A drawback is that pruning cannot be performed on an IDP tree (section 2.2.4) as the whole tree is not available.

Input: Frequency table and unclassified vector	
Output: Classified vector	
1.	IF all the entries in the frequency table are of the same class THEN produce a DT containing a single terminal node of the class, go to step 5 ELSE initialize the DT to include a single root node
2.	compute the entropy value and the IG for all attributes, select the attribute of the largest IG and use this attribute for the current node
3.	Select the attribute value at the node that matches the attribute value of the corresponding attribute in the unclassified vector
3.	IF all the frequency table entries for this node are of the same class THEN label the node as a terminal node of that class, go to step 5 ELSE compute the entropy value and the IG for all attributes, select the attribute of the largest IG and use this attribute for the current node, go to step 3
5.	Use the decision branch as required, update the frequency table
6.	Record new classified vector and return to step 1

Figure 2.3: IDP algorithm [11]

2.2.3 ENTROPY, INFORMATION GAIN AND GAIN RATIO

Criterion that have been used to ‘split’ data at the nodes of DTs include information gain, gain ratio [24], Gini index [40] and the Towing Rule [41]. As information gain and gain ratio are by far the most popular, only these two criteria are described here.

Entropy calculations are performed to determine the information gain. The entropy E of a set of probabilities p_1, \dots, p_n is described by Shannon [42] as

$$E = - \sum_{i=1}^n p_i \log_b p_i . \quad (2.1)$$

The quantity of entropy measures the amount of information. The choice of the unit for measuring information depends on the base b of the log. For base 2 the resulting units of the measures are bits and for base 10 it is decimal. To change from base b to base a simply requires multiplication by $\log_a b$.

In DTs, two entropies need to be measured in order to calculate the information gain of each attribute. The information gain compares the entropy of each attribute with the total entropy of the dataset involved in the calculations at a given node. The attribute with the largest information gain value is then selected for that node.

The total entropy of the dataset is given by

$$E_T = - \sum_{i=1}^{N_c} \frac{Q_{C_i}}{Q_{C_T}} \log_b \left(\frac{Q_{C_i}}{Q_{C_T}} \right), \quad (2.2)$$

where N_c is the number of classes, Q_{C_i} is the number of input vectors (entries) under class C_i and C_T are the classes involved at that node.

The entropy E_V of an attribute value j at a given node is then

$$E_V(j) = - \sum_{i=1}^{N_c} \frac{Q_{C_i^j}}{Q_{C_T^j}} \log_b \left(\frac{Q_{C_i^j}}{Q_{C_T^j}} \right), \quad (2.3)$$

where $Q_{C_i^j}$ is the number of input vectors in class C_i with attribute value j and $Q_{C_T^j}$ are the classes available for attribute value j .

From Equation 2.3, the entropy for all the values of an attribute A_k can be determined from

$$E(A_k) = \sum_{j=1}^{N_v[A_k]} E_V(j) \quad (2.4)$$

where j are now specifically the attribute values in A_k and $N_v[A_i]$ is the number of values of A_k .

The information gain for an attribute A_k can then be determined by

$$IG(A_k) = E_T - E(A_k). \quad (2.5)$$

The information gain method in some cases tends to favour attributes with a larger number of values, adversely affecting performance compared to using the gain ratio criteria [27] which measures the ratio between the information gain of an attribute and its information content. The information content for an attribute A_k is given by

$$E_{split}(A_k) = - \sum_{j=1}^{N_v[A_k]} \frac{Q_j}{Q_T} \log_b \left(\frac{Q_j}{Q_T} \right) \quad (2.6)$$

Where Q_j is the number of input vectors of attribute A_k with value j and Q_T is the number of input vectors in the dataset available for node calculation.

The gain ratio for attribute A_k at a node can then be found from

$$GR(A_k) = \frac{IG(A_k)}{E_{split}(A_k)} \quad (2.7)$$

2.2.4 DECISION TREE PRUNING

This section describes the methods for controlling decision tree growth, which is normally achieved either by applying a stopping criterion or by adopting a pruning method. The main pruning techniques used are pre-pruning, that controls the branch growth of the tree at the growing stage and post-pruning, that performs pruning on a fully grown tree.

Stopping criteria

Kotsiantis [43] listed the popular conditions that have been used to trigger a stopping criterion during the growth phase of the decision tree. Conditions include when all vectors in the training set belong to a single class, decision tree growth

reaching a pre-defined maximum depth, the number of cases in a terminal node being fewer than a minimum number of cases of its parents, the number of cases in new child nodes being fewer than a certain minimum, or when the largest splitting criteria value is less than a preset threshold.

Maimon *et al.* [44] found that choosing stringent stopping criteria can lead to a small and underfitted decision tree with poor classification accuracy for both training and test data, whereas choosing loose stopping criteria tend to generate larger decision trees which suffers from overfitting. Overfitting of data occurs when the generated decision tree becomes significantly dependent either on features not of utmost importance is distinguishing between classes on irrelevant features or on data noise in the training dataset [45]. Such dependency can result in a poor classification performance when dealing with unseen data.

Pruning

To mitigate against the effects of overfitting, pruning techniques are used to generate DTs that produce a representation of the data that is more generalized and so likely to perform better in the classification of test vectors. Pruning is normally achieved by removing sub-trees from the DT that have been generated by relatively few vectors in the training data [46]. The two most commonly-applied pruning techniques are pre-pruning and post-pruning [47]. Pre-pruning uses stopping criteria to halt the growth of the DT before it completely represents the training data. Stopping criteria include the chi-squared test used by Quinlan [24], Fisher's exact test [48], the statistical significant method [49] and the depth limit of cost-sensitive DTs [50]. Post-pruning is carried out after the completion of DT generation. It removes branches (or sub-trees) of the DT with the aim to improving generalization and hence classification performance when tested with unseen vectors [43]. Pruning can begin either from the root node and proceed towards the leaves or from the leaves and progress towards the root [51]. Post-pruning algorithms proposed in the literature include minimal cost complexity pruning [29], reduced error pruning [52], minimum error pruning [53], critical value pruning [54], pessimistic error pruning and error based pruning [27]. Esposito *et al.* [51] and Aha *et al.* [47] have published comparative studies of post-pruning

methods; their results indicate that cost complexity and reduced error pruning tend to over-prune, which leads to smaller DTs but reduces classification performance. They found that the other methods listed above tended to perform under-pruning, producing larger trees, but again adversely affect classification performance relative to an ideally-sized DT.

Pre-pruning is often computationally more efficient as it can avoid the building of an entire irrelevant branch, whereas post pruning requires that the branch is built first before its usefulness is considered [55]. Conversely, however, pre-pruning may stop tree growth too soon before all the information is available [29], in contrast to post-pruning that needs access to the complete DT as it operates on a fully grown tree. A more comprehensive description of pre-pruning methods can be found in a study conducted by Frank [55].

2.2.5 HARDWARE IMPLEMENTATIONS OF DECISION TREES

DTs have a low computation complexity when compared to many other machine learning techniques. For large datasets, or when input data is continually streamed, achieving the rapid generation of DTs requires the development of new algorithms or the acceleration of existing ones. Parallelization can be used to speed up the process of building classification trees and is feasible where parts of the DT are sufficiently computationally separate that their calculations can be carried out independently. Steinhäuser *et al.* [56] proposed a parallel implementation of ID3 by using separate threads to execute independent DT algorithm paths. A significant performance advantage compared with the serial version, especially when the number of vectors and the number of attributes is large was demonstrated. The parallel mode of computation was executed on the Cray multi-threaded architecture.

Kufrin [57] proposed a data distributed parallel formulation ID3 which relies on gathering the statistics of the attribute values and classes for each node. Such statistics can be shared and processed in parallel on a small number of processors in order to determine the best attribute for splitting at a node.

Srivastava *et al.* [58] proposed three parallel formulations, namely synchronous tree construction, partitioned tree construction and a hybrid of the synchronous and partitioned formulations. In the synchronous approach, all processors operate at the same node and each calculates the entropy gain of a different set of attributes. Each processor receives and transmits class distribution information simultaneously. In the second approach, independent processors each work on their own parts of the classification tree, following a succession of nodes as they are expanded. The hybrid formulation has elements of both the synchronous and partitioned approaches in an attempt to overcome their drawbacks. In its implementation, the hybrid scheme follows the synchronous approach as long as the communication cost is not too high, in which case it switches to the partitioned approach.

In the parallelization approach proposed by Narlikar [59], two levels of divide-and-conquer parallelism were considered in the building of C4.5 DT. An outer level extended across tree nodes and an inner level operated within a tree node. The author proposed using a lightweight thread implementation for C4.5 algorithm to take advantage of its intrinsic parallel nature.

Jin *et al.* [60] proposed a new approach to decision tree construction, named SPIES, in which the numerical attributes of a large number of distinct elements were divided into intervals of equal width and a class histogram of the frequency of occurrence of each class was computed. The algorithm computed a subset of candidate split point values for the numerical attributes and stored the class histogram for the points to reduce the space complexity of the algorithm and the communication cost between the processors. The algorithm was parallelized using the FREERIDE framework and the authors obtained almost linear speedups.

Yildiz *et al.* [61] presented three types of parallelism and applied these to two DT algorithms, the C4.5 and the univariate linear discriminate tree. The authors examined the computational effects of realizing DTs using feature-based parallelism, node-based parallelism and data-based parallelism. In the feature-based approach, the attributes under investigation were allocated to separate processors and the results combined to isolate the attribute giving best split. In the

node-based approach, the computations at the nodes themselves were allocated to separate processors. The node-based approach maintained a queue of nodes from where they were extracted and sent to the slave processors. In the data-based approach, the dataset was portioned according to attribute and only the relevant data sent to the relevant processor. The experimental results produced by the study showed that the performance improvement produced by the parallel implementation depended greatly on the number of attributes and attribute values in the dataset. The node-based parallelisation approach demonstrated a good speedup.

Narayanan *et al.* [62] proposed an FPGA implementation for a DT system providing binary classification. The architecture implemented the Gini score calculation in which the impurity computation is the most computationally intensive part of the learning process. A five-fold improvement in calculation time was demonstrated when using a fixed-point integer solution relative to that achieved by a software solution using two embedded PowerPC processors and implemented using floating point operations.

Zaki *et al.* [63] presented a parallel DT algorithm for an symmetrical multi-processor (SMP) system using the SPRINT algorithm. They proposed task parallelism using dynamic subtree partitioning and data parallelism based on attribute scheduling among threads running on the SMP processors. Andrade *et al.* [64] considered clusters of SMPs running under Linux, where each processor is allocated a portion of a partitioned dataset. Steihauser *et al.* [56] found that the main disadvantages of SMP systems is that only one processor can access memory at a time, a considerable restriction in the construction of DTs, which is highly memory intensive operation.

2.3 CONCLUSION

The work presented in this thesis aims to provide an implementation of a real-time incremental embedded machine learning system. A comparison of different machine learning methods showed that DTs and neural networks are capable of performing real-time incremental learning. However, as neural networks have the disadvantage of requiring a large memory capacity, they are not suitable for implementation in embedded systems with limited memory and computational resources.

Following the investigation of machine learning systems, DTs were identified as the most appropriate for real-time incremental implementation. Consequently, this chapter has concentrated on DT systems and described the calculations used as splitting criteria such as entropy and information gain.

A common drawback with decision tree algorithms is overfitting, and there is a range of techniques available to reduce its effects including adopting stopping criteria or by applying pruning. In the work in this thesis, no stopping criteria or pruning were applied as the effects of these approaches in real-time incremental systems require extensive work beyond the main thrust of the current focus. However, stopping criteria and pruning are not undesirable and their implementation should be the subject of further work. However, the reader should be aware that the results presented in this thesis for the new approaches may sometimes be affected by the absence of stopping criteria and pruning.

The hardware approaches described in the literature have been reviewed. A number of parallel approaches are available that have concentrated on improving performance in a range of aspects of building decision trees, such as attribute selection, node expansion and data parallelism. The inherently parallel nature of DT algorithms is highly suited to hardware exploitation. Although the DT algorithms developed in this work are not themselves implemented in hardware, the method to accelerate the approach used to store training vectors and so enable incremental learning are realised in hardware.

3 EXPERIMENTAL PROCEDURE

This chapter describes the process that has been developed for the implementation of the new decision tree approaches. Although a wide variety of practical example applications can be realised by the new approaches, the process generates a software implementation that is specialized for the example application of interest, both at the training and classification stages. Such an approach has advantages when targeting embedded environments in which execution time and memory resources need to be carefully controlled.

3.1 INTRODUCTION

By exhibiting the features of both a defined memory requirement (see Sections 4.6 and 5.4.2) and a maximum computation time per training vector, the novel real-time incremental decision tree algorithms developed in the current research work are suitable for embedded real-time implementation. This is in contrast to the decision tree algorithms described in the literature, such as C4.5 [27] and ITI [35], which are general purpose algorithms that run on computer workstations and allocate memory in a dynamic manner.

In embedded target environments, dynamic memory may not be available and execution resources are likely to be at a premium. Consequently, in this thesis, the software implementation of the decision tree approaches is designed to exhibit the following features.

- No translation between external and internal data representations are needed.
- All memory for the decision tree implementation can be allocated statically.
- Offsets to elements in array structures are known at compile time.

To ensure that the above features are incorporated, the current research generates bespoke decision tree training and classification programs according to the

requirements of the application being considered. This allows the memory architecture to be specified statically before learning takes place.

This thesis will introduce two novel decision tree approaches, namely the Multi-Dimensional Frequency Table (MDFT) and the Hashed Frequency Table Decision Tree (HFTDT). The two approaches will be described in detail in later chapters (MDFT in Chapter 4 and HFTDT in Chapter 5). The purpose of this chapter is not to describe the new approaches, but instead to describe the process by which the software that implements them is produced.

The MDFT and HFTDT methods adopt a frequency table technique to hold the training data necessary to build their decision trees. The advantage using of a frequency table is that it permits an incremental approach to learning that has the potential for implementation in a real-time embedded system. Although earlier approaches (FTDT and IDP [11], [37]) also used frequency tables, both were only able to generate partial decision trees. The MDFT and HFTDT methods, which represent the core of this research work, successfully incorporate the frequency table technique to produce a complete decision tree.

3.2 MDFT AND HFTDT STRUCTURE

MDFT and HFTDT were designed with the ability to generate bespoke code for a given specific application. Figure 3.1 shows that a separate ‘generate’ program is used to produce the ‘training’ and ‘classification’ programs. The programs are discussed in detail in the following sub-sections.

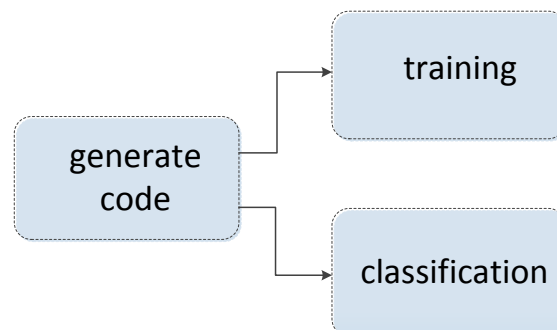


Figure 3.1: The programs involved in the implementation of the MDFT and HFTDT approaches

3.2.1 GENERATE DECISION TREE CODE

The program that generates target code optimized for a specific application is shown in Figure 3.2. Although no standard format is available for describing the structure of data to be used for training or that require classification, common practice is to make available the salient parameters in a ‘names’ file that is defined separately from the data itself. Table 3.1 shows the parameters that can be found in such a names file.

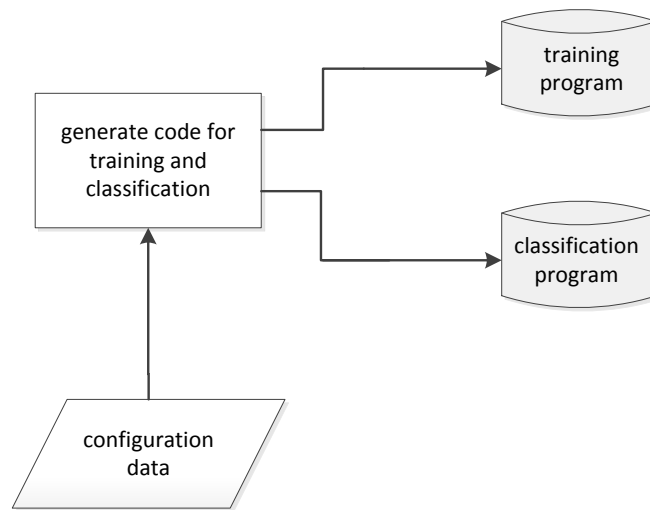


Figure 3.2: The generate program

Table 3.1: The application parameters available from the names file

Parameter	Description
N_c	Number of classes
C_n	Class names
N_a	Number of attributes
A_n	Attribute names
$N_v[N_a]$	Number of values for each attribute
$V[N_a][N_v[N_a]]$	Attribute value names

The design of the frequency table and the main functions of the code depend on the parameters given by the names file. The main code includes functions to perform entropy and information gain calculations, fetching, building the decision tree and for performing classification. The number of iterations of the loops in the decision

tree calculations and the values generated for a lookup table are determined automatically according to the given parameters of the names file.

3.2.1.1 Generating code for training and classification

The number of dimensions of the frequency table is determined by both N_c and N_a , where the conversion process of input data vectors into numerical notation and all nested loops in the generated code depend on the parameters N_c , N_a and $N_v[]$.

The training code consists of several functions that are generated according to the configuration data of the targeted system, covering the following.

- Multi-dimensional frequency table: read/write functions.
 - The read function incorporates a search operation to read values saved in the multi-dimensional frequency table. Both MDFT and HFTDT include nested loops whose number of iterations is set according to the number of classes (N_c), the number of attributes (N_a) and the number of attribute values ($N_v[]$).
 - The write function updates the multi-dimensional frequency table on the arrival of each new training vector. It includes several *for* loops and *if* statements. The loop iterations and *if* statements depend on the number of classes (N_c), the number of attributes (N_a) and the number of attribute values ($N_v[]$).
 -
- Indexing and reverse indexing functions in the HFTDT method.
 - The HFTDT method includes an indexing function that is used to generate a unique index for each input vector to be stored in the hashed multi-dimensional frequency table. The index function depends on the number of classes (N_c), the number of attributes (N_a) and the number of attribute values ($N_v[]$).
 - The reverse index function used by HFTDT forms the complementary operation to the indexing function in that it generates the vector from its index. It also depends on the parameters provided by the configuration data, namely as the number of classes (N_c), the number of attributes (N_a) and the number of attribute values ($N_v[]$).

- Logarithm table.
 - A generated logarithmic lookup is used to implement entropy calculations. Two approaches for choosing the logarithm base were used in this research work. The first is to use Logarithms to the base 2 as adopted regardless of the number of classes involved in a calculation in the implementations of Quinlan [27]. The second is to use a logarithm base that is equal to the number of classes, allowing a normalising of the entropy values. The alternative approaches were tested on a range of examples and were found to give same classification results.
 - The logarithmic lookup table has the advantages of having a shorter execution time [65][66] and reducing hardware realization complexity [67].
 - The logarithmic lookup table was also used in this research work to allow the generation of hardware implementations. The logarithmic values were generated with a precision of 10^{-5} , allowing the resolution equivalent to one vector out of a hundred thousand. This was sufficient to handle all the example applications tested in this research work. A practical test was conducted and found that the number of generated was identical whether the logarithmic lookup table or the GNU mathematical library [68] was used.

3.2.2 DECISION TREE TRAINING

The decision tree training process is shown in Figure 3.3. On application of a new input data vector to the decision tree training program, the original text data is converted into a numerical notation beforehand to save execution time when running the decision tree code.

An illustrative example will now be given of the conversion process being performed on an example from Quinlan[69] that has three attributes two classes, as shown in Table 3.2.

Table 3.2 An example training set, Quinlan [69]

Number	ATTRIBUTES			CLASS
	Height	Hair	Eyes	
1	Short	Blond	Brown	Negative
2	Tall	Dark	Brown	Negative
3	Tall	Blond	Blue	Positive
4	Tall	Dark	Blue	Negative
5	Short	Dark	Blue	Negative
6	Tall	Red	Blue	Positive
7	Tall	Blond	Brown	Negative
8	Short	Blond	Blue	Positive

The conversion into numerical notation can be seen in Table 3.3, where part (a) represents the original names file and (b) the equivalent numerical notation for this example.

Table 3.3: (a) The attributes and classes shown in the original example and (b) the numerical notation after conversion

CLASSES		
Negative	Positive	
ATTRIBUTES		
HEIGHT	HAIR	EYES
Short	Blond	Brown
Tall	Dark	Blue
	Red	

(a)

CLASSES		
0	1	
ATTRIBUTES		
0	1	2
0	0	0
1	1	1
	2	

(b)

Following conversion, the vectors are as shown in Table 3.4.

Table 3.4: The vectors of Table 3.2 after conversion to numerical form

Number	ATTRIBUTES			CLASS
	Height (0)	Hair (1)	Eyes (2)	
1	0	0	0	0
2	1	1	0	0
3	1	0	1	1
4	1	1	1	0
5	0	1	1	0
6	1	2	1	1
7	1	0	0	0
8	0	0	1	1

In order to select an attribute for each node of the decision tree, entropy and information gain calculations need to be carried out. After completing the decision tree, rules that represent the knowledge learned are extracted from the decision tree for use in classification.

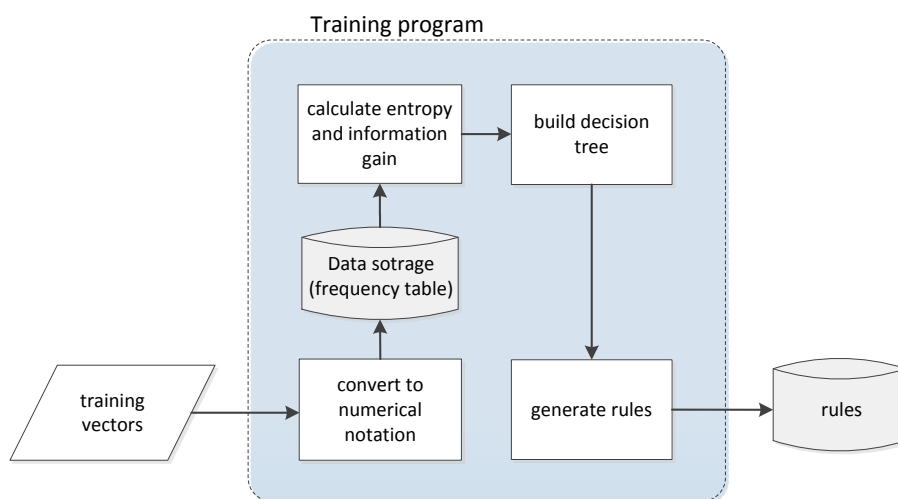


Figure 3.3: The program to train the decision tree

Depending on whether the MDFT or the HFTDT method is being used, the converted vector is stored in a frequency table using one of two available techniques, as shown in Figure 3.4. Both algorithms use an incremental approach and during training they are capable of producing an updated decision tree after receiving each new input data vector. To store the frequency table the MDFT technique uses a multi-dimensional array, while the HFTDT method uses a hashed table.

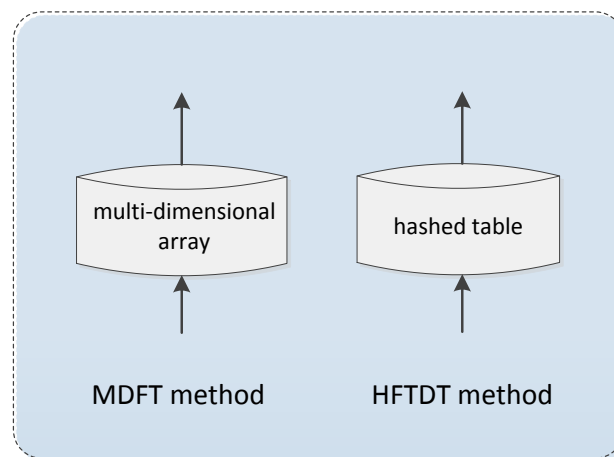


Figure 3.4: Two different approaches used for storing the training data in a frequency table

3.2.3 CLASSIFICATION USING THE DECISION TREE

The classification program is shown in Figure 3.5. As in the training program, the input data are converted to a numerical notation readable by the program, but in the classification program an output decision is then made regarding the class to the converted vectors based on the stored rules.

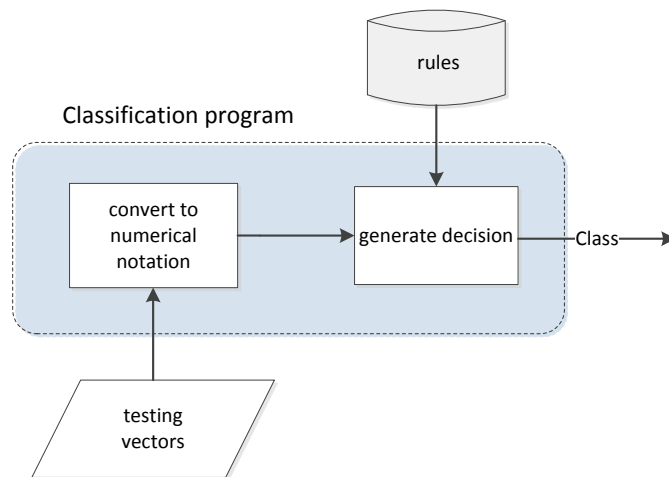


Figure 3.5: Classification using the decision tree

3.3 SUMMARY

In order to produce a decision tree that can be implemented on an embedded target system, including a hardware based solution which has the requirement of statically-defined memory, an implementation process has been defined that generates both training and classification programs that are tailored to a particular learning application.

With the embedded system development process in place, the next two chapters concentrate on the description, implementation and testing of the two new decision tree approaches introduced in this thesis, namely the Multi-Dimensional Frequency Table and the Hashed Frequency Table Decision Tree.

4 MULTI-DIMENSIONAL FREQUENCY TABLE DECISION TREE

This chapter presents a novel real-time incremental learning decision tree approach. The new approach is named the Multi-Dimensional Frequency Table (MDFT) method. The MDFT has been designed to target embedded systems where the number of calculations and the memory requirement needed to generate a DT can be known *a priori*. The chapter firstly includes an introduction to frequency tables and the use of MDFT for decision trees. Secondly, it presents the novelty of the approach and the implementation of the MDFT algorithm. Thirdly, a demonstration of a simple example using the MDFT method is given, including the calculations that have been conducted on each node of the decision tree. Lastly, the chapter presents the results of conducting several test experiments leading to the motivation to develop a successor to MDFT, termed HFTDT.

4.1 INTRODUCTION

Embedded systems generally have limited resources, principally those relating to memory capacity and computation capabilities. The new approach of MDFT is designed to operate within the aforementioned constraints in the realisation of a real-time incremental learning system. The MDFT method adopts a multi-dimensional array that acts as a frequency table, thereby bounding memory usage which can be known *a priori*, as it holds all the iterations of the incoming data vectors, while keeping all correlations between the attributes and classes. Existing decision tree incremental learning methods generally lack the ability to execute in bounded memory as they retain the input training dataset in the nodes of the tree.

4.2 FREQUENCY TABLE FOR DECISION TREES

Learning embedded systems need to have the ability to act within their environment by the application of accumulated knowledge. In this research work, the Frequency Table (FT) [11], [37], [38], [70] (or frequency distribution) maintains an organized summation of the number of occurrences of specific input data vectors. The FT provides a compact version of the input data vectors, in which the correlations between all the attributes are retained for each class.

The FT developed in this work is a multi-dimensional table. The dimensions of the FT depend on the size of the problem in term of attributes, attribute values and classes, their values determining the dimensions of the FT for a specific problem. The number of dimensions of the frequency table for any problem is equal to the number of attributes, with an additional dimension being required to represent the class.

The structure of the FT is not limited to a specific number of dimensions and it is tailored to the characteristics of the application presented. The general structure of the multi-dimensional FT can be described in the form $\{A_1, A_2, \dots, A_{N_a}, C\}$ where each element in the set represents a dimension. There are $(N_a + 1)$ dimensions, divided into N_a attributes $\{A_1, A_2, \dots, A_{N_a}\}$ and an additional dimension for the class of the vector, where C represents one of the N_c classes given as $C = \{C_1, C_2, \dots, C_{N_c}\}$

To illustrate the concept of an MDFT, Figure 4.1 gives a sample example of a three dimensional FT where the number of attributes N_a is 2 and the number of classes is given by N_c . The set of attributes in the example given are $\{A_1, A_2\}$. Each attribute represents a dimension for the FT, where the third dimension is reserved for the class and the number of attribute values is defined as $N_v[0]$ and $N_v[1]$ for the attributes A_1 and A_2 respectively.

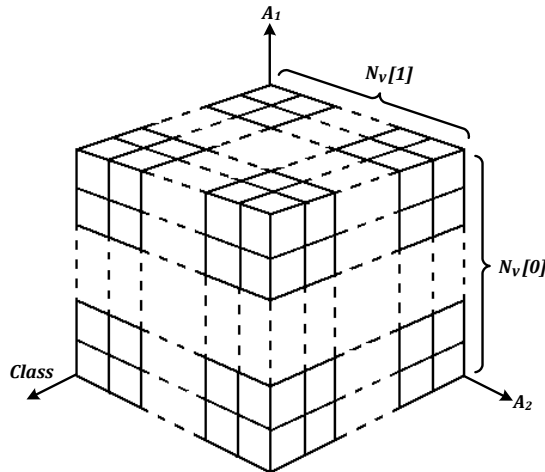


Figure 4.1: A basic example of a three dimensional frequency table

4.3 NOVELTY OF MDFT

The MDFT method presented on this research work is designed for the purpose of meeting the demands of real-time incremental learning applications of embedded systems. The demands can be generalised as follow.

- Known memory usage and its efficient management is crucial in real-time systems[71]. MDFT exhibits fixed usage and computable memory demands that can be calculated prior to the process of DT building.
- Incremental learning is emphasised when having good mechanism for holding old data and efficient management of data[72]. The MDFT technique keeps all correlations and relations of the saved data to improve predictability.
- Calculation time constraints of real-time applications plays a main roll in affecting prediction and performance of any method used. In terms of decision tree, the MDFT produces similar classification performance to C4.5 algorithm but with advantage of knowing in advance how many calculations needed to be done for a particular application.

4.4 DECISION TREE CALCULATIONS

Suitable algorithms for real-time applications should meet time deadlines in order to provide the required response. Therefore, knowing the time needed for performing calculations for any given application is crucial. The maximum time needed for the worst case scenario in generating the DT can be an important decision criterion in assessing the suitability of the algorithm to be used.

Decision tree calculations require that the computation of entropy and information gain are carried out at each node of the DT. A range of arithmetic operations are involved in such calculations. The aforementioned calculations need to be done for every node of the decision tree

Following the approach taken by FTDT [11], the number of calculations needed to determine the entropy and information gain (see section 2.2.3) for each attribute value in both MDFT and the HFTDT methods can be determined as below.

Entropy calculations

The number of additions required in an entropy calculation is equal to the number of classes less one

$$N_{ad}^E = N_c - 1 \quad . \quad (4.1)$$

where N_c is the number of classes in the dataset.

The number of multiplications is equal to the number of classes

$$N_{ml}^E = N_c \quad . \quad (4.2)$$

The number of divisions is equal to the number of classes

$$N_{dv}^E = N_c \quad . \quad (4.3)$$

The number of logarithmic calculations (or the reading of logarithmic values from a lookup table) is equal to the number of classes

$$N_{lg}^E = N_c \quad . \quad (4.4)$$

The number of entropy calculations for an attribute value is then equal to

$$N_V^E = N_{ad}^E + N_{ml}^E + N_{dv}^E + N_{lg}^E \quad . \quad (4.5)$$

The total number of entropy calculations for attribute A_i (where $0 < i \leq N_a$) is given by

$$N_A^E(A_i) = \sum_{j=0}^{N_v[A_i]} N_{V_j}^E \quad , \quad (4.6)$$

where $N_v[A_i]$ is the number of values for attribute A_i .

The entropy calculations can be then determined by summing the entropy calculations of every attribute involved with the node calculations with addition of the calculation involved in the overall entropy computation for the whole dataset involved with the node calculations.

The total number of entropy calculations for node k is

$$N_T^E(k) = \sum_{A_i \in A_k} N_A^E(A_i) + N_V^E \quad , \quad (4.7)$$

where A_k is the set of attributes used in the calculations for node k

Information gain calculations

The number of calculations needed to determine the information gain for an attribute is now determined.

The number of additions is equal to the number of attribute values

$$N_{ad}^{IG}(A_i) = N_v[A_i] , \quad (4.8)$$

where $N_v[A_i]$ is the number of values for attribute (A_i).

The number of multiplications is equal to the number of attribute values

$$N_{ml}^{IG}(A_i) = N_v[A_i] . \quad (4.9)$$

The number of divisions is equal to the number of attribute values

$$N_{dv}^{IG}(A_i) = N_v[A_i] . \quad (4.10)$$

The total number of calculations for an attribute A_i is then given by

$$N_A^{IG}(A_i) = N_{ad}^{IG}(A_i) + N_{ml}^{IG}(A_i) + N_{dv}^{IG}(A_i) . \quad (4.11)$$

The total number of information gain calculations for node k can be found from

$$N_T^{IG}(k) = \sum_{A_i \in A_k} N_A^{IG}(A_i) . \quad (4.12)$$

where A_k is the set of attributes used in the calculations for node k .

4.5 MAXIMUM NUMBER OF CALCULATIONS (WORST CASE SCENARIO)

This section determines the maximum number of calculations required to generate a DT, that is, the worst case scenario. DTs consist of a number of different levels starting from the root node (*level 1*) and ending in leaf nodes in (*level N_a*), where N_a is the number of attributes in the problem. The number of attributes values determines how many nodes are needed at each level. Where the numbers of values for each attribute are not all the same, the worst case for number of nodes in the decision tree occurs when levels are ordered such that the attribute with the largest number of values is chosen at the root and the last level contains the attribute with the least number of values

The following equation shows the number of calculations needed at the root node

$$N(1) = \sum_{A_i=1}^{N_a} \left(N_A^{IG}(A_i) + N_A^E(A_i) \right) + N_V^E, \quad (4.13)$$

where N_a is the number of attributes

The number of calculations needed for subsequent levels of the decision tree is given by

$$N(l) = \prod_{A_i=1}^{l-1} N_v[A_i] \cdot \left(\sum_{A_i=1}^{N_a} \left(N_A^{IG}(A_i) + N_A^E(A_i) \right) + N_V^E \right), \quad (4.14)$$

where l is a level in the decision tree, and $1 < l \leq N_a - 1$

Consequently, the worst case scenario where the maximum number of calculations that need to be conducted to build the decision tree can be found as follows:

$$N = \sum_{A_i=1}^{N_a} \left(N_A^{IG}(A_i) + N_A^E(A_i) \right) + N_V^E + \sum_{l=2}^{N_a-1} \left(\prod_{A_i=1}^{l-1} N_v[A_i] \cdot \left(\sum_{A_i=1}^{N_a} \left(N_A^{IG}(A_i) + N_A^E(A_i) \right) + N_V^E \right) \right) \quad (4.15)$$

for $N_a > 2$

Equation 4.15 implies that the worst case scenario for the decision tree is of the order $O(N_a(N_v[N_a])^{N_a})$.

4.6 CALCULATING THE MEMORY USAGE OF THE MDFT METHOD

In embedded systems memory resources are limited. Consequently, the memory usage of an algorithm needs to be considered before deployment to ensure sufficient memory is available on the target platform identified.

One of the characteristics of MDFT method is its suitability to embedded applications, where the memory required by the algorithm can be calculated and be known *a priori*. The memory used to build the decision tree is determined below.

- The number of elements of the multi-dimensional frequency table is given by

$$N_e = N_c \cdot \prod_{A_i=1}^{N_a} N_v[A_i] \quad , \quad (4.16)$$

and the total memory requirement of MDFT in bytes can be calculated by

$$S_{MDFT} = N_e \cdot Z \quad (4.17)$$

where Z is the memory required for each element of the MDFT.

In the MDFT method, the adopted node structure of the decision tree shown in Figure 4.2 ensures a fixed memory usage. Figure 4.3 (a) shows an example of a DT structure, but this is implemented in MDFT as a node structure linked list as shown in Figure 4.3 (b). This fixed memory usage is achieved by defining two pointers to link the network of nodes of a decision tree. The first pointer links to the child node while the second pointer links to the sibling node. As the pointers are fixed in their memory usage (for a given addressing architecture), the maximum memory usage can be known *a priori*, which makes the approach more suitable for embedded solutions.

In contrast, the number of pointers created at node of other DT approaches, such as in C4.5 and ITI, varies according to the attribute selection at a node, which depends on a number of the children of that node. Consequently, these methods have a memory usage not known in advance, making them less well suited to embedded applications.

```

/* MDFT node structure */
/* u8 and u16 notate unsigned 8 bit and unsigned 16-bit integer numbers respectively */
typedef struct node{
    u8 Att;                /* attribute chosen at node */
    u16 ClassInstance[Nc]; /* number of class vectors at node */
    u16 Child;            /* pointer to child node */
    u16 Sibling;         /* pointer to sibling node */
    Bool AttUsed[Na];    /* one flag for each attribute used in the tree above this node */
    u8 AttValUsed[Na]; /* corresponding attribute value for each attribute flagged in
                        AttUsed */
} NODE;

```

Figure 4.2: MDFT node structure

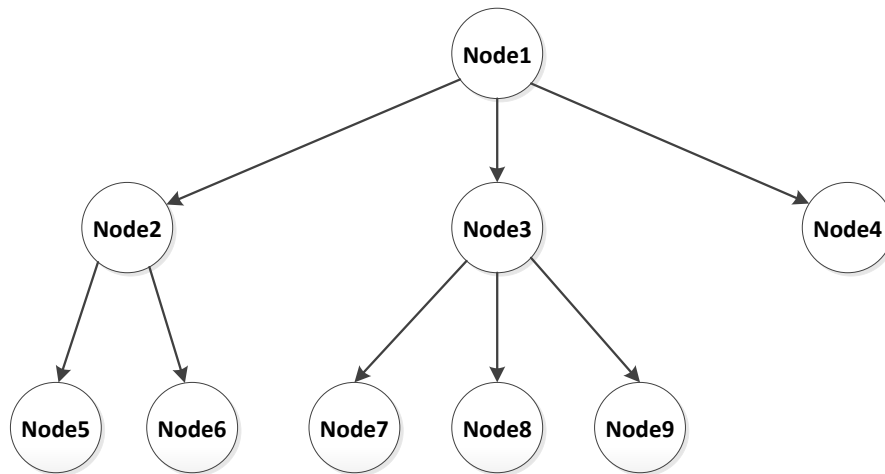


Figure 4.3 (a): Example of a conventionally linked decision tree structure

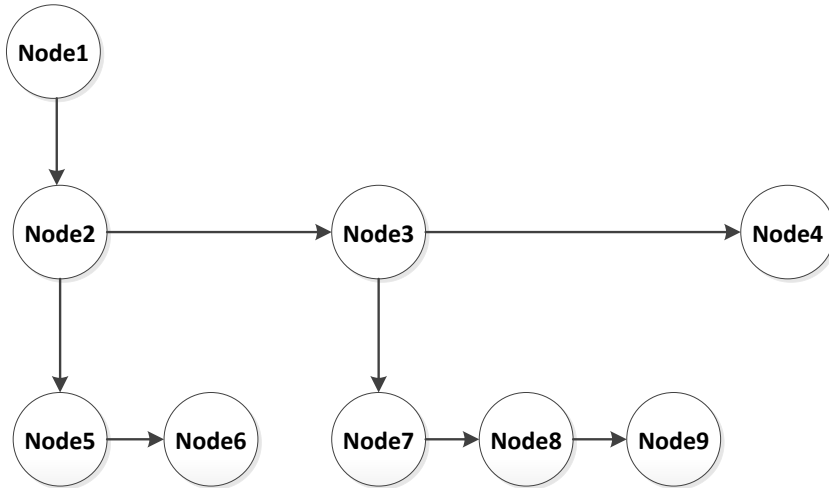


Figure 4.3 (b): Decision tree structure in which the nodes have fixed memory usage by employing child and sibling pointer

From the node structure shown in Figure 4.2, the number of bytes used by each node can be calculated as follows:

$$S_N = 1 + 2 + 2 + 2 + 1 + 1 = 9 \text{ bytes} \quad (4.18)$$

- By referring to Equations 4.17 and 4.18, the total memory usage by an MDFT decision tree in bytes can be calculated as follows :

$$S_T = S_{MDFT} + S_N \cdot N_{Nodes} \quad (4.19)$$

where N_{Nodes} is the number of nodes needed for generating the DT.

4.7 GENERATING A DT AND RULES USING THE MDFT METHOD

Figure 4.4 describes the algorithm of the MDFT method in generating a decision tree. On arrival of a new training vector the MDFT is updated, the main process of generating a decision tree follows the routine shown below and on its completion a set of generated rules will be supplied. The generated rules are created by following the paths from the root node of the decision tree to each of the leaf nodes. The set of rules are then stored to be used in the classification process of new vectors.

Input: Multi-Dimensional Frequency Table (MDFT); set of attributes; training vector and unclassified vector	
Outputs: A Decision Tree, generated rules and classified vector	
Start	
1.	Update the Multi-Dimensional frequency table following the arrival of each new training vector
2.	IF all the stored entries in the MDFT are of the same class THEN produce a DT with a single node (Leaf Node) labeled with that class, go to step 7 ELSE create a node in the DT
3.	Select an attribute among the set of attributes with the highest IG value and label the node with selected attribute
4.	Add a branch for each known value of the selected attribute for that node.
5.	Create a node for each branch
6.	IF all the MDFT entries for this node are of the same class THEN label the node as a leaf node of that class ELSE compute the entropy value and the IG for all attributes, select the attribute of the largest IG and use this attribute for the current node, go to step 4
7.	Generate rules from the DT
8.	Store the DT and the rules and use as required in the classification of new vectors
9.	IF more training is required THEN go to step 1
End	

Figure 4.4: Description of the MDFT algorithm

4.8 AN ILLUSTRATIVE EXAMPLE OF THE MDFT METHOD

In this Section, an example will be used to illustrate how data can be stored in the MDFT and then used to obtain a DT To illustrate the generation of the MDFT algorithm, the weather problem dataset is shown in table 4.1, which has four attributes, two possible outputs, uses various characteristics of the weather to determine whether to play a game of tennis [11].

Table 4.1: The weather problem dataset contains 16 input vectors with 15 unique entries.

Number	ATTRIBUTES				CLASS
	Outlook	Temperature	Humidity	Wind	
1	Sunny	Hot	High	Weak	Don't play
2	Sunny	Hot	High	Strong	Don't play
3	Overcast	Hot	High	Weak	Play
4	Rainy	Mild	High	Weak	Play
5	Sunny	Mild	High	Weak	Play
6	Rainy	Cool	Normal	Weak	Play
7	Rainy	Cool	Normal	Strong	Don't play
8	Overcast	Mild	High	Strong	Play
9	Sunny	Mild	High	Weak	Don't play
10	Sunny	Cool	Normal	Weak	Play
11	Rainy	Mild	Normal	Weak	Play
12	Sunny	Mild	Normal	Strong	Play
13	Overcast	Hot	Normal	Weak	Play
14	Sunny	Mild	High	Weak	Play
15	Overcast	Cool	Normal	Strong	Play
16	Rainy	Mild	High	Strong	Don't play

Table 4.2 shows a summary of the attributes and classes of the weather problem where the two classes {don't play, play} notated by {0, 1} correspondingly, and the attributes {Outlook, temperature, humidity, wind} are represented by $\{A_1, A_2, A_3, A_4\}$ with numerical notation for the values.

Table 4.2: The attributes and classes shown in the weather example with the corresponding notation

CLASSES			
Don't play ($C = 0$)		Play ($C = 1$)	
ATTRIBUTES			
Outlook (A_1)	Temperature (A_2)	Humidity (A_3)	Wind (A_4)
Sunny (0)	Hot (0)	High (0)	Weak (0)
Overcast (1)	Mild (1)	Normal (1)	Strong (1)
Rainy (2)	Cool (2)		

Each element of the MDFT in Table 4.3 represents an input vector which has a unique notation according to the attributes and class value. A function f is defined to provide the number of vectors of input vectors as recorded in each of the MDFT elements, where each element value can be accessed using the attribute and class value as shown in Equation 4.20.

$$\text{number of vectors} = f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) \quad (4.20)$$

where: $v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}$ represent the attribute values for attributes A_1, A_2, A_3 and A_4 . And C is the class values.

To clarify how the calculations are performed in this example, the notation (Q_C^k) signifies the sum of values stored in the MDFT for node k and for class C (where in the current example $C=0$ for class 0, $C=1$ for class 1 and $C=T$ for both classes taken together) while the entropy for node k is shown as E^k .

Table 4.4 illustrates how the input vectors of the weather problem populate the elements in the MDFT.

Table 4.3: MDFT shows a unique notation for each cell according to its attributes and class values.

		Outlook (A_1)										
		Sunny (0)	Overcast (1)	Rainy (2)	Sunny (0)	Overcast (1)	Rainy (2)	Sunny (0)	Overcast (1)	Rainy (2)		
Play ($C=1$)	Humidity (A_3)	High (0)	0,0,0,0,1	1,0,0,0,1	2,0,0,0,1	0,1,0,0,1	1,1,0,0,1	2,1,0,0,1	0,2,0,0,1	1,2,0,0,1	2,2,0,0,1	Weak (0)
		Normal (1)	0,0,1,0,1	1,0,1,0,1	2,0,1,0,1	0,1,1,0,1	1,1,1,0,1	2,1,1,0,1	0,2,1,0,1	1,2,1,0,1	2,2,1,0,1	Weak (0)
High (0)		0,0,0,1,1	1,0,0,1,1	2,0,0,1,1	0,1,0,1,1	1,1,0,1,1	2,1,0,1,1	0,2,0,1,1	1,2,0,1,1	2,2,0,1,1	Strong (1)	
Normal (1)		0,0,1,1,1	1,0,1,1,1	2,0,1,1,1	0,1,1,1,1	1,1,1,1,1	2,1,1,1,1	0,2,1,1,1	1,2,1,1,1	2,2,1,1,1	Strong (1)	
Don't Play ($C=0$)	Humidity (A_3)	High (0)	0,0,0,0,0	1,0,0,0,0	2,0,0,0,0	0,1,0,0,0	1,1,0,0,0	2,1,0,0,0	0,2,0,0,0	1,2,0,0,0	2,2,0,0,0	Weak (0)
		Normal (1)	0,0,1,0,0	1,0,1,0,0	2,0,1,0,0	0,1,1,0,0	1,1,1,0,0	2,1,1,0,0	0,2,1,0,0	1,2,1,0,0	2,2,1,0,0	Weak (0)
		High (0)	0,0,0,1,0	1,0,0,1,0	2,0,0,1,0	0,1,0,1,0	1,1,0,1,0	2,1,0,1,0	0,2,0,1,0	1,2,0,1,0	2,2,0,1,0	Strong (1)
		Normal (1)	0,0,1,1,0	1,0,1,1,0	2,0,1,1,0	0,1,1,1,0	1,1,1,1,0	2,1,1,1,0	0,2,1,1,0	1,2,1,1,0	2,2,1,1,0	Strong (1)
		Temperature (A_2)										
		Hot (0)	Hot (0)	Hot (0)	Mild (1)	Mild (1)	Mild (1)	Cool (2)	Cool (2)	Cool (2)		

Table 4.4: MDFT for the weather problem, where the cells are updated according to the input dataset.

		Outlook (A_1)										
		Sunny (0)	Overcast (1)	Rainy (2)	Sunny (0)	Overcast (1)	Rainy (2)	Sunny (0)	Overcast (1)	Rainy (2)		
Play ($C=1$)	Humidity (A_3)	High (0)	0	1	0	2	0	1	0	0	0	Weak (0)
		Normal (1)	0	1	0	0	0	1	1	0	1	Weak (0)
		High (0)	0	0	0	0	1	0	0	0	0	Strong (1)
		Normal (1)	0	0	0	1	0	0	0	1	0	Strong (1)
Don't Play ($C=0$)	Humidity (A_3)	High (0)	1	0	0	1	0	0	0	0	0	Weak (0)
		Normal (1)	0	0	0	0	0	0	0	0	0	Weak (0)
		High (0)	1	0	0	0	0	1	0	0	0	Strong (1)
		Normal (1)	0	0	0	0	0	0	0	0	1	Strong (1)
		Temperature (A_2)										
		Hot (0)	Hot (0)	Hot (0)	Mild (1)	Mild (1)	Mild (1)	Cool (2)	Cool (2)	Cool (2)		

The weather problem given in Table 4.1 consists of four attributes ($N_a = 4$) and two classes ($N_c = 2$). In the multi-dimensional array, each attribute is represented by a single dimension of the array with an extra dimension required for the class.

After the MDFT has been populated, the decision tree can be generated. The splitting criterion used in generating the decision tree is information gain (see Section 2.2.3).

The following shows the total entropy calculation needed for the root node and is calculated using Equation 2.2 in Section 2.2.3 as follows.

The number of vectors of class 0 in the input vectors supplied is given by the sum of all values in the bottom half (for class 0) of Table 4.4

$$Q_{C_0}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) \quad (4.21)$$

$$= f(0,0,0,0,0) + f(1,0,0,0,0) + \dots + f(2,2,1,1,0) = 5$$

The number of vectors of class 1 can be found by the sum of all values in the upper half (for class 1) of Table 4.4

$$Q_{C_1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) \quad (4.22)$$

$$= f(0,0,0,0,1) + f(1,0,0,0,1) + \dots + f(2,2,1,1,1) = 11$$

The number of vectors of both classes is given by the sum of all the values in the (for both classes) in Table 4.4

$$Q_{C_T}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) \quad (4.23)$$

$$= Q_{C_0}^1 + Q_{C_1}^1 = 16$$

The total entropy at the root node is thus

$$\begin{aligned}
 E_T^1 &= \left(-\frac{Q_{C_0}^1}{Q_{C_T}^1} \log_2 \frac{Q_{C_0}^1}{Q_{C_T}^1} \right) - \left(\frac{Q_{C_1}^1}{Q_{C_T}^1} \log_2 \frac{Q_{C_1}^1}{Q_{C_T}^1} \right) \\
 &= \left(-\frac{5}{16} \log_2 \frac{5}{16} \right) - \left(\frac{11}{16} \log_2 \frac{11}{16} \right) = 0.896038
 \end{aligned} \tag{4.24}$$

Choosing the attribute at the root node

In this Section, the entropy values will be computed for each of the attributes for the root node.

1. Entropy for attribute 'outlook'.

For attribute value 'sunny'

$$Q_{C_0}^1 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3 \tag{4.25}$$

$$Q_{C_1}^1 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 4 \tag{4.26}$$

$$Q_{C_T}^1 = Q_{C_0}^1 + Q_{C_1}^1 = 7 \tag{4.27}$$

For attribute value 'overcast'

$$Q_{C_0}^1 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0 \tag{4.28}$$

$$Q_{C_1}^1 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 4 \tag{4.29}$$

$$Q_{C_T}^1 = Q_{C_0}^1 + Q_{C_1}^1 = 4 \tag{4.30}$$

For attribute value 'rainy'

$$Q_{C_0^1}^1 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2 \quad (4.31)$$

$$Q_{C_1^1}^1 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3 \quad (4.32)$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 5 \quad (4.33)$$

By using Equation 2.4, the entropy for attribute 'outlook' can be found from

$$\begin{aligned} E^1(A_1) &= \frac{Q_{C_T^1}^1}{Q_{C_T^1}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T^1}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T^1}^1} \left[\left(-\frac{Q_{C_0^2}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^2}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^2}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^2}^1}{Q_{C_T^1}^1} \right) \right] = 0.734459 \end{aligned} \quad (4.34)$$

By using Equation 2.5, the information gain for attribute 'outlook' is given by

$$\begin{aligned} IG^1(A_1) &= E_T^1 - E^1(A_1) \\ &= 0.896038 - 0.734459 = 0.161579 \\ &\quad \text{(This is the largest gain at this node)} \end{aligned} \quad (4.35)$$

2. Entropy for attribute 'temperature'

For attribute value 'hot'

$$Q_{C_0^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_1^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 4$$

For attribute value 'mild'

$$Q_{C_0^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_1^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 6$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 8$$

For attribute value 'cool'

$$Q_{C_0^2}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_1^2}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3$$

$$Q_{C_T^2}^1 = Q_{C_0^2}^1 + Q_{C_1^2}^1 = 4$$

By using Equation 2.4, the entropy for attribute 'temperature' can be found from

$$\begin{aligned} E^1(A_2) &= \frac{Q_{C_T^0}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^0}^1}{Q_{C_T^0}^1} \log_2 \frac{Q_{C_0^0}^1}{Q_{C_T^0}^1} \right) - \left(\frac{Q_{C_1^0}^1}{Q_{C_T^0}^1} \log_2 \frac{Q_{C_1^0}^1}{Q_{C_T^0}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^2}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^2}^1}{Q_{C_T^2}^1} \log_2 \frac{Q_{C_0^2}^1}{Q_{C_T^2}^1} \right) - \left(\frac{Q_{C_1^2}^1}{Q_{C_T^2}^1} \log_2 \frac{Q_{C_1^2}^1}{Q_{C_T^2}^1} \right) \right] = 0.858459 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'temperature' is

$$\begin{aligned} IG^1(A_2) &= E_T^1 - E^1(A_2) \\ &= 0.896038 - 0.858459 = 0.03758 \end{aligned}$$

(This is not the largest gain at this node)

3. Entropy for attribute 'humidity'

For attribute value 'high'

$$Q_{C_0^0}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 4$$

$$Q_{C_1^0}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 5$$

$$Q_{C_T^0}^1 = Q_{C_0^0}^1 + Q_{C_1^0}^1 = 9$$

For attribute value 'normal'

$$Q_{C_0^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_1^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 6$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 7$$

By using Equation 2.4, the entropy for attribute 'humidity' can be found

$$\begin{aligned} E^1(A_3) &= \frac{Q_{C_T^0}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^0}^1}{Q_{C_T^0}^1} \log_2 \frac{Q_{C_0^0}^1}{Q_{C_T^0}^1} \right) - \left(\frac{Q_{C_1^0}^1}{Q_{C_T^0}^1} \log_2 \frac{Q_{C_1^0}^1}{Q_{C_T^0}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] = 0.816337 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'humidity' is

$$\begin{aligned} IG^1(A_3) &= E_T^1 - E^1(A_3) \\ &= 0.896038 - 0.816337 = 0.079701 \end{aligned}$$

(this is not the largest gain at this node)

4. Entropy for attribute 'wind'

For attribute value 'weak'

$$Q_{C_0^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_1^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 8$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 10$$

For attribute value 'strong'

$$Q_{C_0^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3$$

$$Q_{C_1^1}^1 = \sum_{v_{A_1}=0}^2 \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 6$$

By using Equation 2.4, the entropy for attribute 'wind' can be found from

$$\begin{aligned} E^1(A_4) &= \frac{Q_{C_T^0}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^0}^1}{Q_{C_T^0}^1} \log_2 \frac{Q_{C_0^0}^1}{Q_{C_T^0}^1} \right) - \left(\frac{Q_{C_0^0}^1}{Q_{C_T^0}^1} \log_2 \frac{Q_{C_0^0}^1}{Q_{C_T^0}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] = 0.826205 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'wind' is given by

$$\begin{aligned} IG^1(A_4) &= E_T^1 - E^1(A_4) \\ &= 0.896038 - 0.826205 = 0.069833 \\ &\text{(this is not the largest gain at this node)} \end{aligned}$$

The DT progress beyond this point is shown in Figure 4.5. For the root node, the attribute 'outlook' is chosen as it has the largest information gain value. There are three attribute values for 'outlook', namely sunny, overcast and rainy. Therefore the root node has three children as follows.

Sunny --> node 2

Overcast --> node 3

Rainy --> node 4

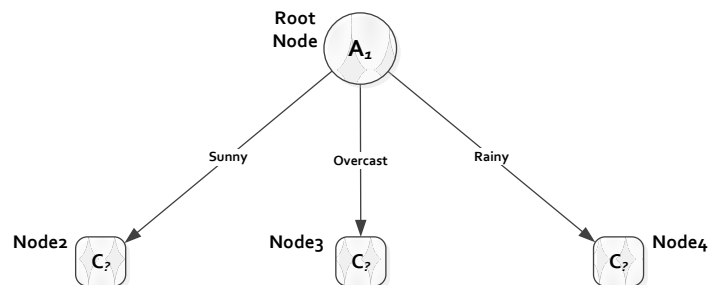


Figure 4.5: Attribute Outlook (A_1) is chosen for the root node.

Calculations of node 2

After choosing 'outlook' for node 1 according to the entropy and information gain calculations given by Equations 4.34 and 4.35, the following calculations are for node 2 to choose the next attribute from the remaining attributes following similar procedure.

Three main relations will be considered in this stage.

1. Outlook-->Temperature
2. Outlook-->'humidity'
3. Outlook-->Wind

For node 2 where value 'sunny' presented, the number of vectors is 7 as given by Equation 4.27. In the following, the calculations for node 2 to choose the next attribute.

Total entropy for node 2

For first child node 2 for root node with attribute value 'sunny', the total node entropy can be calculated using the following parameters by referring to Equations 4.25, 4.26 and 4.27.

$$Q_{C_0}^2 = Q_{C_0^0}^1 = 3$$

$$Q_{C_1}^2 = Q_{C_1^0}^1 = 4$$

$$Q_{C_T}^2 = Q_{C_T^0}^1 = 7$$

From Equation 2.2, the entropy, for all vectors within attribute value 'sunny' can be found from

$$\begin{aligned} E_T^2 &= \left(-\frac{Q_{C_0}^2}{Q_{C_T}^2} \log_2 \frac{Q_{C_0}^2}{Q_{C_T}^2} \right) - \left(\frac{Q_{C_1}^2}{Q_{C_T}^2} \log_2 \frac{Q_{C_1}^2}{Q_{C_T}^2} \right) = \left(-\frac{3}{7} \log_2 \frac{3}{7} \right) - \left(\frac{4}{7} \log_2 \frac{4}{7} \right) \\ &= 0.985228 \end{aligned} \quad (4.36)$$

Attribute entropy for node 2

1. The entropy for attribute 'temperature' when 'outlook' is 'sunny' ($v_{A_1} = 0$) can be found as follows.

For attribute value 'hot'

$$Q_{C_0^0}^2 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2 \quad (4.37)$$

$$Q_{C_1^0}^2 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0 \quad (4.38)$$

$$Q_{C_T^0}^2 = Q_{C_0^0}^2 + Q_{C_1^0}^2 = 2 \quad (4.39)$$

For attribute value 'mild'

$$Q_{C_0^2}^2 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1 \quad (4.40)$$

$$Q_{C_1^2}^2 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3 \quad (4.41)$$

$$Q_{C_T^2}^2 = Q_{C_0^2}^2 + Q_{C_1^2}^2 = 4 \quad (4.42)$$

For attribute value 'cool'

$$Q_{C_0^2}^2 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0 \quad (4.43)$$

$$Q_{C_1^2}^2 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1 \quad (4.44)$$

$$Q_{C_T^2}^2 = Q_{C_0^2}^2 + Q_{C_1^2}^2 = 1 \quad (4.45)$$

By using Equation 2.4,, the entropy for attribute 'temperature' can be found from

$$\begin{aligned} E^2(A_2) &= \frac{Q_{C_T^0}^2}{Q_{C_T}^2} \left[\left(-\frac{Q_{C_0^0}^2}{Q_{C_T^0}^2} \log_2 \frac{Q_{C_0^0}^2}{Q_{C_T^0}^2} \right) - \left(\frac{Q_{C_1^0}^2}{Q_{C_T^0}^2} \log_2 \frac{Q_{C_1^0}^2}{Q_{C_T^0}^2} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^2}{Q_{C_T}^2} \left[\left(-\frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \right) - \left(\frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \right) \right] \\ &\quad + \frac{Q_{C_T^2}^2}{Q_{C_T}^2} \left[\left(-\frac{Q_{C_0^2}^2}{Q_{C_T^2}^2} \log_2 \frac{Q_{C_0^2}^2}{Q_{C_T^2}^2} \right) - \left(\frac{Q_{C_1^2}^2}{Q_{C_T^2}^2} \log_2 \frac{Q_{C_1^2}^2}{Q_{C_T^2}^2} \right) \right] = 0.463587 \end{aligned} \quad (4.46)$$

By using Equation 2.5, the information gain for attribute 'temperature' is given by

$$\begin{aligned} IG^2(A_2) &= E_T^2 - E^2(A_2) \\ &= 0.985228 - 0.463587 = 0.521641 \text{ (is the MaxGain)} \end{aligned} \quad (4.47)$$

2. Entropy for attribute 'humidity' when 'outlook' is 'sunny' ($v_{A_1} = 0$)

For attribute value 'high'

$$Q_{C_0^0}^2 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3$$

$$Q_{C_1^0}^2 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_T^0}^2 = Q_{C_0^0}^2 + Q_{C_1^0}^2 = 5$$

For attribute value 'normal'

$$Q_{C_0^1}^2 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0$$

$$Q_{C_1^1}^2 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_T^1}^2 = Q_{C_0^1}^2 + Q_{C_1^1}^2 = 2$$

By using Equation 2.4, the entropy for attribute 'humidity' can be found from

$$\begin{aligned} E^2(A_3) &= \frac{Q_{C_T^0}^2}{Q_{C_T}^2} \left[\left(-\frac{Q_{C_0^0}^2}{Q_{C_T^0}^2} \log_2 \frac{Q_{C_0^0}^2}{Q_{C_T^0}^2} \right) - \left(\frac{Q_{C_1^0}^2}{Q_{C_T^0}^2} \log_2 \frac{Q_{C_1^0}^2}{Q_{C_T^0}^2} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^2}{Q_{C_T}^2} \left[\left(-\frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \right) - \left(\frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \right) \right] = 0.693536 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'humidity' is given by

$$\begin{aligned} IG^2(A_3) &= E_T^2 - E^2(A_3) \\ &= 0.985228 - 0.693536 = 0.291692 \text{ (is not the MaxGain)} \end{aligned}$$

3. Entropy for attribute 'wind' when 'outlook' is 'sunny' ($v_{A_1} = 0$)

For attribute value 'weak'

$$Q_{C_0^2}^2 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_1^2}^2 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3$$

$$Q_{C_T^2}^2 = Q_{C_0^2}^2 + Q_{C_1^2}^2 = 5$$

For attribute value 'strong'

$$Q_{C_0^2}^2 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_1^2}^2 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_T^2}^2 = Q_{C_0^2}^2 + Q_{C_1^2}^2 = 2$$

By using Equation 2.4, the entropy for attribute 'wind' can be found from

$$\begin{aligned} E^2(A_4) &= \frac{Q_{C_T^2}^2}{Q^2} \left[\left(-\frac{Q_{C_0^2}^2}{Q_{C_T^2}^2} \log_2 \frac{Q_{C_0^2}^2}{Q_{C_T^2}^2} \right) - \left(\frac{Q_{C_1^2}^2}{Q_{C_T^2}^2} \log_2 \frac{Q_{C_1^2}^2}{Q_{C_T^2}^2} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^2}{Q^2} \left[\left(-\frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \right) - \left(\frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \right) \right] = 0.97925 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'wind' is given by

$$\begin{aligned} IG^2(A_4) &= E_T^2 - E^2(A_4) \\ &= 0.985228 - 0.97925 = 0.005978 \text{ (is not the MaxGain)} \end{aligned}$$

The DT progress is shown in Figure 4.6, where attribute 'temperature' is chosen as it has the largest information gain value as given by Equation 4.47.

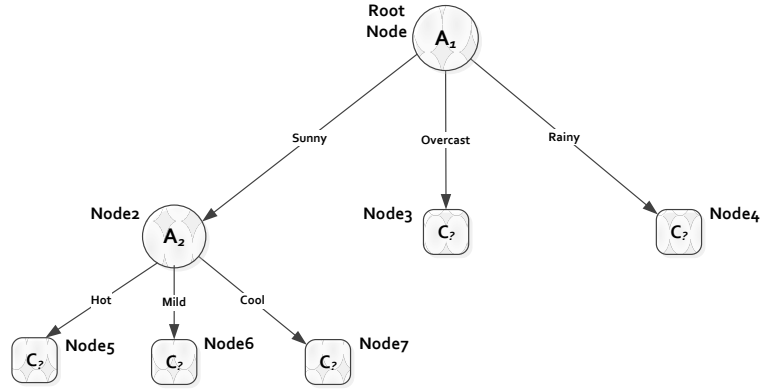


Figure 4.6: Attribute Temperature (A_2) is selected for node 2.

Calculations of node 3

The next node is node 3 where attribute value 'overcast' presented, the number of vectors is 4 as given by Equation 4.30. The calculations for the entropy and the information gain will be repeated on this node as follows.

Total entropy for node 3

For second child node 3 for root node with value 'overcast', the total node entropy can be calculated using the following parameters by referring to Equations 4.28, 4.29 and 4.30.

$$Q_{C_0}^3 = Q_{C_0^1}^1 = 0$$

$$Q_{C_1}^3 = Q_{C_1^1}^1 = 4$$

$$Q_{C_T}^3 = Q_{C_T^1}^1 = 4$$

From Equation 2.2, the entropy for all vectors within value 'overcast' is given by

$$E_T^3 = \left(-\frac{Q_{C_0}^3}{Q_{C_T}^3} \log_2 \frac{Q_{C_0}^3}{Q_{C_T}^3} \right) - \left(\frac{Q_{C_1}^3}{Q_{C_T}^3} \log_2 \frac{Q_{C_1}^3}{Q_{C_T}^3} \right) = \left(-\frac{0}{4} \log_2 \frac{0}{4} \right) - \left(\frac{4}{4} \log_2 \frac{4}{4} \right) = 0 \quad (4.48)$$

According to the entropy calculation node 3 is a leaf node as all the vectors belongs only to one class. The DT progress beyond this point is shown in Figure 4.7.

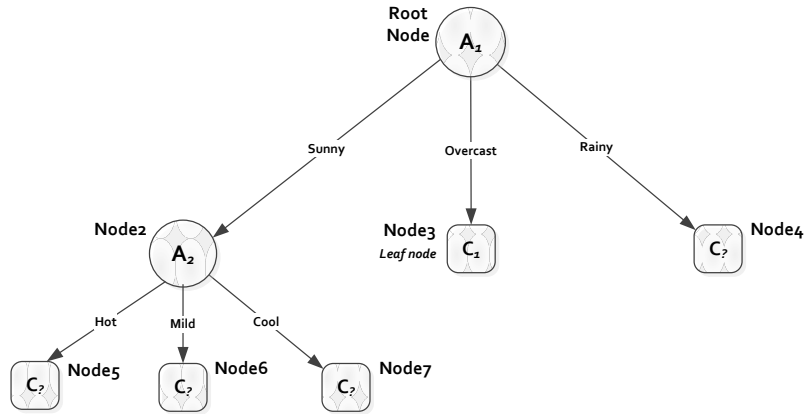


Figure 4.7: Node 3 is a leaf node.

Calculations of node 4

For node 4 where value 'rainy' presented, the number of vectors is 5 as given by Equation 4.33. In the following, the calculations for node 4 to choose the next attribute.

Total entropy for node 4

For third child node 4 for root node with attribute value 'rainy', the total node entropy can be calculated using the following parameters by referring to Equations 4.31, 4.32 and 4.33.

$$Q_{C_0}^4 = Q_{C_0^2}^1 = 2$$

$$Q_{C_1}^4 = Q_{C_1^2}^1 = 3$$

$$Q_{C_T}^4 = Q_{C_T^2}^1 = 5$$

From Equation 2.2, the entropy for all vectors within value 'rainy' can be found from

$$\begin{aligned} E_T^4 &= \left(-\frac{Q_{C_0}^4}{Q_{C_T}^4} \log_2 \frac{Q_{C_0}^4}{Q_{C_T}^4} \right) - \left(\frac{Q_{C_1}^4}{Q_{C_T}^4} \log_2 \frac{Q_{C_1}^4}{Q_{C_T}^4} \right) = \left(-\frac{2}{5} \log_2 \frac{2}{5} \right) - \left(\frac{3}{5} \log_2 \frac{3}{5} \right) \\ &= 0.970951 \end{aligned}$$

Attribute entropy for node 4

1. Entropy for attribute 'temperature' when 'outlook' is 'rainy' ($v_{A_1} = 2$)

For attribute value 'hot'

$$Q_{C_0}^4 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0$$

$$Q_{C_1}^4 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0$$

$$Q_{C_T}^4 = Q_{C_0}^4 + Q_{C_1}^4 = 0$$

For attribute value 'mild'

$$Q_{C_0}^4 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_1}^4 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_T}^4 = Q_{C_0}^4 + Q_{C_1}^4 = 3$$

For attribute value 'cool'

$$Q_{C_0^4}^4 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_1^4}^4 = \sum_{v_{A_3}=0}^1 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_2^4}^4 = Q_{C_0^4}^4 + Q_{C_1^4}^4 = 2$$

By using Equation 2.4, the entropy for attribute 'temperature' can be found from

$$\begin{aligned} E^4(A_2) &= \frac{Q_{C_T^4}^4}{Q^4} \left[\left(-\frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \right) - \left(\frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \right) \right] \\ &\quad + \frac{Q_{C_T^4}^4}{Q^4} \left[\left(-\frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \right) - \left(\frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \right) \right] \\ &\quad + \frac{Q_{C_T^4}^4}{Q^4} \left[\left(-\frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \right) - \left(\frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \right) \right] = 0.950978 \end{aligned} \quad (4.49)$$

By using Equation 2.5, the information gain for attribute 'temperature' is given by

$$\begin{aligned} IG^4(A_2) &= E_T^4 - E^4(A_2) \\ &= 0.970951 - 0.950978 = 0.019973 \\ &\quad (\text{this is the largest gain at this node}) \end{aligned} \quad (4.50)$$

2. Entropy for 'humidity' when 'outlook' is 'rainy' ($v_{A_1} = 2$)

For attribute value 'high'

$$Q_{C_0^4}^4 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_1^4}^4 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_T^4}^4 = Q_{C_0^4}^4 + Q_{C_1^4}^4 = 2$$

For attribute value 'normal'

$$Q_{C_0^4}^4 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_1^4}^4 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_T^4}^4 = Q_{C_0^4}^4 + Q_{C_1^4}^4 = 3$$

By using Equation 2.4, the entropy for attribute 'humidity' can be found from

$$\begin{aligned} E^4(A_3) &= \frac{Q_{C_T^4}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \right) - \left(\frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \right) \right] \\ &\quad + \frac{Q_{C_T^4}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \right) - \left(\frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \right) \right] = 0.950978 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'humidity' is

$$\begin{aligned} IG^4(A_3) &= E_T^4 - E^4(A_3) \\ &= 0.970951 - 0.950978 = 0.019973 \\ &\quad (\text{this is not the largest gain at this node}) \end{aligned}$$

3. Entropy for attribute 'wind' when 'outlook' is 'rainy' ($v_{A_1} = 2$)

For attribute value 'weak'

$$Q_{C_0^4}^4 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0 \quad (4.51)$$

$$Q_{C_1^4}^4 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 3 \quad (4.52)$$

$$Q_{C_T^4}^4 = Q_{C_0^4}^4 + Q_{C_1^4}^4 = 3 \quad (4.53)$$

For attribute value 'strong'

$$Q_{C_0^4}^4 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2 \quad (4.54)$$

$$Q_{C_1^4}^4 = \sum_{v_{A_2}=0}^2 \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0 \quad (4.55)$$

$$Q_{C_T^4}^4 = Q_{C_0^4}^4 + Q_{C_1^4}^4 = 2 \quad (4.56)$$

By using Equation 2.4, the entropy for attribute 'wind' can be found from

$$E^4(A_4) = \frac{Q_{C_T^4}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \right) - \left(\frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \right) \right] \\ + \frac{Q_{C_T^4}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_0^4}^4}{Q_{C_T^4}^4} \right) - \left(\frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \log_2 \frac{Q_{C_1^4}^4}{Q_{C_T^4}^4} \right) \right] = 0$$

By using Equation 2.5, the information gain for attribute 'wind' is given by

$$IG^4(A_4) = E_T^4 - E^4(A_4) \\ = 0.970951 - 0 = 0.970951 \\ \text{(this is the largest gain at this node)}$$

The DT progress is shown in Figure 4.8, shows that attribute 'wind' is chosen as it has the largest information gain value. There are three values for 'temperature', namely hot, mild and cool. Therefore node 2 has three children as follows.

Hot --> node 5

Mild --> node 6

Cool --> node 7

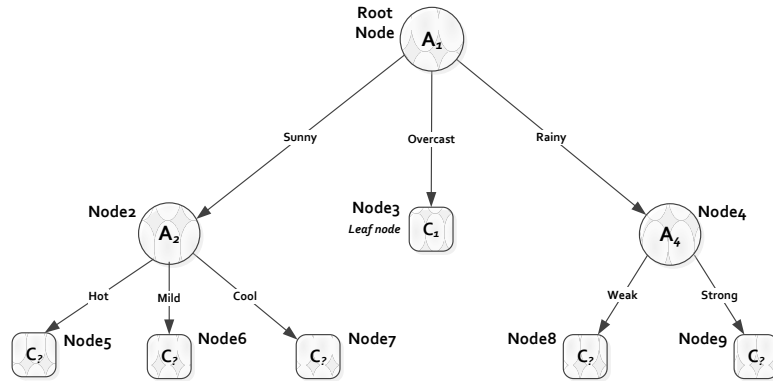


Figure 4.8: Attribute Wind (A_4) is selected for node 4.

Calculations of node 5

The next node 5 where its parent node 2 chooses attribute 'temperature' as given by Equation 4.47. The following calculations are for node 5 to choose the next attribute from the remaining two attributes.

Two main relations to be considered at this stage are

1. Outlook-->Temperature-->'humidity'
2. Outlook--> Temperature-->Wind

For node 5 where value 'hot' is presented, the number of vectors is 2 as given by Equation 4.39. In the following, the calculations for node 5 to choose the next attribute.

Total entropy for node 5

For first child node 5 to parent node 2 with attribute value 'hot', the total node entropy can be calculated using the following parameters that are given by Equations 4.37, 4.38 and 4.39.

$$Q_{C_0}^5 = Q_{C_0^1}^1 = 2$$

$$Q_{C_1}^5 = Q_{C_1^1}^1 = 0$$

$$Q_{C_T}^5 = Q_{C_T^1}^1 = 2$$

From Equation 2.2 the total entropy can be found from

$$E_T^5 = \left(-\frac{Q_{C_0}^5}{Q_{C_T}^5} \log_2 \frac{Q_{C_0}^5}{Q_{C_T}^5} \right) - \left(\frac{Q_{C_1}^5}{Q_{C_T}^5} \log_2 \frac{Q_{C_1}^5}{Q_{C_T}^5} \right) = 0$$

According to the entropy calculation, node 5 is a leaf node as all the vectors belongs to class 0. The DT progress beyond this point has been updated as shown in Figure 4.9.

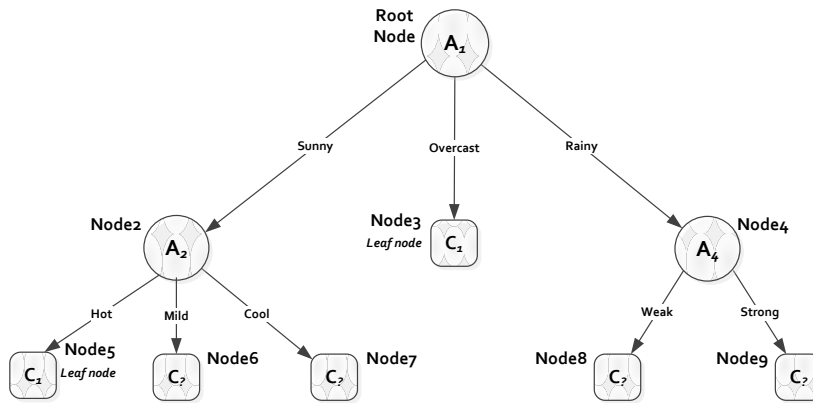


Figure 4.9: Node 5 is a leaf node.

Calculations of node 6

For node 6 where value 'mild' is presented, the number of vectors is 4 as given by Equation 4.42. In the following, the calculations of node 6 to choose the next attribute.

Total entropy for node 6

For second child node 6 to parent node 2 with value 'mild', the total entropy can be calculated using the following parameters that are given by Equations 4.40, 4.41 and 4.42.

$$Q_{C_0}^6 = Q_{C_0}^2 = 1$$

$$Q_{C_1}^6 = Q_{C_1}^2 = 3$$

$$Q_{C_T}^6 = Q_{C_T}^2 = 4$$

From Equation 2.2, the total entropy can be found from

$$E_T^6 = \left(-\frac{Q_{C_0}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_0}^6}{Q_{C_T}^6} \right) - \left(\frac{Q_{C_1}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_1}^6}{Q_{C_T}^6} \right) = 0.811278$$

Attribute entropy for node 6

1. Entropy for attribute 'humidity' when ($v_{A_1} = 0$ and $v_{A_2} = 1$)

For attribute value 'high'

$$Q_{C_0}^6 = \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1 \quad (4.57)$$

$$Q_{C_1}^6 = \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2 \quad (4.58)$$

$$Q_{C_T}^6 = Q_{C_0}^6 + Q_{C_1}^6 = 3 \quad (4.59)$$

For attribute value 'normal'

$$Q_{C_0}^6 = \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0 \quad (4.60)$$

$$Q_{C_1}^6 = \sum_{v_{A_4}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1 \quad (4.61)$$

$$Q_{C_T}^6 = Q_{C_0}^6 + Q_{C_1}^6 = 1 \quad (4.62)$$

By using Equation 2.4, the entropy for attribute 'humidity' can be found from

$$\begin{aligned} E^6(A_3) &= \frac{Q_{C_T}^6}{Q_{C_T}^6} \left[\left(-\frac{Q_{C_0}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_0}^6}{Q_{C_T}^6} \right) - \left(\frac{Q_{C_1}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_1}^6}{Q_{C_T}^6} \right) \right] \\ &\quad + \frac{Q_{C_T}^6}{Q_{C_T}^6} \left[\left(-\frac{Q_{C_0}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_0}^6}{Q_{C_T}^6} \right) - \left(\frac{Q_{C_1}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_1}^6}{Q_{C_T}^6} \right) \right] = 0.688722 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'humidity' is given by

$$\begin{aligned}
 IG^6(A_3) &= E_T^6 - E^6(A_3) & (4.63) \\
 &= 0.811278 - 0.688722 = 0.122556 \\
 &\text{(this is the largest gain at this node)}
 \end{aligned}$$

2. Entropy for attribute 'wind'

For attribute value 'weak'

$$Q_{C_0^0}^6 = \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_1^0}^6 = \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 2$$

$$Q_{C_T^0}^6 = Q_{C_0^0}^6 + Q_{C_1^0}^6 = 3$$

For attribute value 'strong'

$$Q_{C_0^1}^6 = \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 0$$

$$Q_{C_1^1}^6 = \sum_{v_{A_3}=0}^1 f(v_{A_1}, v_{A_2}, v_{A_3}, v_{A_4}, C) = 1$$

$$Q_{C_T^1}^6 = Q_{C_0^1}^6 + Q_{C_1^1}^6 = 1$$

By using Equation 2.4, the entropy for 'wind' can be found from

$$\begin{aligned}
 E^6(A_4) &= \frac{Q_{C_T^0}^6}{Q_{C_T}^6} \left[\left(-\frac{Q_{C_0^0}^6}{Q_{C_T^0}^6} \log_2 \frac{Q_{C_0^0}^6}{Q_{C_T^0}^6} \right) - \left(\frac{Q_{C_1^0}^6}{Q_{C_T^0}^6} \log_2 \frac{Q_{C_1^0}^6}{Q_{C_T^0}^6} \right) \right] \\
 &\quad + \frac{Q_{C_T^1}^6}{Q_{C_T}^6} \left[\left(-\frac{Q_{C_0^1}^6}{Q_{C_T^1}^6} \log_2 \frac{Q_{C_0^1}^6}{Q_{C_T^1}^6} \right) - \left(\frac{Q_{C_1^1}^6}{Q_{C_T^1}^6} \log_2 \frac{Q_{C_1^1}^6}{Q_{C_T^1}^6} \right) \right] = 0.688722
 \end{aligned}$$

By using Equation 2.5, the information gain for 'wind' is given by

$$IG^6(A_4) = E_T^6 - E^6(A_4) \quad (4.64)$$

$$= 0.811278 - 0.688722 = 0.122556$$

(this is not the largest gain at this node)

As both attributes 'humidity' and 'wind' have similar information gain as given by Equations 4.62 and 4.63, an arbitrary decision [73] made to choose the attribute 'humidity'. The DT has been updated as shown in Figure 4.10, where attribute 'humidity' is chosen for node 6.

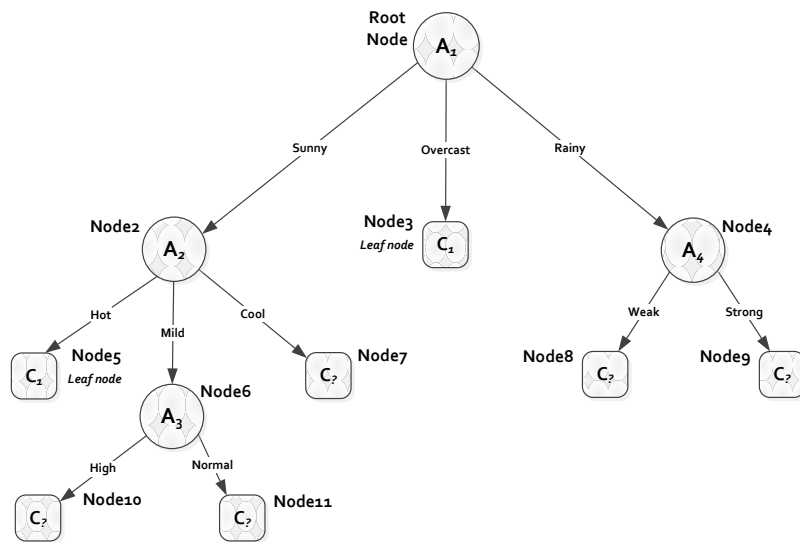


Figure 4.10: Attribute 'humidity' (A_3) is selected for node 6.

Calculations of node 7

For node 7 where the value is 'cool', the number of vectors is 1 as given by Equation 4.45. In the following, the calculations for node 7 to choose the next attribute.

Total entropy for node 7

For the third child node 7 to the parent node 2 with attribute value 'cool', the total node entropy can be calculated using the following parameters that are given by Equations 4.43, 4.44 and 4.45.

$$Q_{C_0}^7 = Q_{C_0^2}^2 = 0$$

$$Q_{C_1}^7 = Q_{C_1^2}^2 = 1$$

$$Q_{C_T}^7 = Q_{C_T^2}^2 = 1$$

From Equation 2.2, the total entropy can be found from

$$E_T^7 = \left(-\frac{Q_{C_0}^7}{Q_{C_T}^7} \log_2 \frac{Q_{C_0}^7}{Q_{C_T}^7} \right) - \left(\frac{Q_{C_1}^7}{Q_{C_T}^7} \log_2 \frac{Q_{C_1}^7}{Q_{C_T}^7} \right) = 0$$

The entropy calculation shows that node 7 is a leaf node as one vector of class 1 is left. The DT beyond this point is shown in Figure 4.11, where the DT is updated with node 7 as a leaf node.

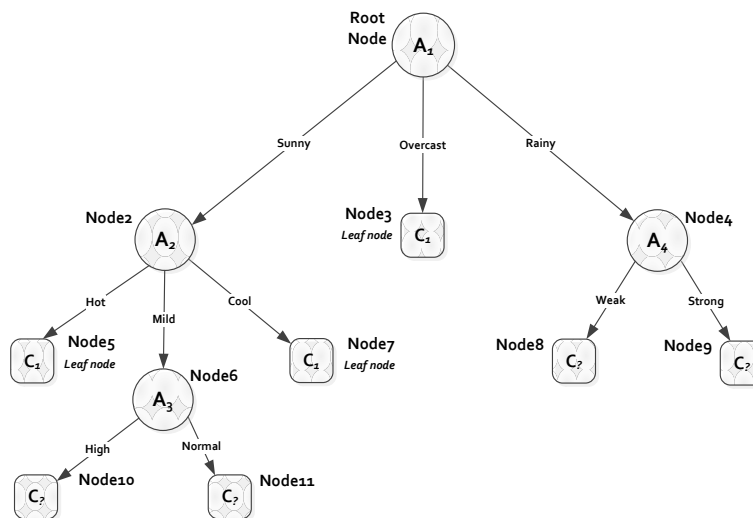


Figure 4.11: Node 7 is a leaf node.

There are two values for 'wind', namely weak and strong. Therefore there are two children for node 4 as follows.

Weak --> node 8

Strong --> node 9

Two main relations to be considered at this stage

1. Outlook-->Wind-->Humidity
2. Outlook-->Wind--> Temperature

Calculations of node 8

The next node is 8 where its parent node 4 has chosen attribute 'wind' as given by Equation 4.50. The following calculations are for node 8 to choose the next attribute from the remaining two attributes.

For node 8 where the value is 'weak', the number of vectors is 3 as given by Equation 4.53. In the following, the calculations to choose the next attribute..

Total entropy for node 8

For first child node 8 to parent node 4 with attribute value 'weak', the total node entropy can be calculated using the following parameters that are given by Equations 4.51, 4.52 and 4.53.

$$Q_{C_0}^8 = Q_{C_0^4}^8 = 0$$

$$Q_{C_1}^8 = Q_{C_1^4}^8 = 3$$

$$Q_{C_T}^8 = Q_{C_T^4}^8 = 3$$

From Equation 2.2, the total entropy can be found from

$$E_T^8 = \left(-\frac{Q_{C_0}^8}{Q_{C_T}^8} \log_2 \frac{Q_{C_0}^8}{Q_{C_T}^8} \right) - \left(\frac{Q_{C_1}^8}{Q_{C_T}^8} \log_2 \frac{Q_{C_1}^8}{Q_{C_T}^8} \right) = 0$$

The entropy calculation shows that node 8 is a leaf node as all the vectors of class 1. The DT has been updated as shown in Figure 4.12.

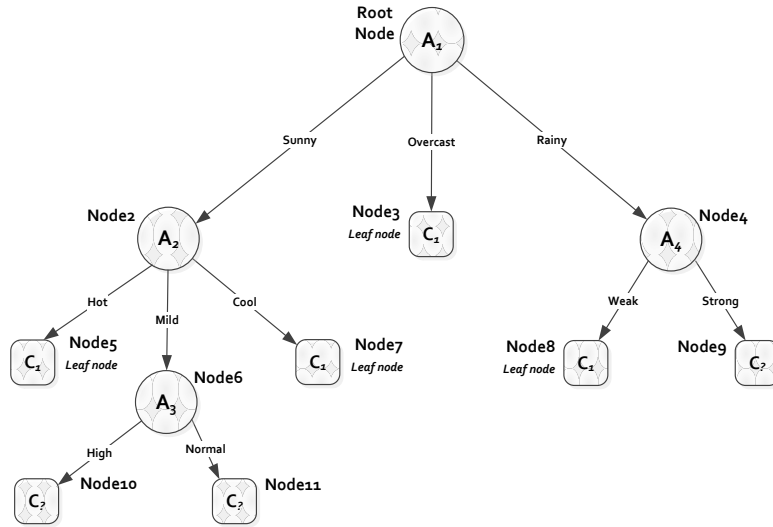


Figure 4.12: Node 8 is a leaf node.

Calculations of node 9

For second child node 9 to parent node 4 with attribute value 'strong', the total node entropy can be calculated using the following parameters that are given by Equations 4.54, 4.55 and 4.56.

$$Q_{C_0}^9 = Q_{C_0^1}^4 = 0$$

$$Q_{C_1}^9 = Q_{C_1^1}^4 = 3$$

$$Q_{C_T}^9 = Q_{C_T^1}^4 = 3$$

From Equation 2.2 the total entropy is given by

$$E_T^8 = \left(-\frac{Q_{C_0}^9}{Q_{C_T}^9} \log_2 \frac{Q_{C_0}^9}{Q_{C_T}^9} \right) - \left(\frac{Q_{C_1}^9}{Q_{C_T}^9} \log_2 \frac{Q_{C_1}^9}{Q_{C_T}^9} \right) = 0$$

The DT has been updated as shown in Figure 4.13, where the entropy calculation shows that node 9 is a leaf node as all vectors are of class 1.

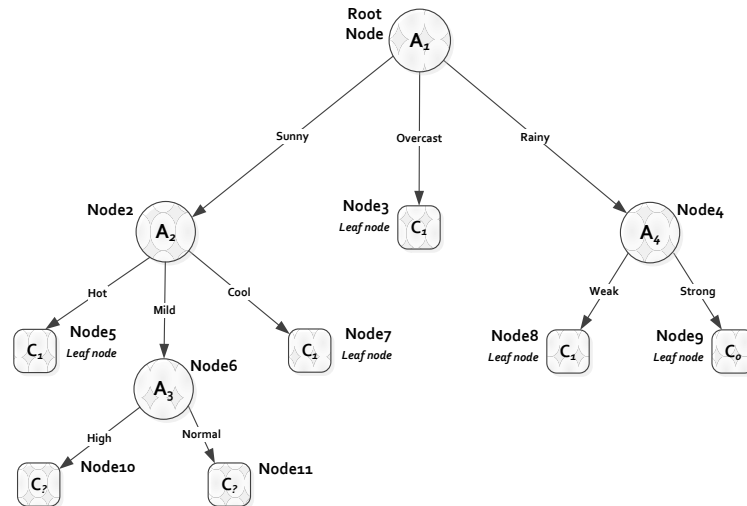


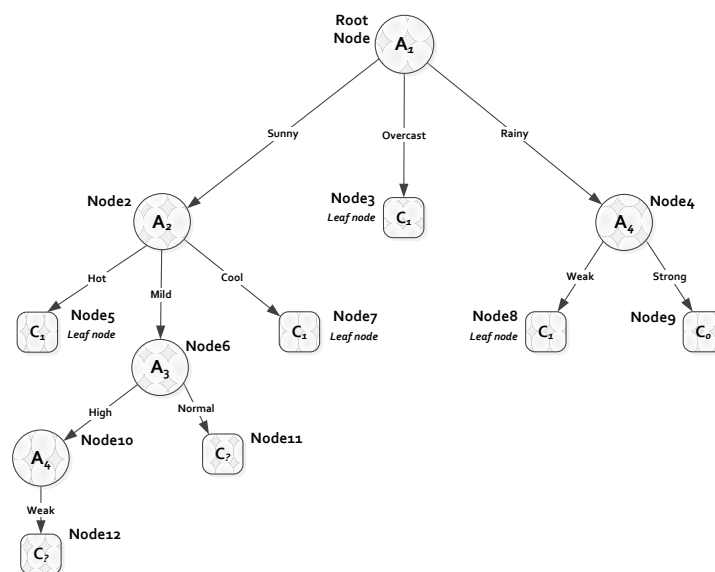
Figure 4.13: Node 9 is a leaf node.

Node 10

Next node is node 10 where its parent node 6 has chosen attribute 'humidity' as given by Equation 4.63. The relation considered at this stage is

Outlook-->Temperature-->Humidity-->Wind

For node 10 where the value is high, the number of vectors is 3 as given by Equation 4.59. The only attribute left is 'wind' and will be chosen for node10. The DT beyond this point has been updated as shown in Figure 4.14.

Figure 4.14: Attribute Wind (A_4) is selected for node 10.

Node 11

For node 11 where the value is normal, there is one vector as given by Equation 4.62. This node is a leaf node as the remaining vector is for class 1 and therefore no further calculations needed. The DT has been updated as shown in Figure 4.15.

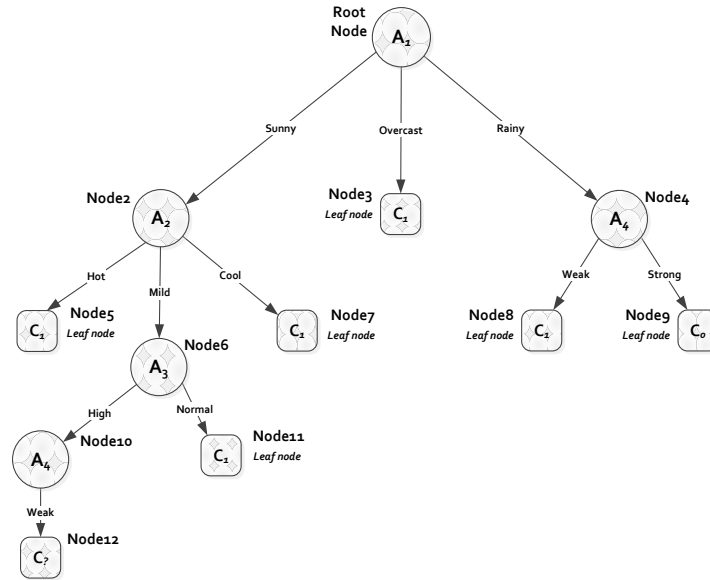


Figure 4.15: Node 11 is a leaf node.

Node 12

For node 12, it is considered as leaf node. The value 'weak' on this node has three vectors distributed as follows.

$$Q_{C_0}^{12} = 1$$

$$Q_{C_1}^{12} = 2$$

The structure of the node as shown in Figure 4.2, keeps statistics of the number of vectors for each class. Referring to the values of class *don't play* (0) as in $Q_{C_0}^{12}$ and *play* (1) as in $Q_{C_1}^{12}$ the probability of class *play* is twice the occurrence of class *don't play*. In this case the decision is choosing the class with higher probability of occurrence. The complete decision tree is shown in Figure 4.16, where node 12 is updated as a leaf node.

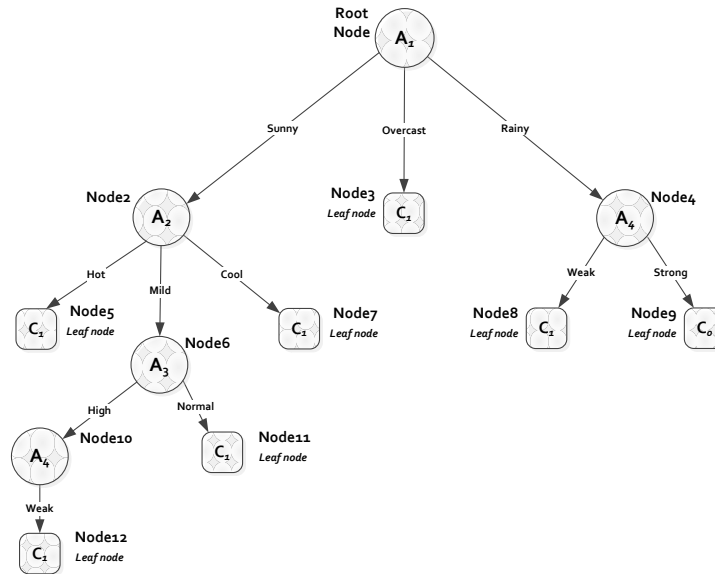


Figure 4.16: The completed DT, where Node 12 is a leaf node.

The classification Rules generated by the MDFT method are summarised as shown in Table 4.5.

Table 4.5: Classification (IF-THEN) rules model obtained by the MDFT method for the completed DT shown in Figure 4.16.

1	Outlook "Sunny" AND Temperature "Hot" \xrightarrow{THEN} Class "Play"
2	Outlook "Sunny" AND Temperature "Mild" AND Humidity "High" AND Wind "Weak" \xrightarrow{THEN} Class "Play"
3	Outlook "Sunny" AND Temperature "Mild" AND Humidity "Normal" \xrightarrow{THEN} Class "Play"
4	Outlook "Sunny" AND Temperature "Cool" \xrightarrow{THEN} Class "Play"
5	Outlook "Overcast" \xrightarrow{THEN} Class "Play"
6	Outlook "Rainy" AND Wind "Weak" \xrightarrow{THEN} Class "Play"
7	Outlook "Rainy" AND Wind "Strong" \xrightarrow{THEN} Class "Don't play"

4.9 EVALUATION OF THE MDFT METHOD

The purpose of this section is to evaluate the effect on the calculation time and memory requirements when using MDFT to classify a range of applications with a range of different numbers of attributes and attribute values. In the MDFT method, each attribute and the classes of any problem are represented as a dimension of the FT. In the code design of the MDFT method, entropy calculations call the search function *FindAttributeInstances* for the purpose of accessing the FT's values. The implementation details of the function can be found in Appendix B, but *FindAttributeInstances* includes a number of nested *for* loops, with the number of loops being directly proportional to the number of dimensions, and the number of iterations of each loop depending on the number of attribute values of each dimension. Consequently, the complexity of the search function increases with the number of dimensions.

The aim of the experiments is to demonstrate the effect of different numbers of attributes and attribute values on the execution time and memory requirements when using the MDFT method. The number of dimensions of the MDFT frequency table is equal to the sum of the number of attributes and the class, while the number of values represented in each dimension of the frequency table is the number of attribute values. MDFT was tested with 16 different problems that can be divided into two groups (A and B), where each group consists of eight different problems that share the same number of dimensions, but with different numbers of values of attributes and class.

The experiments were conducted under the Ubuntu 12.04 operating system [74] running as a Virtual Box machine [75] with 2.4GB of dedicated memory on 2.83GHz Intel Core 2 quad processor. The code for all the approaches was written in C and gcc version 4.6.3 [76] used to generate the executables. To generate the results shown in Figures 4.5 and 4.6, the vector values for experimental datasets were generated using *randi* function available in Matlab [77]. The examples of group A have a four dimensional frequency table which includes three attributes and a class. The number of attribute values for each dimension of the eight examples of group A are 4, 6, 8, 10, 12, 14, 16, and 18. Group B consists of a further eight examples that generate an eight dimensional frequency table, which includes seven

attributes and a class. The number of values for each dimension of the eight examples are 2, 3, 4, 5, 6, 7, 8, and 9. Because of the way in which each group has been designed, the total number of attribute values and classes for both groups is 16, 24, 32, 40, 48, 56, 64, and 72.

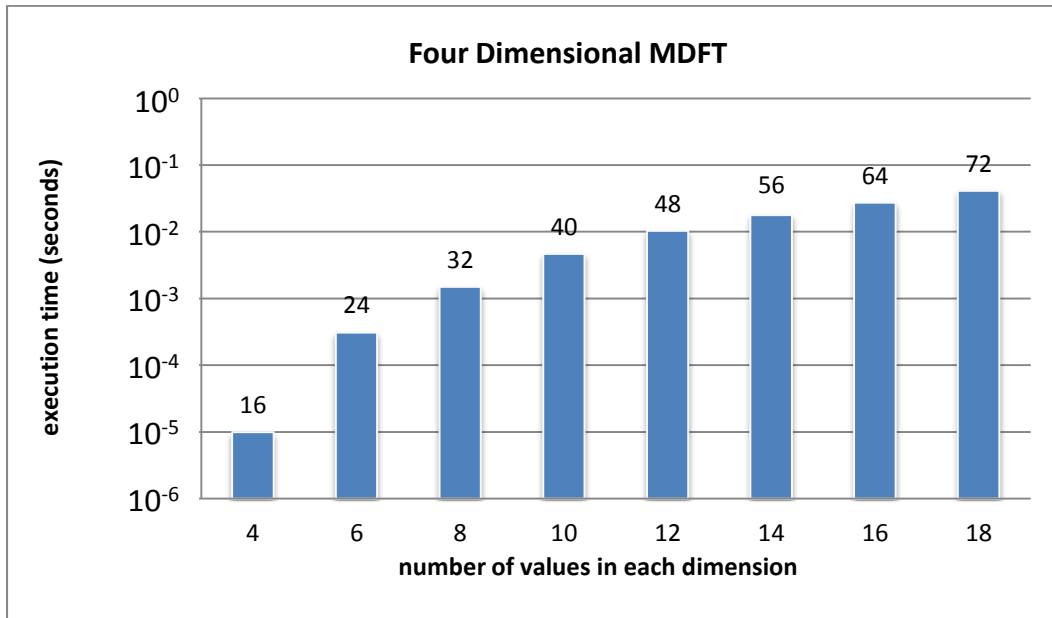


Figure 4.17: Decision tree calculation time of the example four dimensional MDFTs, each bar showing the total number of attribute values in each example.

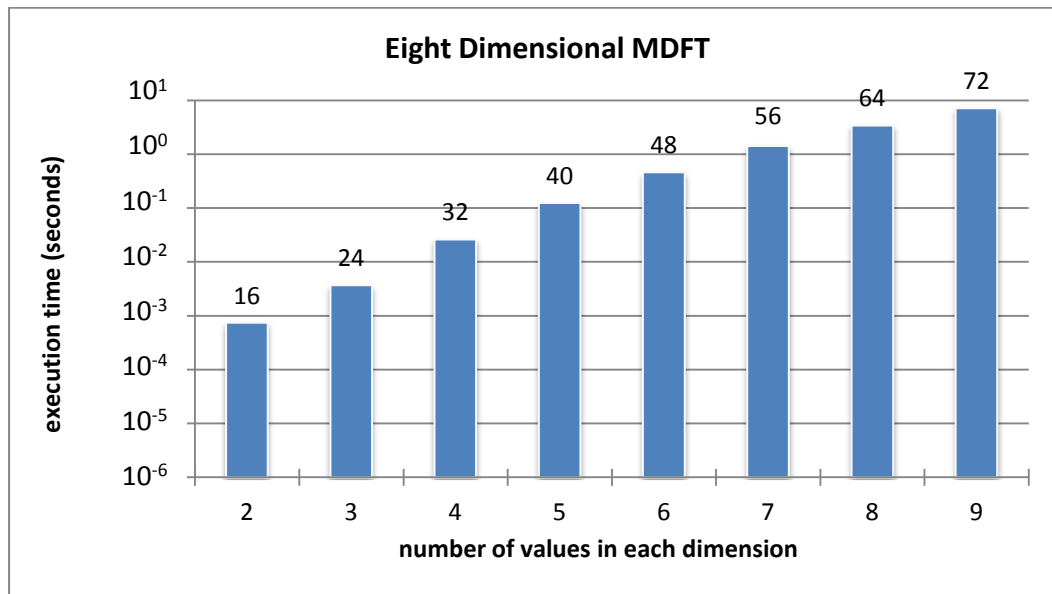


Figure 4.18: Decision tree calculation time of the example eight dimensional MDFTs, each bar showing the total number of attribute values in each example.

The results shown in Figures 4.17 and 4.18 demonstrate that the DT calculation time increases as the number of attribute values is increased for a given number of dimensions. The eight dimensional MDFT required more time to compute a decision tree than did the four dimensional MDFT for a given number of attribute values. It can also be noticed that the number of elements in the eight dimensional group increase substantially with attribute values when compared with the four dimensional group.

To explain the effect of the number of dimensions on the calculation time, Equation 4.65 shows the number of additions needed at the root node for the computations made by the search function *FindAttributeInstances*.

$$N_{ad}^R = 2 \cdot \left((N_a + 1) \cdot N_e - \left(\sum_{i=1}^{N_a} N_v[A_i] + N_c \right) \right) \quad (4.65)$$

On examination of Equation 4.65, it can be seen that the number of additions for the root node doubles when number of attributes increases by 1, assuming that the number of elements in the MDFT remains unchanged.

Equation 4.66 can be used to obtain the worst case for the number of additions at level l of the DT, if the attributes are organized in the tree so that the one with the largest number of values is at the root and the one with the fewest is at the lowest level.

$$N_{ad}(l) = \prod_{A_i=1}^{l-1} N_v[A_i] \cdot \left(\frac{(N_a - l) \cdot N_e}{\prod_{A_i=1}^{l-1} N_v[A_i]} - \sum_{i=1}^{N_a - l} N_v[A_i] \right) \quad (4.66)$$

Figures 4.19 and 4.20 show that, for the four and eight dimensional examples, the number of additions required increases with the number of attributes. The demand on the need for the addition operation in the *FindAttributeInstances* function is significantly greater for the eight dimensional MDFT compared with the four dimensional examples.

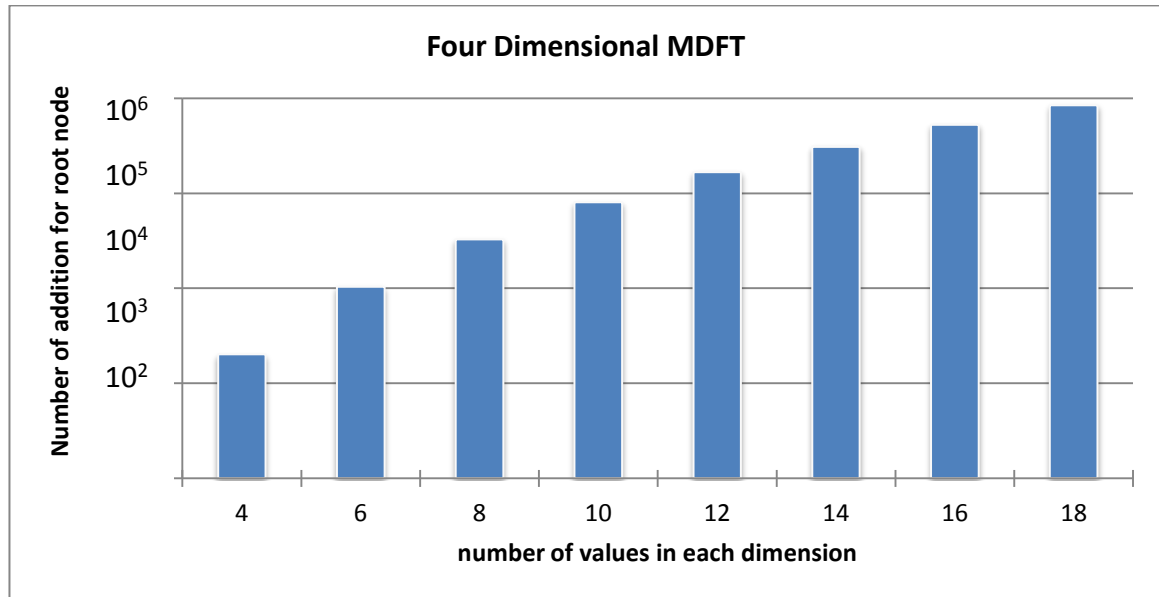


Figure 4.19: Number of additions needed at the root node of the example four dimensional MDFT

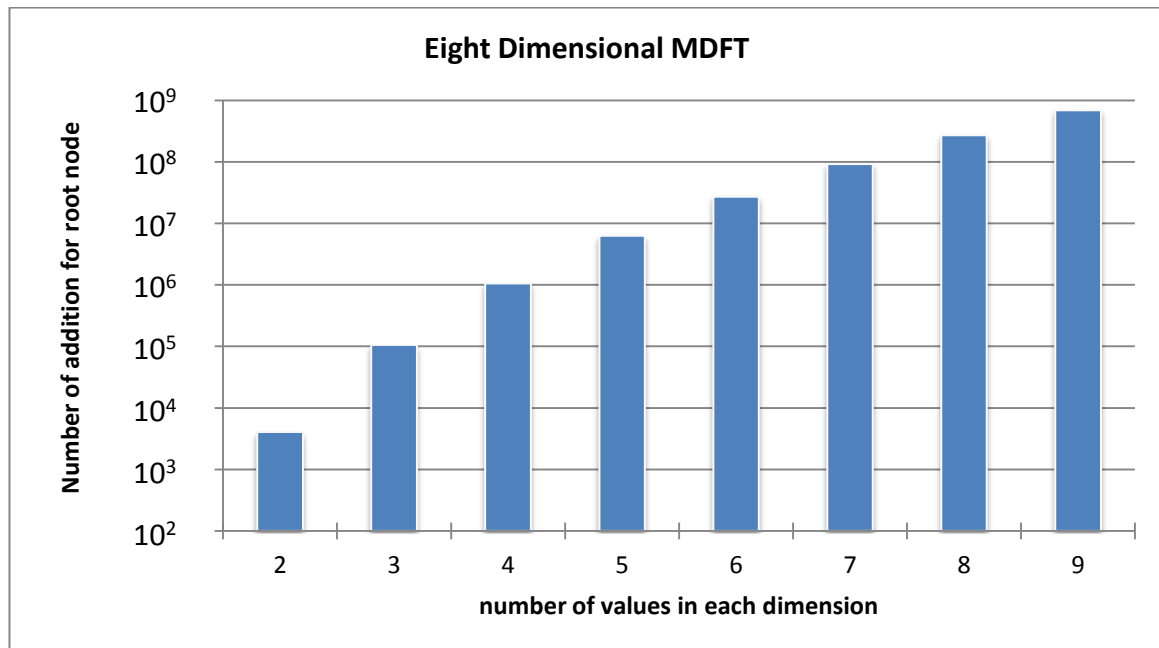


Figure 4.20: Number of additions needed at the root node of the example eight dimensional MDFT

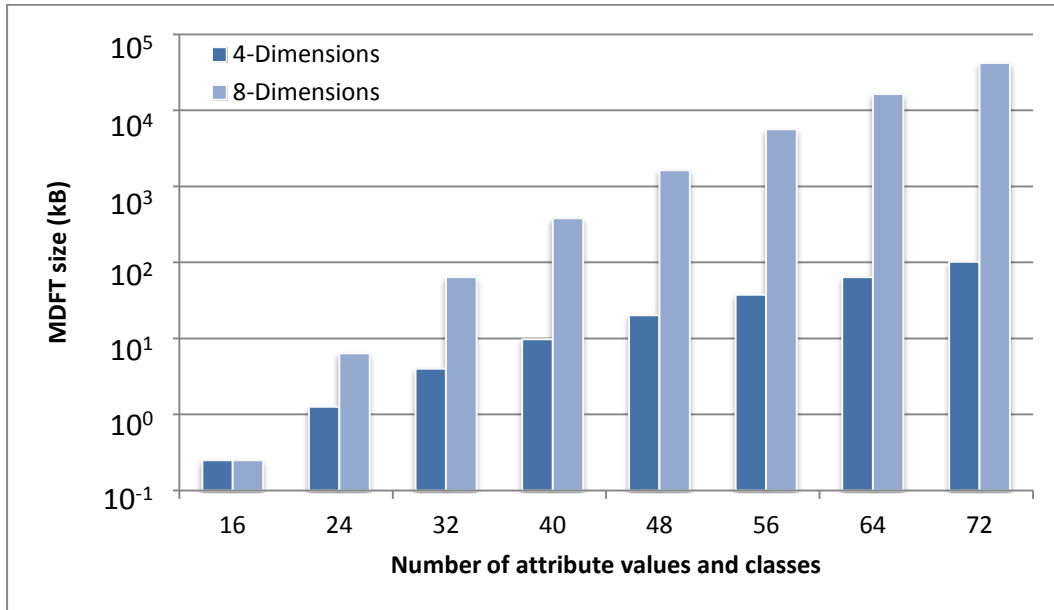


Figure 4.21: Memory requirements of the MDFT for the four and eight dimensional examples

The results shown in Figure 4.21 show that the memory consumed increases when moving from the four dimensions to the eight dimensions even when the total number of attribute values remain the same. This is due to the increase in the number of elements of the MDFT. Therefore having a fixed number of dimensions of the FT for different examples can reduce the memory requirements needed to generate the DT as this can be crucial to embedded systems where the memory is limited.

4.10 SUMMARY

This chapter has introduced a new incremental learning method termed MDFT. The multi-dimensional array acts as a frequency table that stores the incoming data vectors for further stages to build decision tree. MDFT successfully builds a complete DT as illustrated by the weather example, in contrast to previous methods using a frequency table which managed to build only two levels of the tree.

In this chapter, the worst case calculation scenario for the decision tree was presented where the complexity of the algorithm increases with the number of attributes.

The node structure used in the MDFT method ensures a fixed memory usage. The advantage of such structure is that it has a fixed node memory requirement where the maximum memory usage can be known *a priori*, making the approach suitable for embedded systems. The MDFT method is designed to target embedded systems where the efficient use of memory and effective performance are required. The memory usage for the MDFT method can be calculated and known in advance.

The MDFT approach has a limitation in that it does not scale well with the number of attributes and attribute values, and a substantial memory resource demand is incurred by the frequency table as the numbers of these values increases. MDFT can be applied in embedded systems as long as the memory requirements of the given application are considered *a priori* and matched to that available in the system. To overcome the memory requirement drawback of MDFT, the HFTDT approach is introduced in the next chapter and this proposes the use of a fixed number of dimensions instead of a multi-dimensional frequency table to store the data required to generate the DT.

5 HASHED FREQUENCY TABLE DECISION TREE

This chapter introduces and describes the details of the implementation of the Hashed Frequency Table Decision Tree (HFTDT) algorithm. The new method produces the same quality of decision tree as produced by its predecessor MDFT. The HFTDT and MDFT methods can be implemented as real-time learning methods, as the maximum number of calculations needed to produce a decision tree can be defined. The operation of HFTDT is demonstrated by means of an example that shows the operations that need to be conducted at each node of the decision tree.

5.1 INTRODUCTION

The HFTDT method is a development of MDFT. The MDFT results discussed previously in Section 4.9, showed that the memory required when targeting larger problems is unlikely to be available on most workstations or embedded systems. The HFTDT approach is able to reduce the memory required in the implementation of the frequency table by using a form of hash table. In this approach, only the active (non-zero) elements of the frequency table need to be allocated memory in the hash table. Also, the new technique reduces the number of dimensions of the frequency table array into a two-dimensional array that represents the hash table, which also reduces the memory required to hold the frequency table when compared with the MDFT approach.

The HFTDT method converts the frequency table into a hash table while keeping a record of every vector of the input data. The method includes two main functions: the index function which generates unique keys for the input vectors to be stored in the hash table; and a reverse index function that reads the stored keys and retrieves the original input vectors.

5.2 THE HASHED FREQUENCY TABLE

The hashed frequency table (HFT) structure shown in Figure 5.1, consists of an identifier (id) that holds a key value generated by the index function, and a counter that holds the number of occurrences of the key generated from the input vectors. The counter value for a key is set to one on creation and its value is incremented each time the vector is repeated in the input data.

```

/* Hashed Frequency Table entry structure */
struct my_struct {
    u64 id;                /* 64-bit key */
    u16 counter;          /* 16-bit holding the number of iterations of id */
};

```

Figure 5.1: HFT structure

The basic HFT structure reserves eight bytes for the key (id) and two bytes for the counter, although the number of bits used for the key and counter can be changed according to the requirements of the problem.

The number of bits required for storing the id can be determined as follows.

- Calculate the maximum possible number of keys P that can be generated for the targeted problem by finding the product of the number of values for all the attributes and classes (see Equation 5.1).

$$P = N_c \prod_{i=1}^{N_a} N_v[A_i] \quad (5.1)$$

- The number of bits required for id can be found by determining the number of bits needed to represent P . In the C language, typical examples are 8, 16, 32 and 64 bits, although other integral powers of two are possible if type libraries are used. In a hardware implementation, a wider range of possible values can be realised.

5.3 TECHNIQUES FOR THE INDEX AND REVERSE INDEX FUNCTIONS

This section demonstrates two techniques used for the implementation of each of the index and reverse index functions. The first technique is discussed in Sections 5.3.1 and 5.3.2, where the index and reverse index functions are realised using mathematical operations such as multiplication, division and modulo. The second technique as discussed in Section 5.3.4 and 5.3.5 uses only bitwise operators in the indexing functions such as AND, OR and shift. The advantage of using the second technique is to provide operations that are better suited to hardware implementation.

5.3.1 INDEX FUNCTION

The index function is used to generate a unique key for a given input vector. The generated key is then stored in the hash table. The method used for indexing depends on the number of attribute values and classes. Equation 5.2 shows a data vector for n attributes.

$$V_i = [v_{A_1}, v_{A_2}, v_{A_3}, \dots, v_{A_n}, C] \quad (5.2)$$

where v_{A_n} is the attribute value of attribute A_n and C is the class value in the input vector.

The method multiplies each attribute value of the input vector with an incremental product of the number of attribute values and classes. Then it sums the results with class value to give a unique key.

Equation 5.3 is used to calculate the key

$$key = N_c \left(\sum_{k=1}^{N_a-1} v_{A_k} \left(\prod_{i=0}^{N_a-2} N_v[A_{(N_a-i)}] \right) + v_{A_{N_a}} \right) + C \quad (5.3)$$

where N_c is the number of classes, N_a is the number of attributes and $N_v[A_{(N_a-i)}]$ is the number of attribute values for attribute $A_{(N_a-i)}$

5.3.2 REVERSE INDEX FUNCTION

In order to access the required keys in the hash table for the process of building the DT, a 'reverse index function' is required. The purpose of the reverse index function is to retrieve a copy of the original input vector that contains the correct attribute and class values.

Equations 5.4 and 5.5 operate in a recursive manner to determine the original input vector. The 'mod' operation used in Equation 5.4 returns the remainder of its argument and v_{A_j} is the value of its argument rounded down to the nearest integer. To explain the process, consider when $i = 1$ (the initial value of i). From Equation 5.4 the value k_1 is the input key for the first attribute and the resulting value v_{A_1} is its attribute value. From Equation 5.5, the value k_2 is calculated to be used in Equation 5.4 to determine v_{A_2} . The process is repeated until $i = N_a + 1$ and the vector is retrieved.

$$v_{A_i} = \left\lfloor \frac{k_i}{N_c \prod_{i=1}^{N_a-1} N_v[A_{i+1}]} \right\rfloor \quad (5.4)$$

$$k_{i+1} = k_i \bmod \left(N_c \prod_{i=0}^{N_a-1} N_v[A_{i+1}] \right) \quad (5.5)$$

Above, the k_i are the input key values, $v_{A_{(N_a+1)}}$ are the attribute values, N_a is the number of attributes, N_c is the number of classes and $N_v[A_{i+1}]$ is the number of attribute values for attribute A_{i+1} .

Equation 5.6 represents the expected output vector of the reverse index function

$$V_o = [v_{A_1}, v_{A_2}, v_{A_3}, \dots, v_{A_{N_a}}, C] , \quad (5.6)$$

where C is the class value in the output vector.

5.3.3 AN ILLUSTRATIVE EXAMPLE

To facilitate the understanding of the first hash table technique, a small illustrative example is now presented. The example inputs a vector to the index function to generate a key, and then the key is provided to the reverse index function to generate an output vector. The verification of correctness is that the input and output vectors are identical.

Consider the following example where the dataset consists of three attributes with four values each and three classes. Assume an input vector $V = (2, 0, 1, 1)$.

1. The first step uses the index function to generate a key for the input vector and this can be done using Equation 5.3 as follows:

$$key = 3 \cdot ((2 \cdot 4 \cdot 4) + (0 \cdot 4) + 1) + 1 = 100 \quad (5.7)$$

2. The converse operation is performed by the reverse index function to generate an output vector for the key obtained. The output vector can be calculated using Equations 5.4 and 5.5 as follows:

$$v_{A_1} = \left\lfloor \frac{100}{3 \cdot 4 \cdot 4} \right\rfloor = \left\lfloor \frac{100}{48} \right\rfloor = 2 \quad , \text{where } 100 \text{ is the input key} \quad (5.8)$$

$$k_2 = 100 \bmod (48) = 4$$

$$v_{A_2} = \left\lfloor \frac{4}{3 \cdot 4} \right\rfloor = \left\lfloor \frac{4}{12} \right\rfloor = 0 \quad (5.9)$$

$$k_3 = 4 \bmod (12) = 4$$

$$v_{A_3} = \left\lfloor \frac{4}{3} \right\rfloor = 1 \quad (5.10)$$

$$k_4 = 4 \bmod (3) = 1 \quad (5.11)$$

Consequently, the output vector is $V_o = (2, 0, 1, 1)$, in agreement with the input vector. Further verification tests were carried out using a large number of input vectors generated by a number of the test examples used in this work.

5.3.4 ALTERNATIVE INDEX FUNCTION TECHNIQUE

The alternative technique adopts bitwise operators to implement the same functionality as described in the previous section. The index function simply applies an OR and right shift operation to the input vector to generate a unique key.

The left shift operation is analogous to the multiplication used in the index function described in Section 5.3.1. A left shift by 1 position is analogous to multiplying by 2. However, the operation available for performing such products is limited to multiplications by integral powers of 2.

To demonstrate how index function works few preparation steps needed to consider at the beginning, which are:

- a) Create a vector that holds the number of values of each attribute and class.
- b) Create a second vector *num_att_values* of length $N_a + 1$ that holds the smallest integral power of 2 that is no less than the corresponding vector created in step a).
- c) Generate a vector *acc_num_att_values* with $N_a + 1$ entries that holds the cumulative values of *num_att_values*, then take the logarithm to the base 2 of each element.

The key is then generated as follows.

- a) Set the key to first element of the input vector.
- b) The next element of the input vector is left shifted by the number of bits defined in corresponding element *acc_num_att_values* and then ORed with the current value.
- c) The process is repeated (return to b)) until all the elements of the input vector have been accessed. The recursive calculation of the key is given by

$$\begin{aligned}
 key &= input_vector [0], \\
 key &= key | (input_vector [i] \ll acc_num_att_values[i-1], i = 1, 2, \dots, N_a.
 \end{aligned}
 \tag{5.12}$$

5.3.5 ALTERNATIVE REVERSE INDEX FUNCTION TECHNIQUE

The alternative reverse index function is modification of the technique given in Section 5.3.4. The reverse index function uses the vectors generated in steps b) and c) of Section 5.3.4 to generate the output vector.

The 'modulo' and the 'division' operations used in the reverse index function in Section 5.2.2 are substituted as follows.

$x \& (y - 1)$ is analogous to $\text{mod}(x, y)$ where $\&$ is the bitwise AND operation (5.13)

$x \gg \log_2(y)$ is analogous to (x / y) where \gg is shift right operation. (5.14)

The above equations assume y is an integral of power 2.

The following are the steps taken in the reverse index function to produce an output vector based on the key by utilizing Equations 5.13 and 5.14.

- a) As shown in Equation 5.15, the first entry of the output vector is the bitwise ANDing of the key and the first entry of the vector *num_att_values* (Section 5.3.4).

$$\text{output_vector}[0] = \text{key} \& (\text{num_att_values}[0] - 1) \quad (5.15)$$

- b) To produce the remainder of the output vector, Equation 5.14 is applied, involving right shift and bitwise AND operations. The process is repeated until the final element of the output vector is produced

$$\text{output_vector}[i] = (\text{key} \gg \text{acc_num_att_values}[i - 1]) \& (\text{num_att_values}[i] - 1) \quad i=1,2,\dots,N_a. \quad (5.16)$$

5.4 HFTDT METHOD CALCULATIONS

To consider the implementation of the HFTDT method for a real-time system, this section determines the maximum number of calculations required to generate a decision tree under the worst case scenario and obtains the required memory capacity.

5.4.1 MAXIMUM NUMBER OF CALCULATIONS (WORST CASE SCENARIO)

The HFTDT method shares the same number of calculations required to generate a decision tree, including for the worst case scenario, as the MDFT method (Section 4.5).

5.4.2 CALCULATING MEMORY USAGE

The HFTDT method is suitable for real-time applications, as memory usage can be determined in *a priori*. The memory used to build the decision tree can be determined as follows.

The hashed frequency table (HFT) structure was shown in Figure 5.1 and the number of bytes it occupies is given by

$$S_{HFT} = N_e \cdot (8 + 2) = 10N_e \quad (5.17)$$

where N_e is the number of entries in the HFT, each consisting of a 64-bit id (8 bytes) and a two byte counter.

The HFTDT method shares the node structure with MDFT, shown in Figure 4.2. The structure maintains a fixed memory usage for the nodes of the decision tree, and therefore the maximum memory usage can be calculated and be known *a priori*.

As can be found in Figure 4.2, the memory occupied by each node can be calculated as follows

$$S_N = 1 + 2 + 2 + 2 + 1 + 1 = 9 \text{ bytes} . \quad (5.18)$$

Combining Equations 5.17 and 5.18, the total memory usage of the HFTDT decision tree in bytes is given by

$$S_T = S_{HFT} + (S_N \cdot N_{Nodes}) \quad (5.19)$$

where N_{Nodes} is the number of nodes in the decision tree where a maximum number of nodes can be set

5.5 GENERATING THE DT USING HFTDT

HFTDT is a development of its predecessor MDFT, but modified such that the storage technique for the input data vectors uses a hashed frequency table. Both the HFTDT and the MDFT methods depend on the entropy and information gain calculations to build a decision tree (refer to Section 4.4). The HFTDT algorithm is designed to allow the data stored in the hash table to be extracted one at a time through the *rev_index_func* function to provide the necessary information needed for entropy calculations at an individual node. This procedure is repeated for every node in the DT where the process cycles through the stored data as long as an open node exists, where open node is any node that needs to be investigated.

The algorithm HFTDT is shown in Figure 5.2, where the input is the stored data in the hashed frequency table and the output is a decision tree that summarises all the attributes' correlations. Referring to the algorithm in Figure 5.3, several steps need to be conducted to achieve the storage of the element in the HFT.

A set of rules is generated after completing the DT. The generated rules instantiate the knowledge obtained from DT, which is then stored to be used later in the classification of test data.

Input: Hashed Frequency Table (HFT); set of attributes; training vector and unclassified vector	
Outputs: A Decision Tree, generated rules and classified vector	
Start	
1.	Update the Hashed Frequency Table (HFT) following the arrival of each new training vector
2.	IF all the stored entries in the HFT are of the same class THEN produce a DT with a single node (Leaf Node) labeled with that class, go to step 7 ELSE create a node in the DT
3.	Select an attribute among the set of attributes with the highest IG value and label the node with selected attribute
4.	Add a branch for each known value of the selected attribute for that node.
5.	Create a node for each branch
6.	IF all the HFTDT entries for this node are of the same class THEN label the node as a leaf node of that class ELSE compute the entropy value and the IG for all attributes, select the attribute of the largest IG and use this attribute for the current node, go to step 4
7.	Generate rules from the DT
8.	Store the DT and the rules and use as required in the classification of new vectors
9.	IF more training is required THEN go to step 1
End	

Figure 5.2: Description of the HFTDT algorithm

5.6 AN ILLUSTRATIVE EXAMPLE FOR HFTDT METHOD

This section shows a fully worked example that demonstrates how to generate a decision tree using the HFTDT method. The weather problem [11] introduced in Table 4.1 is used.

Table 5.1 shows a map of all the keys that can be generated for the weather problem. The keys can be obtained using the index function given in Section 5.3.1

Table 5.1: A map of the keys of the weather problem

		Outlook (A_1)											
		Sunny	Overcast	Rainy	Sunny	Overcast	Rainy	Sunny	Overcast	Rainy			
Play	'humidity' (A_3)	High	1	25	49	9	33	57	17	41	65	Wind (A_4)	Weak
		Normal	5	29	53	13	37	61	21	45	69		Weak
		High	3	27	51	11	35	59	19	43	67		Strong
		Normal	7	31	55	15	39	63	23	47	71		Strong
Don't Play	'humidity' (A_3)	High	0	24	48	8	32	56	16	40	64	Wind (A_4)	Weak
		Normal	4	28	52	12	36	60	20	44	68		Weak
		High	2	26	50	10	34	58	18	42	66		Strong
		Normal	6	30	54	14	38	62	22	46	70		Strong
		Hot	Hot	Hot	Mild	Mild	Mild	Cool	Cool	Cool			
		Temperature (A_2)											

Table 5.2 shows the data as it is stored in hash table. Note that the hash table is a compact version of the MDFT as it holds only its non-zero elements. The HFT is effectively a two dimensional table as it contains elements that themselves consist of two elements.

Table 5.2: Hashed frequency table of the weather problem

id	Counter
0	1
2	1
25	1
57	1
9	2
69	1
70	1
35	1
8	1
21	1
61	1
15	1
29	1
47	1
58	1

A function f is defined to provide the number of input vectors as recorded in each of the HFTDT elements as shown below.

$$\text{number of vectors} = f(id) \quad (5.20)$$

where id represents a key in the hash table .

The weather problem given in Table 5.2 consists of four attributes ($N_a = 4$) and two classes ($N_c = 2$). The HFTDT method uses an HFT that holds the keys and the corresponding values of each input vector.

After the HFT has been populated, the decision tree can be generated. The splitting criterion used in generating the decision tree is information gain.

The following shows the calculation of the total entropy which is required at for the root node, see Equation 2.2.

$$\begin{aligned} Q_{C_0}^1 &= \sum \text{Values in HFT where the class} = (\text{Don't Play}) \\ &= f(0) + f(2) + f(8) + f(58) + f(70) = 5 \end{aligned} \quad (5.21)$$

$$\begin{aligned} Q_{C_1}^1 &= \sum \text{Values in HFT where the class} = (\text{Don't Play}) \\ &= (f(9) + f(15) + f(21) + f(25) + f(29) + f(35) + f(47) + f(57) \\ &\quad + f(61) + f(69)) = 11 \end{aligned} \quad (5.22)$$

$$Q_{C_T}^1 = \sum \text{Values in HFT of all classes} = Q_{C_0}^1 + Q_{C_1}^1 = 16 \quad (5.23)$$

The total entropy at the root node is thus

$$\begin{aligned} E_T^1 &= \left(-\frac{Q_{C_0}^1}{Q_{C_T}^1} \log_2 \frac{Q_{C_0}^1}{Q_{C_T}^1} \right) - \left(\frac{Q_{C_1}^1}{Q_{C_T}^1} \log_2 \frac{Q_{C_1}^1}{Q_{C_T}^1} \right) \\ &= \left(-\frac{5}{16} \log_2 \frac{5}{16} \right) - \left(\frac{11}{16} \log_2 \frac{11}{16} \right) = 0.896038 \end{aligned} \quad (5.24)$$

The values obtained are the same as those using MDFT (Section 4.8). The complete implementation of the example using HFT can be found in Appendix A. The decision tree generated is shown in Figure 4.16 and the classification rules are shown in Table A.1.

5.7 COMPARISON BETWEEN HFTDT AND MDFT

This section demonstrates a comparison between the HFTDT and MDFT methods. Table 5.3 shows the comparison in terms of the number of dimension of the frequency table used, memory requirements by the frequency table and the suitability for embedded systems applications.

Table 5.3: comparison between the HFTDT and MDFT methods

	MDFT	HFTDT
Number of dimension of the frequency table	Variable <ul style="list-style-type: none"> depends on the number of attributes and class) 	Fixed (2-Dimensional hash table)
Memory requirements for the frequency table	Fixed <ul style="list-style-type: none"> increase substantially with number of dimensions 	Variable <ul style="list-style-type: none"> much less than MDFT depends on the number of input vectors
Maximum number of calculations	<ul style="list-style-type: none"> Can be known <i>a priori</i> 	<ul style="list-style-type: none"> Can be known <i>a priori</i>
Embedded systems Suitability	Suitable for smaller problems <ul style="list-style-type: none"> depends on the memory available by the system 	Suitable for larger problems <ul style="list-style-type: none"> depends on the memory available by the system

From Table 5.3, the comparison between MDFT and HFTDT methods can be summarised as follows. The memory requirements of the MDFT can be known *a priori*, but can increase substantially with the number of dimensions of the FT, while HFTDT achieves a reduction in memory usage as it adopts a two-dimensional hash table to hold the active non-zero elements of the MDFT frequency table. The memory requirements of HFTDT depend on the number of elements stored in the hash table, which can increase with the presence of new vectors. The two methods have a deterministic time in which the number of calculations can be known in advance. The MDFT frequency table size must not exceed the memory available in the system, while in HFTDT the number of unique data vectors determines the number of HFT elements that are required and hence the memory requirement.

5.8 SUMMARY

This chapter has described a novel decision tree algorithm termed HFTDT. The HFTDT uses a hashed frequency table as storage for the input vectors.

HFTDT generates a compressed version of the MDFT table, with hashed frequency table that contains only non-zero elements. The HFTDT method uses a two-dimensional array to represent the hash table, with the aim of achieving a substantial reduction in the memory required to hold the frequency table compared to MDFT approach. Consequently, HFTDT will be able to represent larger problems than MDFT using less memory

HFTDT has two main functions, the index and reverse index functions. The index function is capable of generating unique keys for the input vectors to be then saved in the hash table, where the reverse index function can read the hash table keys and generate the vectors. Two techniques for indexing and reverse indexing are discussed in this chapter. The first technique depends on arithmetic multiplications, divisions and modulo operations. While the second technique depends on bitwise operators such as AND, OR and left or right shift. Both techniques perform similar functionality; the second technique was originally intended mainly for hardware implementation, which will be discussed later in Chapter 7, but the approach was also found to reduce the calculation time for the software implementations described in the next chapter.

6 EXPERIMENTS TO GENERATE DECISION TREES USING HFTDT

This chapter presents the experimental validation of HFTDT as a machine learning method to assess its suitability usage in implementation on embedded system platforms where the memory and computation time are the main resource constraints. The aspects tested in the experiments are classification accuracy, computation time, scalability, robustness and memory usage. The experiments compare HFTDT with three widely used machine learning methods, namely kNN, C4.5 and ITI.

6.1 INTRODUCTION

The main criteria [78] that are widely followed in evaluating and comparing classification methods in machine learning can be summarized as follows.

- Accuracy of the classifier. This refers to how well the classifier can predict the class label of input data vectors. Testing should be carried out using previously unseen datasets so as to assess the general classification performance of the learning system. Testing using the training data can only assess the specific classification performance for that dataset.
- Computational time. This refers to the time taken by the classifier to build and generate the DT in order to measure the computational cost of executing the algorithmic calculations.
- Scalability, is the ability of the classification algorithm to continue to act efficiently as the application becomes more complex. The execution time must remain acceptable as the computational cost increases.
- Robustness. This refers to the ability of the classification algorithm to continue to perform satisfactorily even when the data supplied is noisy or

contains missing values. One of the datasets selected for investigation in the current work contains missing values.

- Interpretability refers to the level of understanding and insight that is provided by the classification algorithm. This measure can be subjective and difficult to assess. It is not relevant in the comparison of methods that all generate output of same type and so these criteria were not used in the current study.
- Memory usage of the classification algorithm during execution. Memory usage increases with the number of vectors used for training and the complexity of the rule system needed to classify new test data [79]. Generally, such additional memory requirements come with a commensurate increase in computational resources [80].

6.2 CLASSIFIERS USED IN THE EXPERIMENT

The HFTDT method was compared with three classification methods, namely kNN, where the C source code can be obtained from Ostlund [81], C4.5 where the C source code can be obtained from Quinlan [82] and ITI where the C source code can be obtained from Utgoff [83]. C4.5 and ITI are DT classifiers, where ITI supports incremental learning and C4.5 does not. kNN (k-nearest neighbour) is a well-known statistical classification approach that has been intensively used in the literature as a benchmark to assess other machine learning methods [84].

6.3 TEST DATASETS

The HFTDT method has been tested using three different datasets, namely nursery, agaricus-lepiota and chess (KRKPA7), all of which are available from the UCI machine learning repository [85]. The datasets have been collected by researchers for the purpose of carrying out experiments in their particular fields of interest. The datasets represent a range of numbers of attributes, attribute values and classes. A brief description of the datasets is now given.

Nursery

The nursery dataset rank the applications made to nursery schools in Ljubljana, Slovenia during the 1980s at which time an objective explanation was required to justify rejected applications.

The nursery dataset has five classes, which are not recommend, recommend, very recommend, priority and special priority. The eight attributes have between two and five categorical values to describe the evaluations carried out on applications. These are the parents' occupations, child's nursery, form of the family, number of children, housing conditions, financial standing of the family, social conditions and health conditions. The dataset consists of 12960 feature vectors of which 8640 are for training and 4320 for testing.

Agaricus-lepiota

The agaricus-lepiota dataset holds the recorded characteristics of mushrooms drawn from the Audubon Society field guide to North American mushrooms. There are 22 attributes and 8124 feature vectors in the dataset, 5416 for training and 100 vectors for testing. The attributes are cap-shape, cap-surface, cap-colour, bruises?, odour, gill-attachment, gill-spacing, gill-size, gill-colour, stalk-shape, stalk-root, stalk-surface-above-ring, stalk-surface-below-ring, stalk-colour-above-ring, stalk-colour-below-ring, veil-type, veil-colour, ring-number, ring-type, spore-print-colour, population and habitat. The attribute values are all categorical and there are two classes namely edible and poisonous. The dataset contains around 30% missing values in the stalk-root attribute.

Chess

The chess dataset records the results of the chess end-game (a white king and rook versus a black king-and pawn with the latter on A7, usually abbreviated KRKPA7). It is the white's turn to move. The dataset has 36 attributes and 3196 feature vectors in the dataset, where 2130 are used for training and 1066 are for testing.

Each attribute corresponds to a particular position on the board, namely A00, A01...A35. The attribute values are all categorical and there are two classes, white can win (won) and white cannot win (no-win).

The characteristics of the datasets are summarized in Table 6.1. The HFT structure introduced in Section 5.2, has a key whose memory requirement in number of bits can be determined by calculating the product of the number of attribute values with the number of classes.

Table 6.1: Summary of dataset characteristics

Dataset name	Attributes (N_a)	Classes (N_c)	Attribute values	Product of number of attribute values and classes	HFT key (bits)
Nursery	8	5	27	6.48×10^4	16
Agaricus- lepiota	22	2	126	3.28×10^{15}	52
Chess	36	2	73	2.06×10^{11}	38

6.4 TEST RESULTS

The datasets consist of a training set and a test set. The training vectors were randomly assigned to a number of subsets to allow the progression of training to be assessed using the test set as additional training vectors are provided. Referring to Section 6.1, the criteria investigated were the number of nodes in the DT, computation time for training, memory usage and the classification accuracy for unseen datasets. The experiments were conducted under the Ubuntu 12.04 operating system [74] running as a Virtual Box machine [75] with 2.4GB of dedicated memory on 2.83GHz Intel Core 2 quad processor. The code for all the approaches was written in C and gcc version 4.6.3 [76] used to generate the executables.

In the next subsections, all the points on the graphs are points in which the HFTDT apply incremental learning. The training stops every few hundreds of training vectors being read and used to build a DT. The classification data is then applied to calculate the classification error.

6.4.1 NUMBER OF NODES

This experiment measures the number of nodes in the DTs produced by HFTDT, C4.5 and ITI. No kNN results were produced as there is no concept of nodes in the technique. C4.5 and ITI both adopt pruning techniques in an attempt to generalise and produce a smaller DT, HFTDT does not implement any pruning (as explained in Section 2.3) which in general tends to produce a larger, less generalised tree.

The results in Figures 6.1 to 6.3 for each of the three datasets show that ITI produced relatively smaller DTs and this is as a result of the heavy pruning the method adopts. Figures 6.1 and 6.3 show that HFTDT produced DTs smaller in size than the trees produced by C4.5, although for the agaricus-lepiota dataset in Figure 6.2, it can be seen that HFTDT produces a relatively large DT which occurs due to the presence of missing values in the training set. The percentage of vectors having missing values in each subset of the agaricus-lepiota training set that holds more than 3000 records is in the range 15% to 30%. As the HFTDT algorithm does not employ pruning, each missing value is dealt with as an extra attribute value resulting in increase in the number of nodes compared with ITI. The tree continues to grow as more training vectors are applied, and although the DT satisfactorily classifies the training data, this leads to overfitting. In general and without pruning, the more noise there is in the dataset or the more prevalent are missing values in the training set, the greater will be the likelihood of developing a DT that incorporates nodes and branches that represent spurious vectors that do not reflect the underlying nature of the dataset.

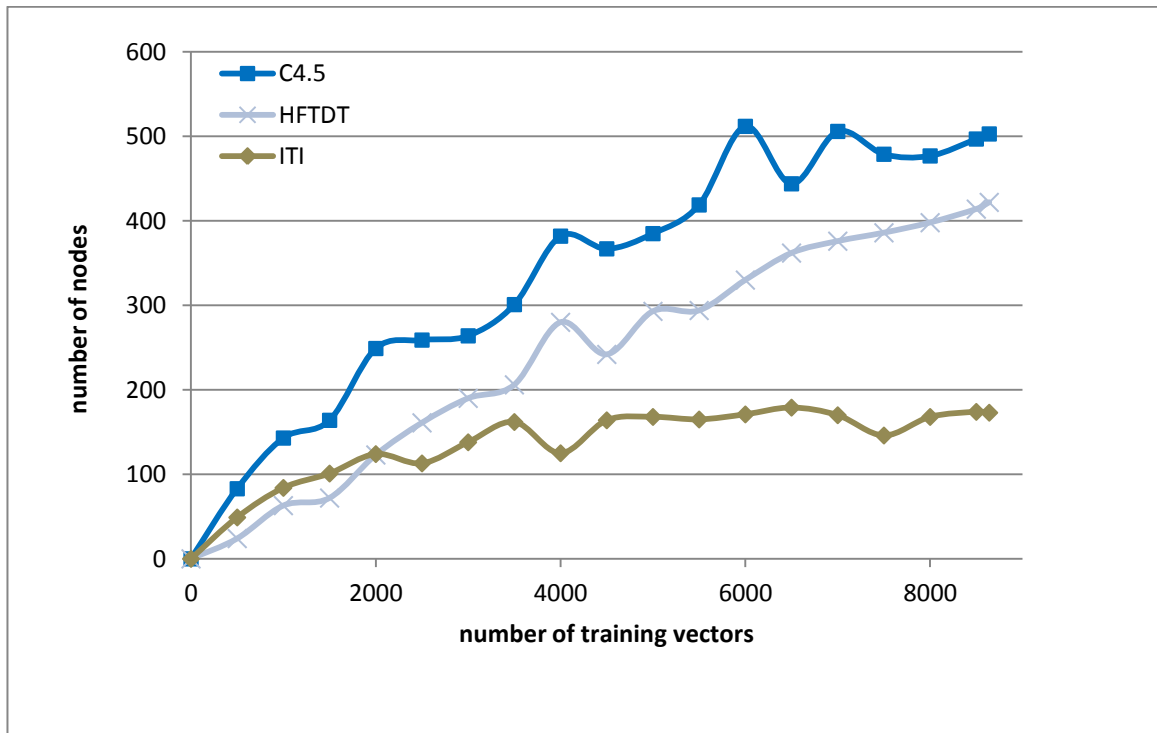


Figure 6.1: Number of nodes for the nursery dataset

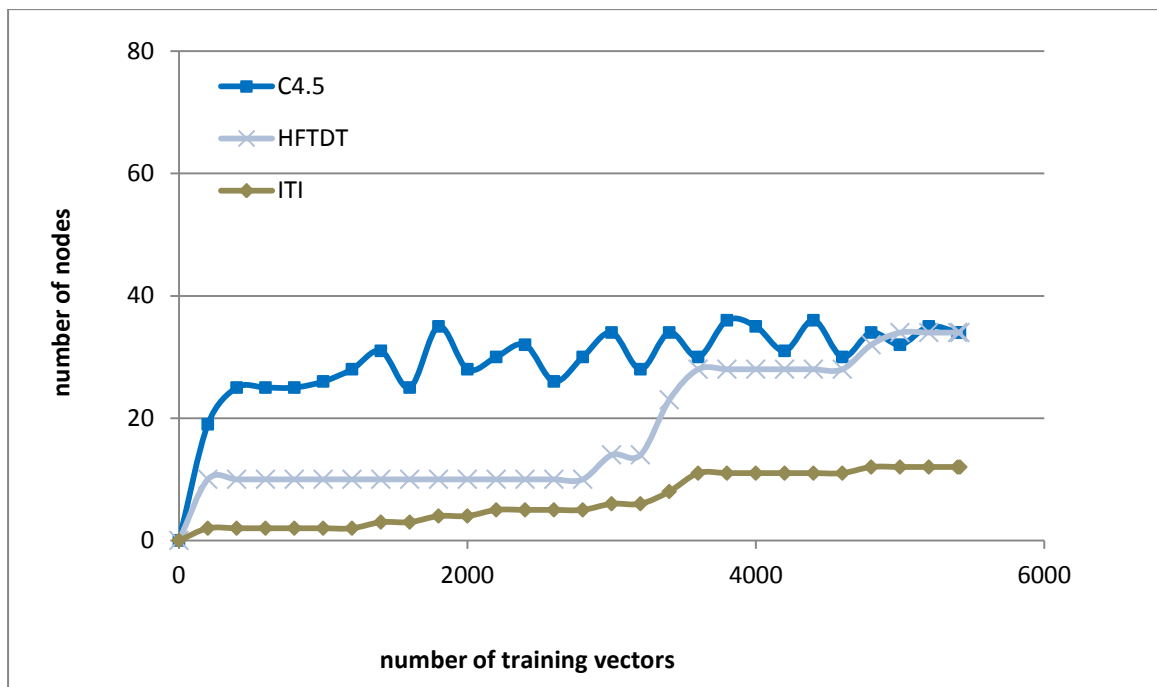


Figure 6.2: Number of nodes for the agaricus-lepiota dataset

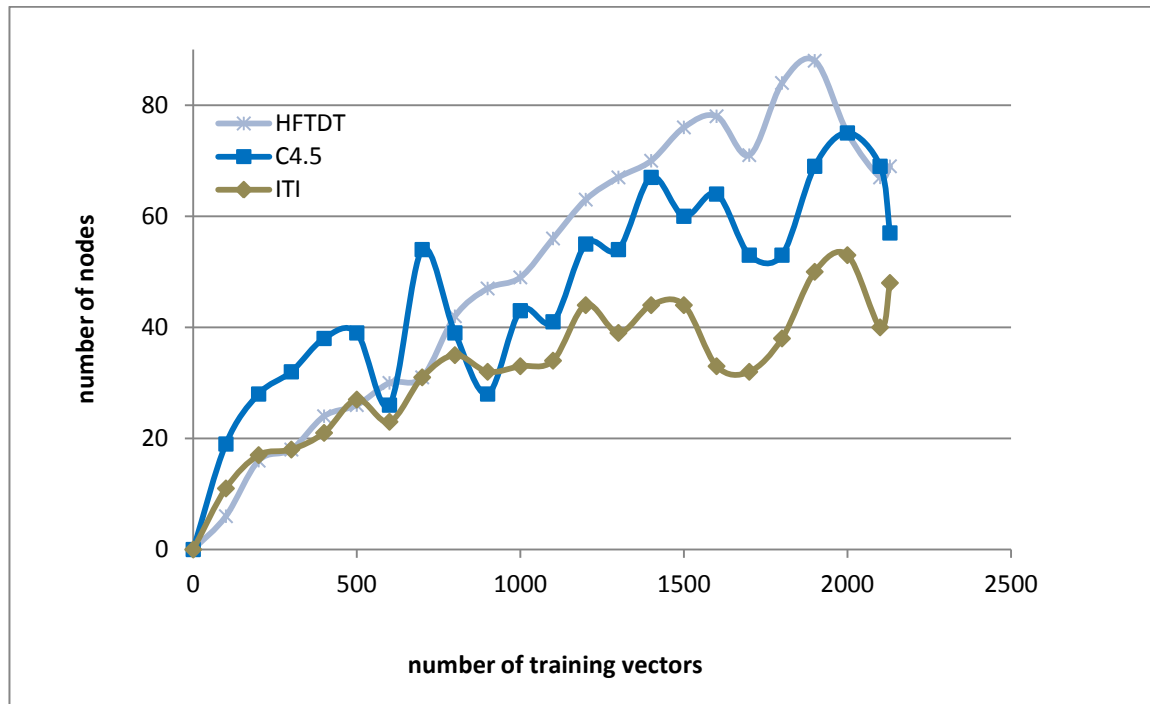


Figure 6.3: Number of nodes for the chess dataset

6.4.2 CALCULATION TIME

This part of the experiment compares the execution time of the four classifiers when applied to the three datasets. The results of all datasets show that kNN has the longest classification calculation time as it relies on comparing every test vector with the all training vectors [86] and it takes longer to execute (6 to 40 times) when compared to HFTDT.

However, the performance of HFTDT is slower than ITI and C4.5 as revealed in Figures 6.4 to 6.6. This is due to the approach taken in the HFTDT algorithm design that is aimed towards providing a hardware solution that is capable of implementing multiple entropy calculations for a group of nodes which are fed by the sequential reading of data stored in the hash table. The solution in the sequential software environment can only be carried out for a single node at a time,

causing an increase in execution time compared to other DT methods, since the HFT accesses need to be repeated for each node. Further, such feeding of HFT data to the individual entropy and information gain calculations must also be performed sequentially in software, whereas the intention in the design of the HFT is of a hardware realisation that allows parallel transmission of the data to such calculations.

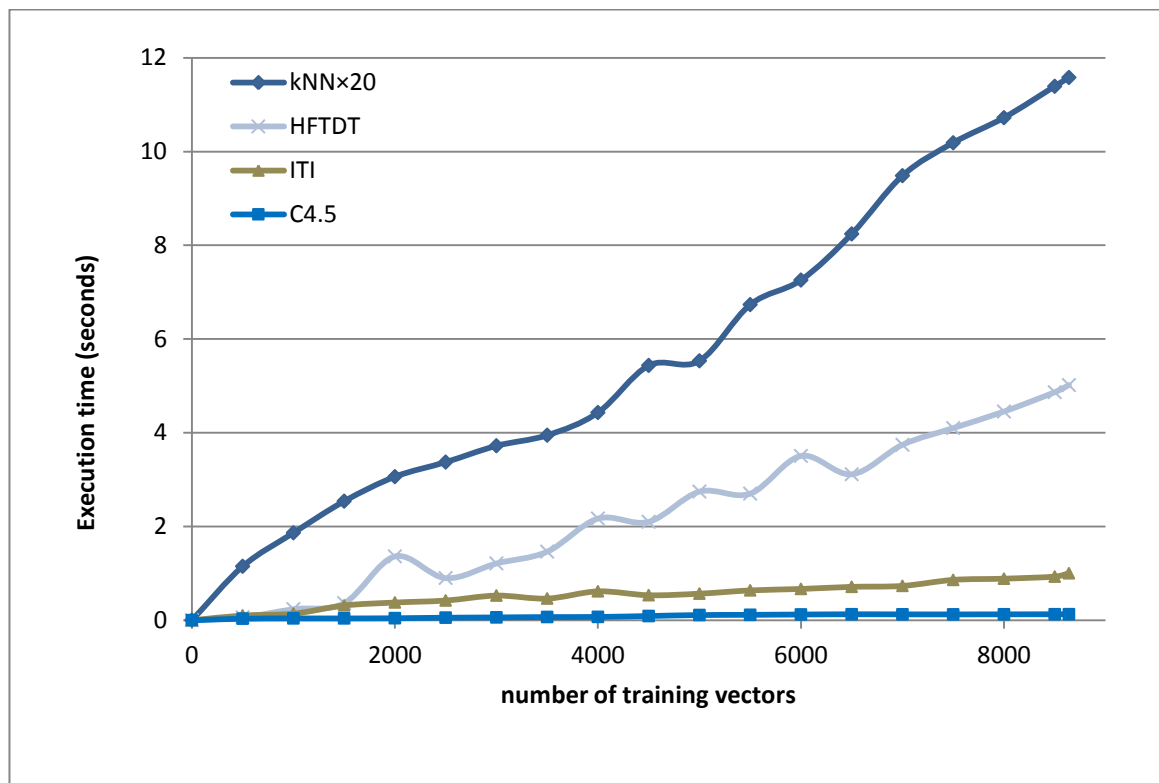


Figure 6.4: Execution time for the nursery dataset

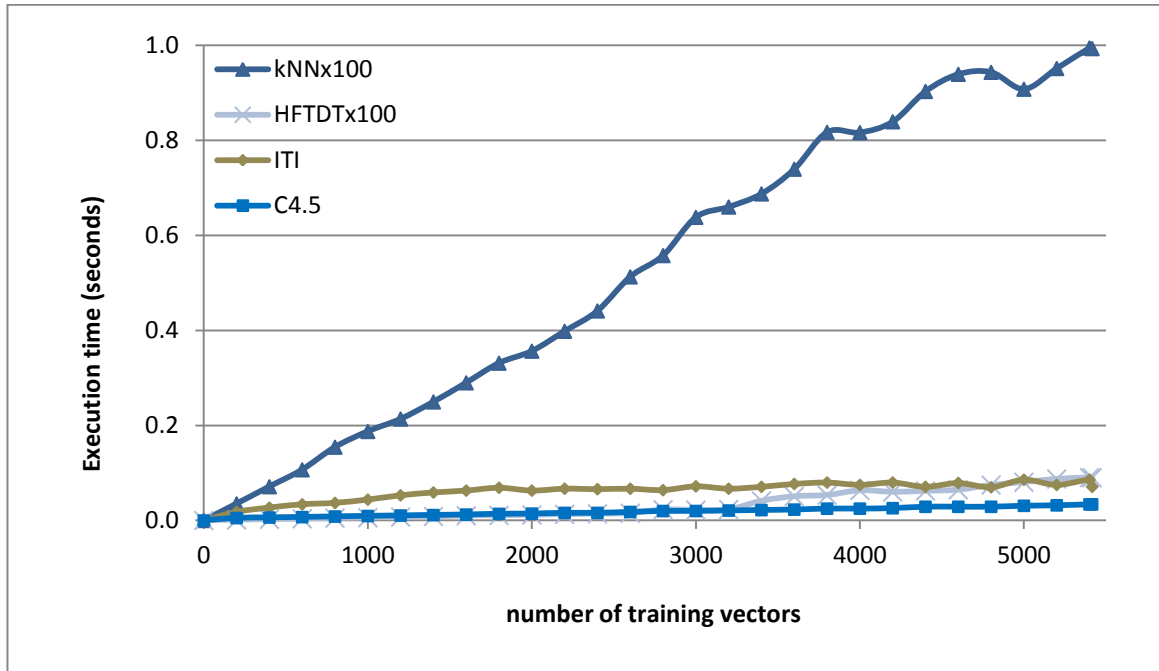


Figure 6.5: Execution time for the agaricus-lepiota dataset

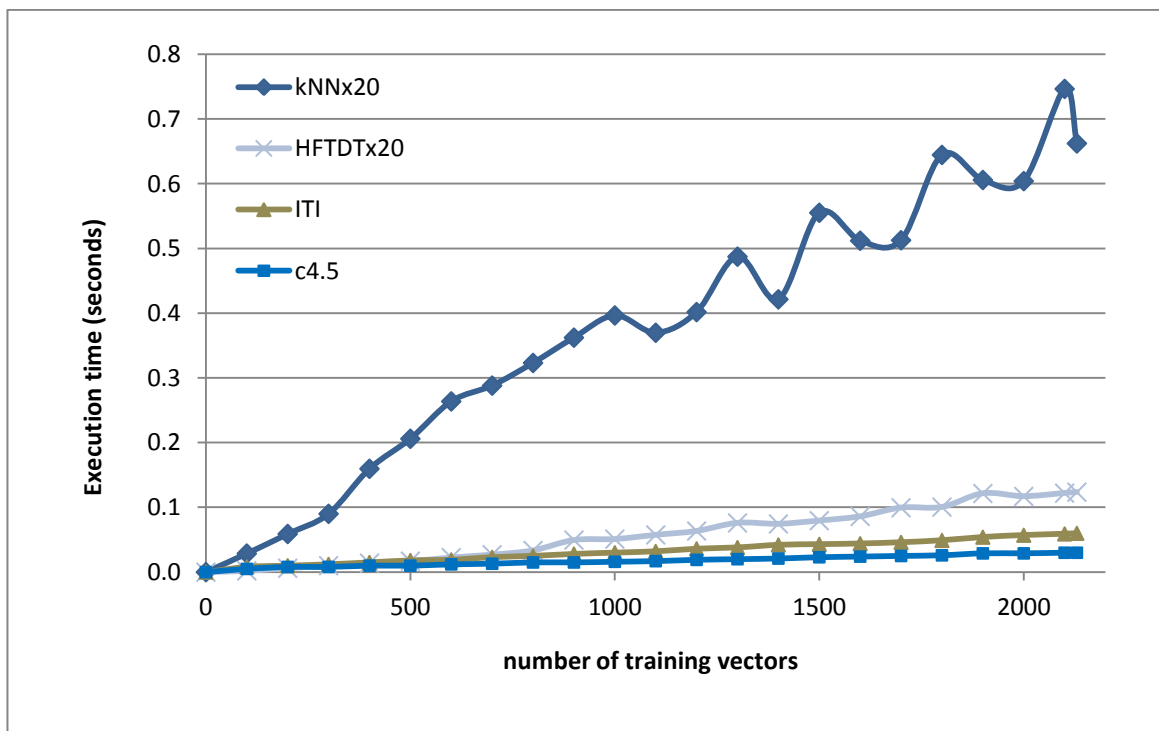


Figure 6.6: Execution time for the chess dataset

6.4.3 MEMORY USAGE

This experiment investigates the memory used by the classifiers. For embedded systems with limited memory, developers favour algorithm classifiers that require less memory in order to meet resource constraints. In real-world examples, the datasets often become larger with time, leading to an increased demand for memory resources. In particular, incremental methods tend to require more memory as they need to keep a record of the entire training set [87][88]. As an example, ITI, which operates incrementally, keeps records of parts of the training data at each node.

The measurements of memory obtained for ITI and C4.5 include that used to build and train the DT as well as the structures generated by the code to represent the DT. For HFTDT, measurements include that needed by the HFT as well the memory used to describe the DT.

The results shown in Figures 6.7 to 6.9 for the three test datasets demonstrate that the memory consumption for HFTDT is significantly less than that required by the other classification methods assessed. This is partly due to the simple structure of the hashed frequency table (see Section 5.2), but also the adoption of the fixed node structure described in Section 5.4. The memory usage results demonstrate the principal contribution of HFTDT to the current range of DT algorithms found in the literature, namely its ability to operate in environments that are restricted in terms of memory resource.

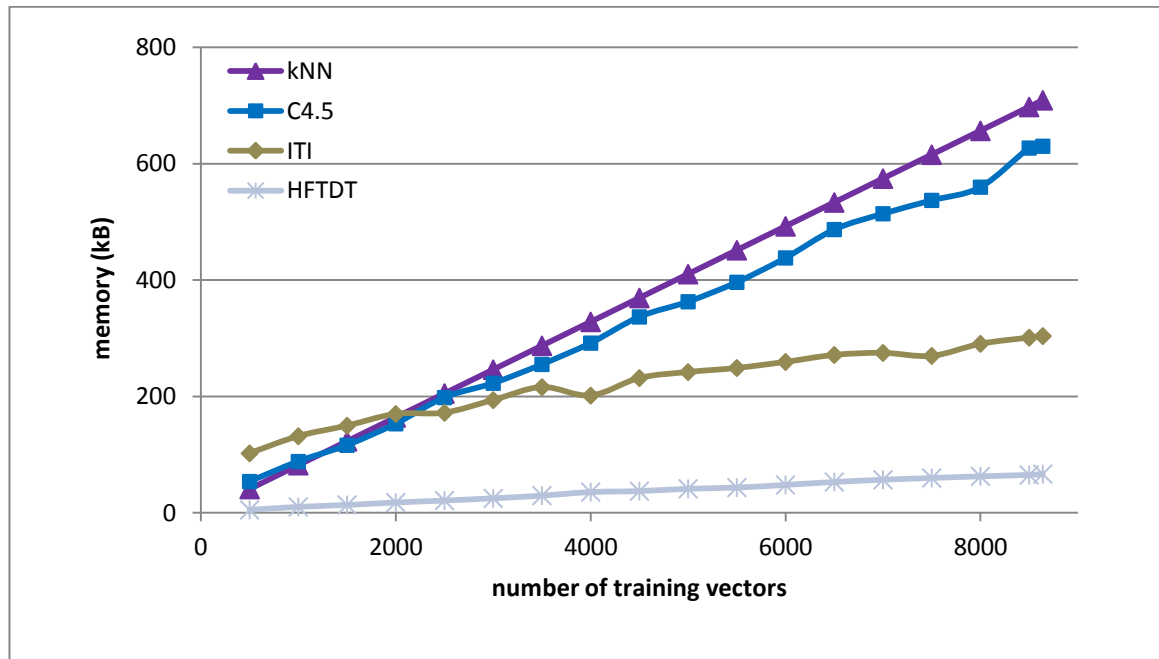


Figure 6.7: Memory usage for the nursery dataset

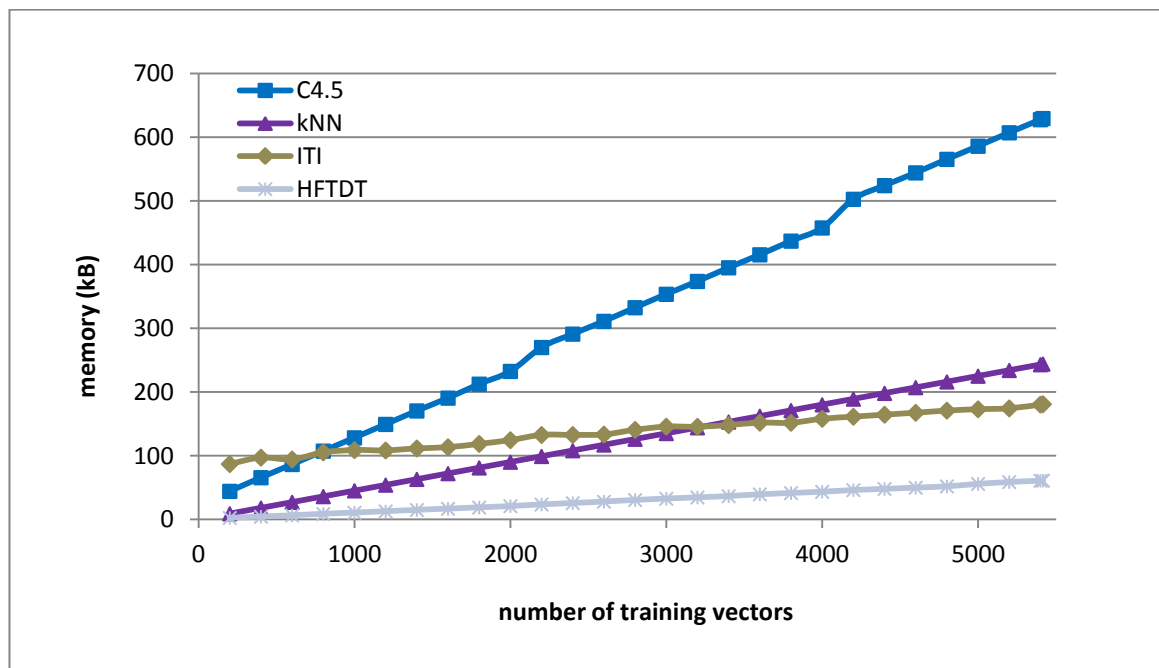


Figure 6.8: Memory usage for the agaricus-lepiota dataset

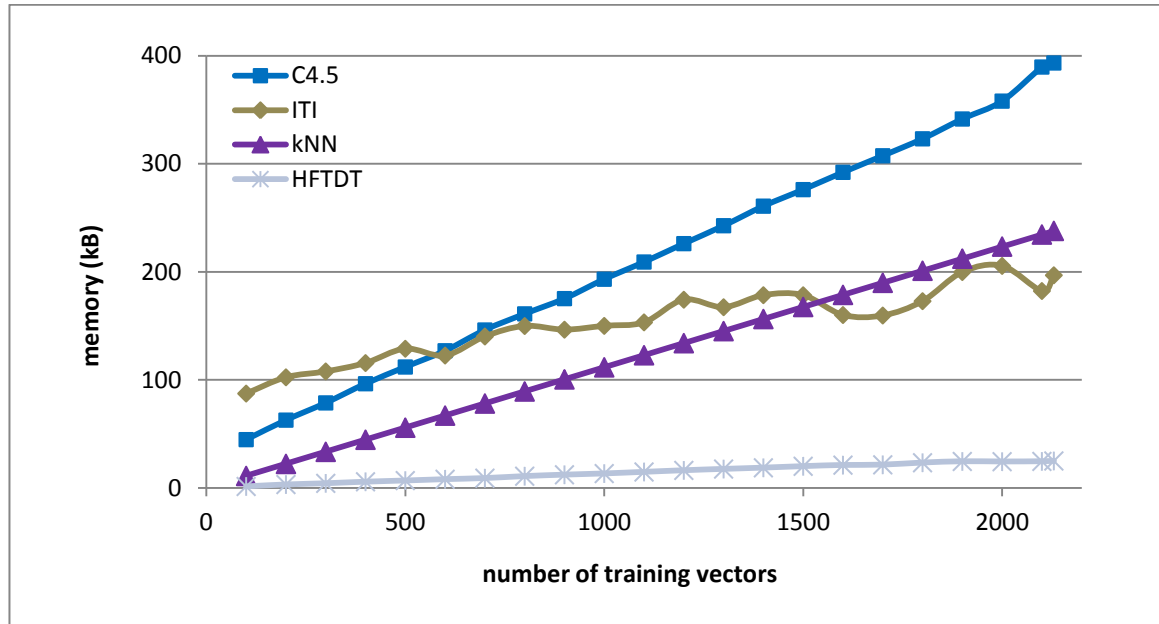


Figure 6.9: Memory usage for the chess dataset

6.4.4 CLASSIFICATION ACCURACY

Classification accuracy is the principal measurement used to evaluate the performance of learning algorithms [89]. In classification assessment, a training set is first used to build the DT and the test is performed on unseen data. The accuracy of a classifier on a given test set is the percentage of test set vectors that are correctly classified [78], as measured by the classification error expressed as

$$\text{classification error} = \left(\frac{\text{incorrectly classified vectors}}{\text{total number of vectors in the test set}} \right) \times 100\% . \quad (6.1)$$

The results for the nursery dataset shown in Figure 6.10 demonstrate that the HFTDT classification performance was closest to that of kNN for smaller numbers of training vectors, but reaching a constant steady-state performance as more training vectors are added and then performing better than the other classifiers.

The results for the chess dataset are shown in Figure 6.12 and HFTDT achieved better performance than kNN, but somewhat worse than ITI and C4.5. In comparison with the other DT methods investigated, HFTDT does not use pruning, and the difference in performance is likely to be due to its poorer generalization.

For the agaricus-lepiota dataset the performance of HFTDT is relatively poor when the number of training vectors is fewer than 3000, but its performance gets better with more vectors used in training and the accuracy is best among the classifiers between sample 3600 and 4000. kNN and C4.5 performed similarly, while ITI performed poorly between samples 2000 and 3000 due to the underfitting caused by pruning. As discussed in section 6.5.1, when there are 3000 or more training vectors in the agaricus-lepiota dataset, 15% to 30% of the vectors contain missing values, adversely affecting the classification accuracy. Acuna *et al.* [90] found that the missing value rates between 1-5% are manageable, 5-15% require sophisticated methods to handle, but rates above 15% may severely impact interpretation. The ITI approach to missing values is that whenever a vector with a missing value is needed for a test in a DT, the vector is simply saved at the node, without it being passed to other branches [91]. C4.5 uses a probabilistic approach to handle missing values by using a corrected gain ratio criteria [92], while the kNN uses a technique to handle one or more missing values, where the missing values are replaced with estimated ones based on information available in the dataset.[93].

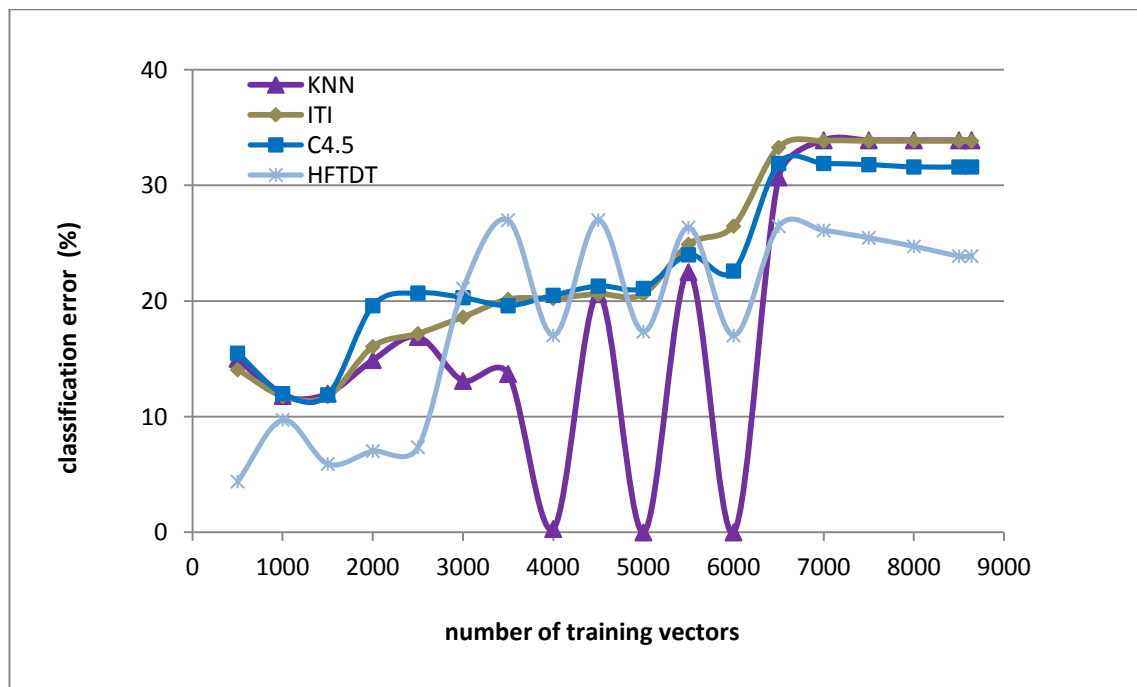


Figure 6.10: Classification error for the nursery dataset

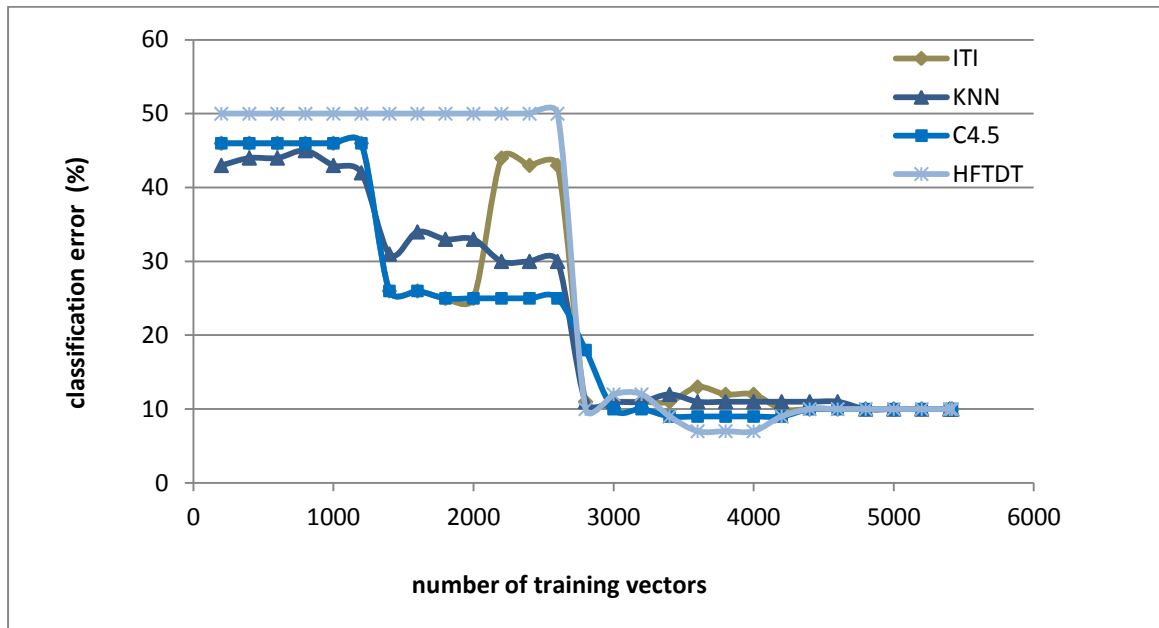


Figure 6.11: Classification error for the agaricus-lepiota dataset

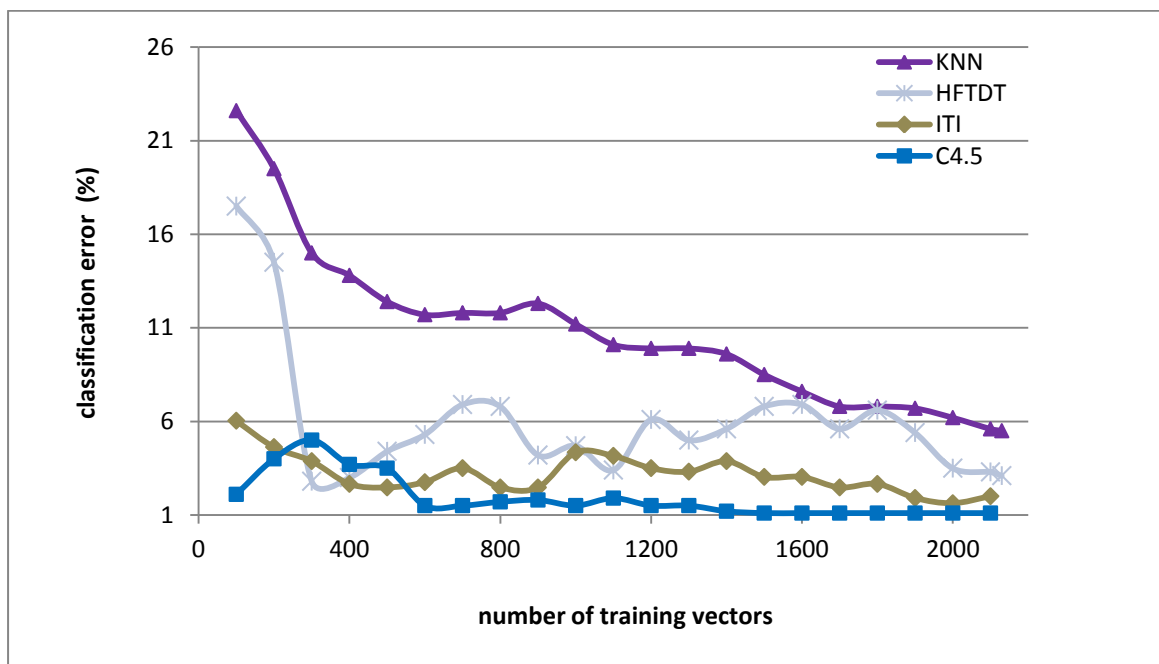


Figure 6.12: Classification error for the chess dataset

6.5 CONCLUSION

The chapter has presented four experiments with the aim of evaluating the performance of the HFTDT method, namely the number of nodes, computation time, memory usage and classification accuracy. The results show that HFTDT can successfully be used to generate DTs from a range of datasets and partially satisfies the requirements of embedded systems.

HFTDT demonstrated a substantial reduction in memory usage compared to existing DT methods. The memory usage of ITI, C4.5 and kNN increase with the number of vectors applied during training as they maintain records of the dataset and therefore are unsuitable for embedded system implementation.

The experimental results show that HFTDT satisfies robustness, even when compared to kNN, ITI and C4.5, all of which employ specific approaches to handle missing values. HFTDT treats any missing value as a new attribute value, requiring additional calculations compared to the other DT methods. Robustness can be improved by adopting a suitable technique to prepare the data before training, typically by performing data cleaning, data transformation or reduction [78]. However, all such methods require that all the data are available initially and so are not suitable for use in incremental classification applications. Scalability and calculation times can be improved by applying a pruning technique such as those adopted in C4.5 where error-based pruning is applied and in ITI which applies a pre-pruning technique. These approaches result in a substantial reduction in the number of nodes, yet are known to suffer from underfitting.

7 HARDWARE IMPLEMENTATION OF HFTDT

This chapter describes a hardware implementation of the most time-consuming function of the HFTDT implementation. The first section describes the profiling carried out to measure the times spent in the execution of each function of HFTDT, with the express purpose of determining those that are the most time consuming. The second section discusses the hardware approach adopted to improve the performance of the HFTDT method. The hardware solution uses a state-of-the-art high-level synthesis (HLS) tool to implement the most time-consuming function on a range of Xilinx FPGAs families.

7.1 INTRODUCTION

Due to its ability to execute operations in parallel, hardware implementations have the potential to provide substantial execution time performance advantages over sequential software implementations.

HFTDT is an incremental learning decision tree method designed to target embedded systems. The nature of DTs allows parallel realisation and the hardware implementation can be utilized to exploit two types of parallelism in the implementation of DTs. The first type is high-level thread-level parallelism that can be employed when expanding nodes. The second type is low-level instruction parallelism inside each node that can be exploited to accelerate the necessary calculations.

Not all DT algorithms are suitable for embedded systems handling real-time incremental problems, as the two main criteria introduced in Section 2.2.2 need to be met. The HFTDT source code was written in the C language to facilitate the move to hardware implementation using HLS tools. Parts of the code were re-written to allow easier implementation in hardware, as well as employing look-up tables to avoid the need to synthesise mathematical libraries.

7.2 HFTDT CODE PROFILING

This section concentrates on profiling the execution of the HFTDT code. The purpose of this activity is to identify those parts of the code whose hardware implementation would yield the most benefit in terms of execution time reduction. Profiling measures the time spent executing each function in software in order to determine the most time-consuming.

7.2.1 SOURCE CODE RECOMPILATION

To profile the source code of the HFTDT method, the GNU ‘gprof’ profiler [94] was used. gprof collects and arranges statistics of the code under analysis and provides information about the time spent in each function, the number of times it was called and which functions called other functions during the execution.

The HFTDT source code is written in the C language and contains 12 functions together containing approximately one thousand lines of code. The information presented by the gprof profiler includes all the features that have been used in the code, and it excludes any unused features from the profile information.

The results of the gprof are made available as a flat profile that shows the total time taken by each function operating in isolation, and a call graph that shows the time spent in each parent function and its children functions.

7.2.2 RESULT OBTAINED USING THE FLAT PROFILE

This section presents the gprof flat profile results. The HFTDT source code has been profiled and tested by the datasets introduced in Section 6.2, namely nursery, agaricus-lepiota and chess.

7.2.2.1 THE FLAT PROFILE

The flat profile generated by the gprof [94] includes a statistical summary table of the execution information of a program's functions. Figure 7.1 shows an example of the flat profile results for the HFTDT code and provides information of the percentage of the total running time taken by each function the (*% time*) column, the time taken by a function and those that call it (*cumulative seconds*), the running time of the function (*self seconds*), the number of times the function is called (*calls*), the average time spent in this function per call (*self ms/call*) and the average time in milliseconds spent in the function and its descendants per call (*total ms/call*).

```
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self   self   total
time  seconds  seconds  calls  s/call  s/call  name
91.72    20.97    20.97 59622960    0.00    0.00  rev_index_func
 7.88    22.77     1.80  17040    0.00    0.00  FindAttributeInstances
 0.39    22.86     0.09   3500    0.00    0.00  updateHashTable
 0.04    22.87     0.01   4320    0.00    0.00  function_test_data
 0.04    22.88     0.01   3500    0.00    0.00  index_func
 0.00    22.88     0.00  70425    0.00    0.00  read_word
 0.00    22.88     0.00   1012    0.00    0.02  AttributeEntropy
 0.00    22.88     0.00    374    0.00    0.06  ExpandNextNode
 0.00    22.88     0.00    374    0.00    0.01  NodeEntropy
 0.00    22.88     0.00     9    0.00    0.00  read_to_EOL
 0.00    22.88     0.00     1    0.00    22.77  function_struct
```

Figure 7.1: Example of information provided by a flat profile generated by gprof for the nursery dataset

Figure 7.1 was generated for the nursery dataset and it can be seen that together the first two functions in the list consume the vast majority of the execution time. The first of these, *rev_index_func* consumes around 90% of the total time of the code. This function is used to read the attribute values from the stored keys in the hash table (see section 5.3). The function *FindAttributeInstances* is a second function that could be considered, but as its execution time is of an order of magnitude less than that of *rev_index_func* it was not considered further in the current work. The function *rev_index_func* is a clear candidate for consideration for hardware acceleration.

The results of the flat profile obtained by the gprof for the samples of the nursery dataset can be found in Figure 7.2 and show that the *rev_index_func* was again the most time-consuming function using an average of around 90% of the total running time. The second most time-consuming function was the *FindAttributeInstances* which took on average less than one tenth of the time consumed by the *rev_index_func*.

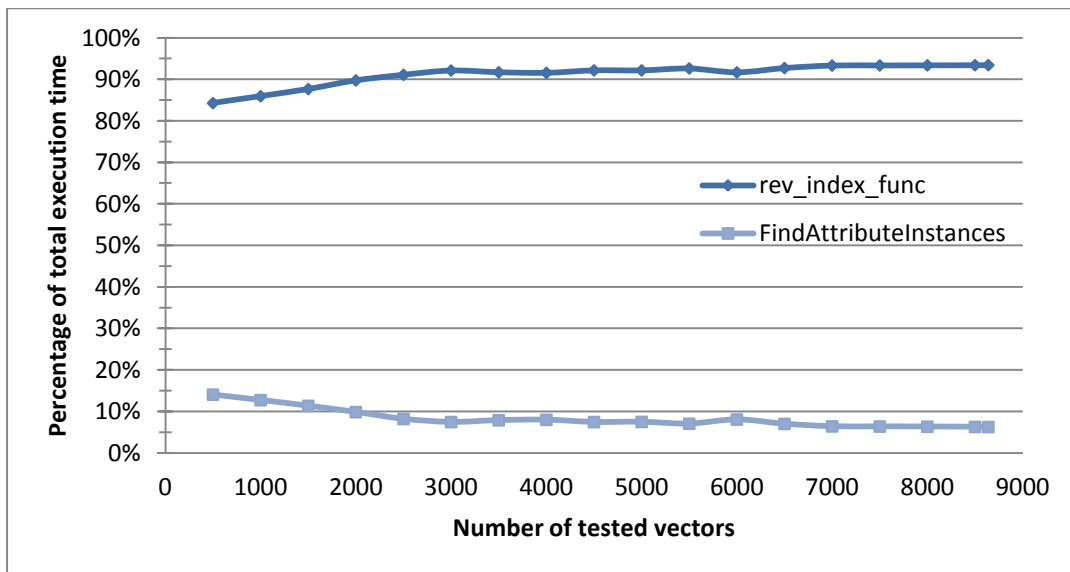


Figure 7.2: Flat profile results for the nursery dataset

The flat profile results for agaricus-lepiota dataset are shown in Figure 7.3 and confirm the influence of *rev_index_func* on the total running time, taking an average of around 88% of the total execution time. Again, the second most time-consuming function was *FindAttributeInstances*, taking an average of one eighth of the time consumed by *rev_index_func*.

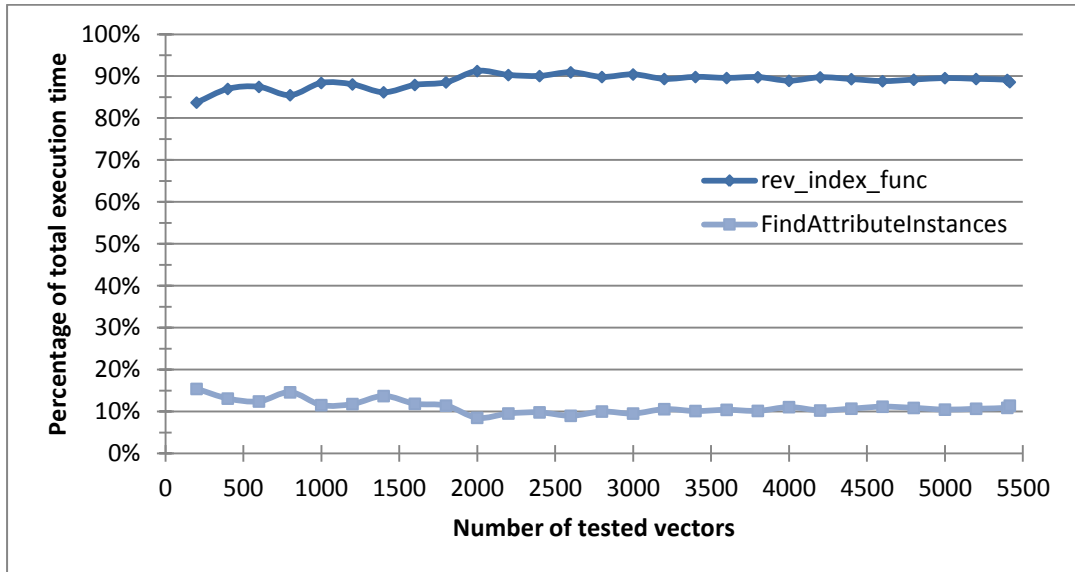


Figure 7.3: Flat profile results for the agaricus-lepiota dataset

The results obtained by the flat profiling of the chess dataset are shown in Figure 7.4 and again *rev_index_func* dominates as the most time consuming function this time consuming around 93% of the total execution time.

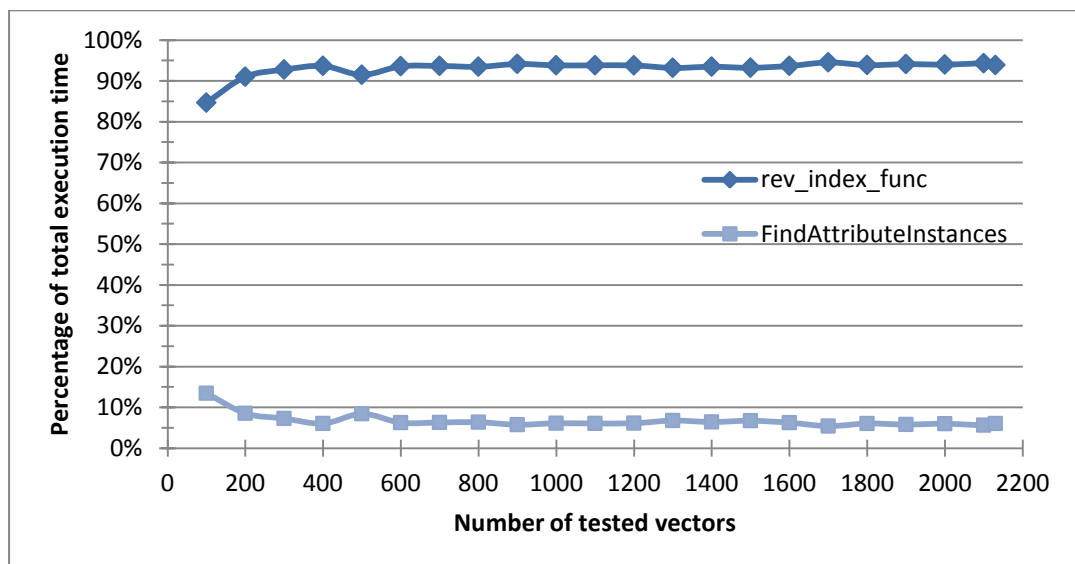


Figure 7.4: Flat profile results for the chess dataset

The results of the flat profiling for the three datasets show clearly that the *rev_index_func* is constantly the most time-consuming function. As the function consumes around 90% of the execution time, *rev_index_func* will be the sole function considered for hardware acceleration in the current work.

7.2.3 RESULTS OBTAINED USING THE CALL GRAPH

Call graphs were produced for the nursery, chess and agaricus-lepiota. Figure 7.5 shows an example of a call graph, which is part of the profiling results generated by *gprof* indicating the time spent in each parent function and its children functions [94].

The *gprof* call graph is sorted by the total time spent in the execution of each function and its children. The function listed in the rightmost column that is on the same row as in index shown in the leftmost column is the function identified as under consideration in that section. Within a section, the function under consideration is called by those listed above it and calls those listed below it. The second column is the percentage time consumed by the function under consideration and its children, while the third and the fourth columns are respectively the time spent in the function alone and the total time in the function and its children combined. The final column is the number of times the function has been called following the calling path shown compared with the total number of calls to that function by all routes. If function has no parents in the code being profiled, such as the function *main* in Figure 7.5, the word 'spontaneous' is used in place of a name.

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	22.88		main [1]
		0.00	22.77	1/1	function_struct [4]
		0.09	0.00	3500/3500	updateHashTable [8]
		0.01	0.00	4320/4320	function_test_data [9]
		0.01	0.00	3500/3500	index_func [10]
		0.00	0.00	70393/70425	read_word [11]
		0.00	0.00	9/9	read_to_EOL [12]
		0.20	2.30	1870/17040	NodeEntropy [7]
		1.60	18.67	15170/17040	AttributeEntropy [6]
[2]	99.5	1.80	20.97	17040	FindAttributeInstances [2]
		20.97	0.00	59622960/59622960	rev_index_func [5]
		0.00	22.77	374/374	function_struct [4]
[3]	99.5	0.00	22.77	374	ExpandNextNode [3]
		0.00	20.27	1012/1012	AttributeEntropy [6]
		0.00	2.50	374/374	NodeEntropy [7]
		0.00	22.77	1/1	main [1]
[4]	99.5	0.00	22.77	1	function_struct [4]
		0.00	22.77	374/374	ExpandNextNode [3]
		20.97	0.00	59622960/59622960	FindAttributeInstances [2]
[5]	91.6	20.97	0.00	59622960	rev_index_func [5]
		0.00	20.27	1012/1012	ExpandNextNode [3]
[6]	88.6	0.00	20.27	1012	AttributeEntropy [6]
		1.60	18.67	15170/17040	FindAttributeInstances [2]
		0.00	2.50	374/374	ExpandNextNode [3]
[7]	10.9	0.00	2.50	374	NodeEntropy [7]
		0.20	2.30	1870/17040	FindAttributeInstances [2]
		0.09	0.00	3500/3500	main [1]
[8]	0.4	0.09	0.00	3500	updateHashTable [8]
		0.01	0.00	4320/4320	main [1]
[9]	0.0	0.01	0.00	4320	function_test_data [9]
		0.01	0.00	3500/3500	main [1]
[10]	0.0	0.01	0.00	3500	index_func [10]
		0.00	0.00	32/70425	read_to_EOL [12]
		0.00	0.00	70393/70425	main [1]
[11]	0.0	0.00	0.00	70425	read_word [11]
		0.00	0.00	9/9	main [1]
[12]	0.0	0.00	0.00	9	read_to_EOL [12]
		0.00	0.00	32/70425	read_word [11]

Figure 7.5: Example of a call graph generated by gprof for the nursery dataset

The raw call graphs produced by gprof can be better represented for human consumption by a graphical format as shown in Figure 7.6. From Figure 7.6, it can be seen that the *main* function has six children and *function_struct* is the parent of the path that eventually leads to the most time-consuming function *rev_index_func*.

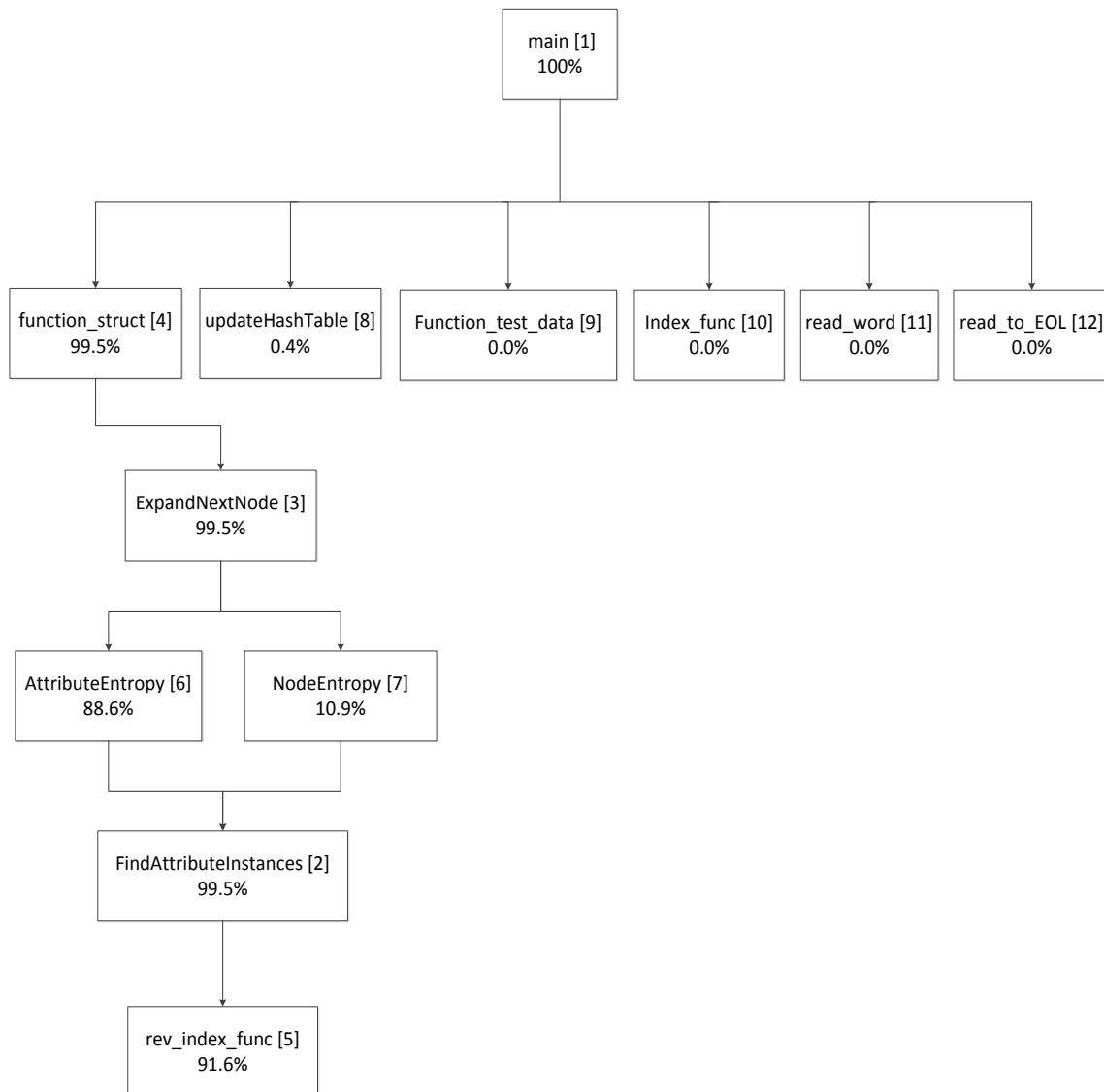


Figure 7.6: Call graph for HFTDT showing the call sequence and the execution times of the functions

The analysis of the call graph shows the sequence of functions calls and has emphasised the domination of *rev_index_func* in consuming the vast majority of the execution time. Appendix B includes the call graph analyses for the HFTDT code running all three datasets and in each case it was demonstrated that *rev_index_func* takes around 90% of the execution time.

7.3 HARDWARE DESIGN FOR THE MOST TIME-CONSUMING FUNCTION

The hardware approach on this research work focuses on improving the performance of the most time-consuming function of the HFTDT algorithm, namely *rev_index_func*

7.3.1 HIGH-LEVEL SYNTHESIS TOOLS

To provide a hardware solution, the approach taken in the current work is to adopt a high-level synthesis (HLS) tool. HLS tools perform automated conversion from a high-level language such as C, C++ or SystemC, to an electronic system level (ESL) description and such tools first became commercially available during the 1990s [95]. The HLS development is shown in Figure 7.7, where the HLS tool converts the algorithms written in a high-level programming language into a hardware description language (HDL) such as Verilog or VHDL. The HDL can then be synthesised yielding a register transfer level (RTL) design that can target hardware platforms such as ASICs or FPGAs.

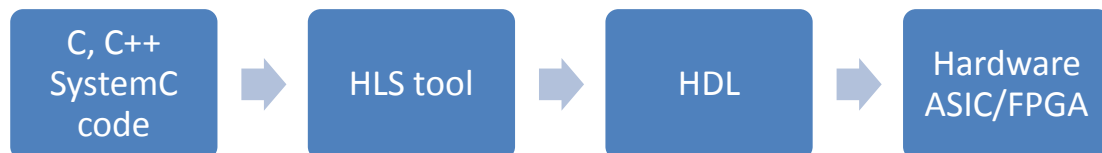


Figure 7.7: Process of high-level language translation to HDL using HLS tools

The steps in the process of generating HDL using an HLS tool can be summarized as follows [96], [97].

- (1) Compilation of functional specification, in which the translation of the source code to an internal representation is carried out. This step involves several code optimization such as dead-code and false data dependency elimination as well as loop transformations.

- (2) Allocation defines the type and number of hardware resources needed to satisfy the design constraints, such as instance, functional and storage units, as well as connectivity components.
- (3) Scheduling, in which each operation required in the specification model is constrained to fit into a clock cycle.
- (4) Binding of variables to a storage unit and operations to a functional unit capable of its execution.
- (5) Output processing, in which the RTL source code is written in the target language.

7.3.2 CHOOSING AN HLS TOOL

There are several HLS tools available for hardware designers, such as Vivado HLS [98], Catapult C [99], C-to-Silicon [100], Compaan [101], CyberWorkBench [102] and Symphony C [103]. Meeus *et al.* [104] evaluated several HLS tools, including the aforementioned, and recommended the use of the AutoPilot Xilinx tool (now Vivado HLS). The evaluation criteria included the source language used, ease of implementation, tool complexity, user interface and documentation, support for data types, design exploration capabilities and correctness of the generated design. BDTI [105] found that the advantage of using AutoPilot is that the quality of the design results produced were equivalent to hand-written RTL code. The Academic Department in which the author is currently studying has commercial licences for both Vivado and Calypto, so both were originally considered. The HLS tool selected for the current research work is version v_2013.4 of the Vivado HLS tool that was released in December 2013 [106], and was chosen for its ease of use, fast synthesis and documentation availability.

7.3.3 HARDWARE DESIGN USING THE HLS TOOL

Figure 7.8 shows the HLS development process adopted. The first step is to verify the HFTDT C code by means of a test bench that applies example data used in the software only version described in Chapter 6. The second step is to verify the RTL design by confirming its functionality matches that of the C algorithm, simply by re-use of a test bench used for C verification.

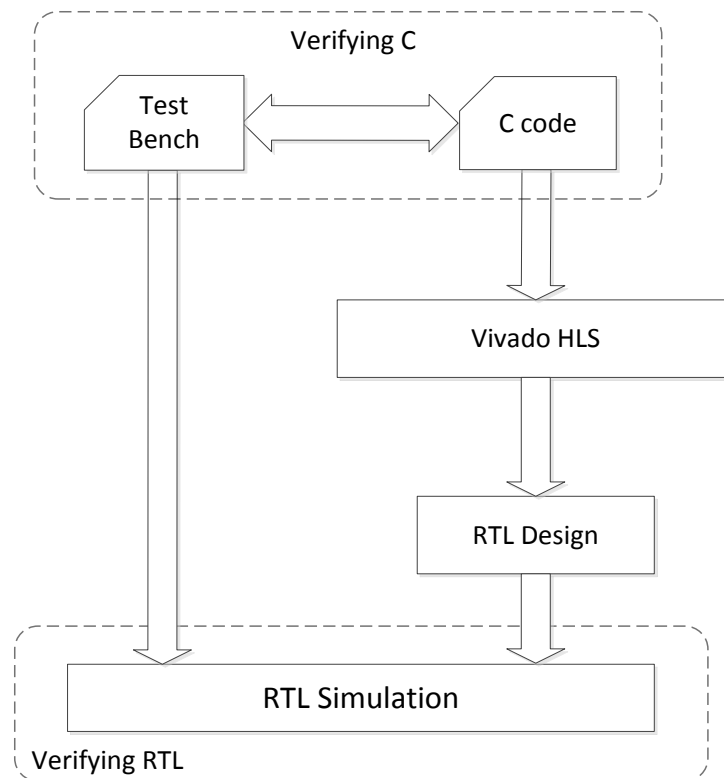


Figure 7.8 Overview of the Vivado high-level synthesis design process

7.3.4 DESIGN SYNTHESIS AND CO-SIMULATION

The part of the HFTDT source code that consumes most of the execution time is targeted in the hardware design. As shown in Figure 7.9, the C code used at the top level represents the reverse index function, *rev_index_func*. The test bench is the C *main* function as shown in Figure 7.8 and it is not synthesisable. The *main* function

includes an array that holds the dataset used in the verification process, this being the numerical form of one of the nursery, chess and agaricus-lepiota examples.

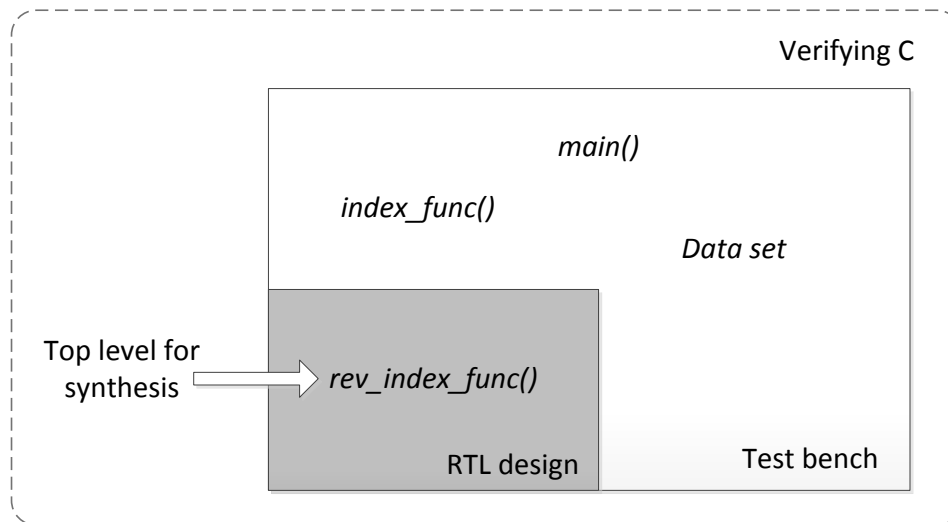


Figure 7.9 Concept of C Verification

The C verification process that was carried out is now described. In the test bench, the *main* function passes dataset vectors one at a time to the *index_func*, which generates a unique 64 bit key for each vector. The key is then passed to the *rev_index_func*, which reads the key and generates an output vector. To verify the design, the outputs of the *rev_index_func* are compared to the original dataset vectors; the test being passed only if the two sets of vectors match exactly.

In Vivado HLS, RTL verification is achieved using co-simulation (see Figure 7.8). The RTL co-simulation uses the C test bench to generate input stimuli for execution on the RTL design. As in the C verification process, the RTL design verification compares the input vectors with the outputs of the design. The final step of the design is to export the RTL design as a block of intellectual property.

For the software implementation, the actual execution time of the reverse index function is found by knowing the time taken by a single call to *rev_index_func* and how many times it is called by using the results from the flat profile table. For the results obtained in hardware, the time taken to process each vector of the samples

is calculated and then multiplied by the number of times the reverse index function is called.

7.3.5 TARGETED FPGAS

Vivado HLS supports the Xilinx 7-series [107] and the Xilinx Zynq SoC [108] FPGAs. The FPGAs listed in Table 7.1 are the four families supported by Vivado HLS and that were considered in the current hardware design.

Table 7.1: Targeted FPGAs for the hardware design

Family	FPGA
Virtex7	XC7VX980T
Kintex7	XC7K70T
Artix7	XC7A75T
Zynq	XC7Z100T

The Zynq SoC family contains an ARM Cortex-A9 microprocessor and an FPGA fabric, the latter being supported by the Vivado HLS tools. The Xilinx product specification data sheet for the 7-series FPGA families [107] indicates that the high-end Virtex7 devices are designed for applications demanding high performance, Kintex7 are mid-range devices and the Artix7 devices are designed for applications where low power consumption is important. Figure 7.10 shows a comparison of the 7-series devices in terms of their relative power consumption and performance.

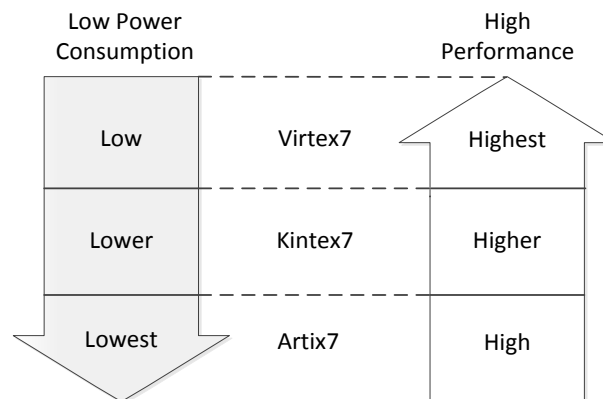


Figure 7.10: Xilinx 7-series FPGAs families [109]

In this work, the intention is to select the FPGA family device that exhibits the shortest calculation time for the hardware implementation of the *rev_index_func* function, in order to reduce the time taken to execute the system code overall. The evaluation was based on the maximum clock frequency that could be achieved by each FPGA family following synthesis.

7.3.6 DATASET USED IN THE HARDWARE DESIGN

The C test benches use the numerical versions of the datasets, as introduced in Section 3.2.2. As each dataset has different numbers of attributes and classes, the generated top-level function *rev_index_func* is different for each dataset, with differences being also being apparent in the two arrays named *acc_num_att_values* and *num_att_values* (as discussed in Chapter 5).

7.4 SIMULATION RESULTS

Each of the FPGAs listed in Table 7.1 were targeted and three hardware designs were produced, one for each of the datasets nursery, agaricus-lepiota and chess.

Figure 7.11 is an example of the reverse index function for the agaricus-lepiota dataset that has been accelerated in hardware. Referring to Section 5.3.5 to produce the output vector the process involves right shift and bitwise AND operations. The 'key' is obtained from the

hashed frequency table and the generated output vector is saved in the P_array. The size of P_array varies according to the characteristics of the dataset. In this example the P_array has 23 elements including 22 attributes and class.

```
u64 rev_index_func(u64 key, u8 P_array[23])
{
    P_array[0] = (key & (u64) 1);
    P_array[1] = (key >> (u64) 1) & ((u64) 1);
    P_array[2] = (key >> (u64) 2) & ((u64) 1);
    P_array[3] = (key >> (u64) 3) & ((u64) 1);
    P_array[4] = (key >> (u64) 4) & ((u64) 3);
    P_array[5] = (key >> (u64) 6) & ((u64) 3);
    P_array[6] = (key >> (u64) 8) & ((u64) 3);
    P_array[7] = (key >> (u64) 10) & ((u64) 3);
    P_array[8] = (key >> (u64) 12) & ((u64) 3);
    P_array[9] = (key >> (u64) 14) & ((u64) 3);
    P_array[10] = (key >> (u64) 16) & ((u64) 3);
    P_array[11] = (key >> (u64) 18) & ((u64) 7);
    P_array[12] = (key >> (u64) 21) & ((u64) 7);
    P_array[13] = (key >> (u64) 24) & ((u64) 7);
    P_array[14] = (key >> (u64) 27) & ((u64) 7);
    P_array[15] = (key >> (u64) 30) & ((u64) 7);
    P_array[16] = (key >> (u64) 33) & ((u64) 15);
    P_array[17] = (key >> (u64) 37) & ((u64) 15);
    P_array[18] = (key >> (u64) 41) & ((u64) 15);
    P_array[19] = (key >> (u64) 45) & ((u64) 15);
    P_array[20] = (key >> (u64) 49) & ((u64) 15);
    P_array[21] = (key >> (u64) 53) & ((u64) 15);
    P_array[22] = (key >> (u64) 57) & ((u64) 1);
}
```

Figure 7.11: Code for the accelerated function for the agaricus-lepiota dataset

7.4.1 NURSERY DATASET

The nursery dataset has eight attributes and five classes, and is the simplest of the three datasets. Figure 7.12 shows the execution time results obtained from the hardware solutions for a range of FPGAs, as well as times obtained from the software implementation. The actual values obtained can be found in Appendix C Table C.1.

The simulation results showed that, compared with the software implementation, the execution time for the hardware solution was around three times shorter for the Artix7 and around 4.2 times shorter for the Kintex7. Also, the maximum clock frequency achievable was the lowest at 521 MHz for the Artix7 and greatest at 743 MHz for the Kintex7.

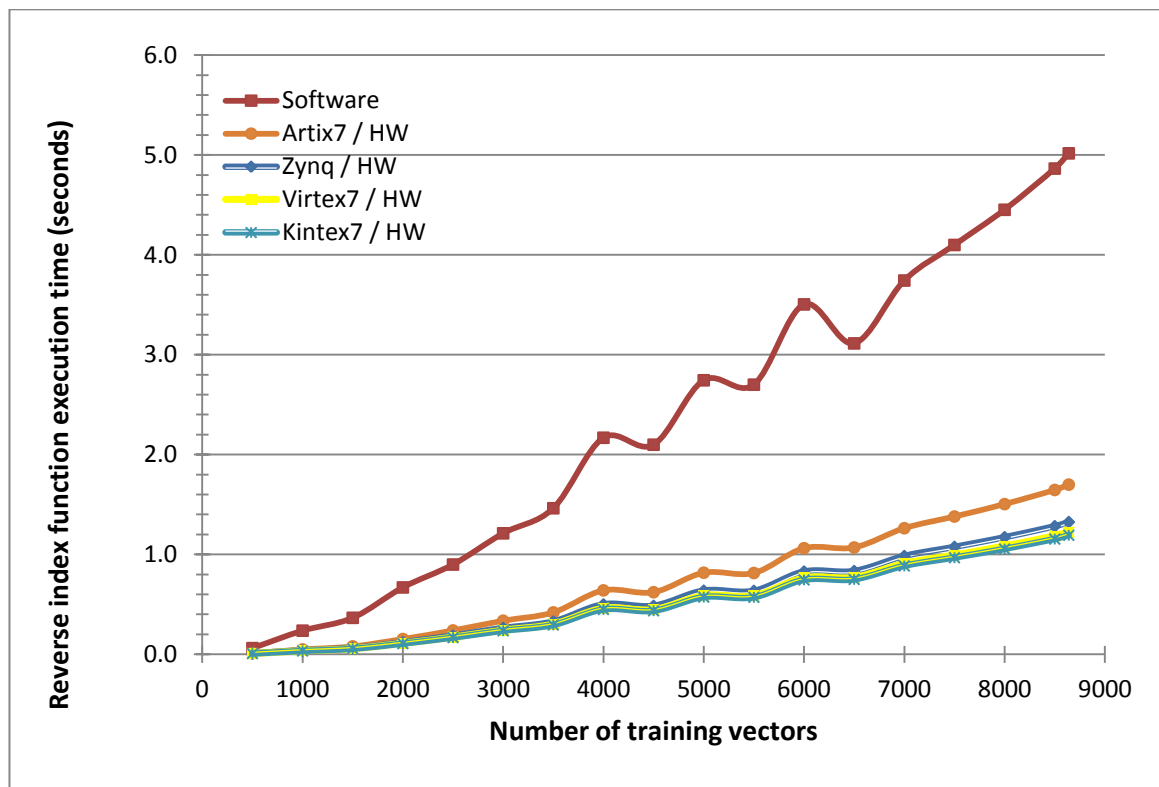


Figure 7.12: Hardware acceleration results of targeting a range of FPGAs for the nursery dataset

Table 7.2 includes the hardware report of the number of slices, where each slice is a group of four look up tables (LUTs) and eight flip-flops (FFs). The Kintex7 achieved a higher clock frequency when compared to the Virtex7 but the number of slices used in the Virtex7 was fewer than that required on the Kintex7. The variation in the number of slices used in the solutions for all the FPGA families is probably due to inconsistencies in the place and route process of Vivado HLS [110].

Table 7.2: Hardware design report for the nursery dataset

FPGA FAMILY	SLICES	LUTs	FFs	Mean execution time reduction (%)	Clock frequency achieved
Virtex7	8	25	19	75.6	724 MHz
Kintex7	9	25	19	76.2	743 MHz
Zynq	9	25	19	73.6	669 MHz
Artix7	9	25	19	66.1	521 MHz

7.4.2 RESULTS FOR THE AGARICUS-LEPIOTA DATASET

The dataset of the agaricus-lepiota problem has 22 attributes and two classes. Figure 7.13 shows the execution times of the hardware and software implementations of the *rev_index_func* for the agaricus-lepiota dataset. Appendix C Table C.2 shows the actual figures obtained.

The results of simulation given in Figure 7.13 show that hardware acceleration reduces the execution time by between 6.5 times for the Artix7 and 9.6 times for the Virtex7.

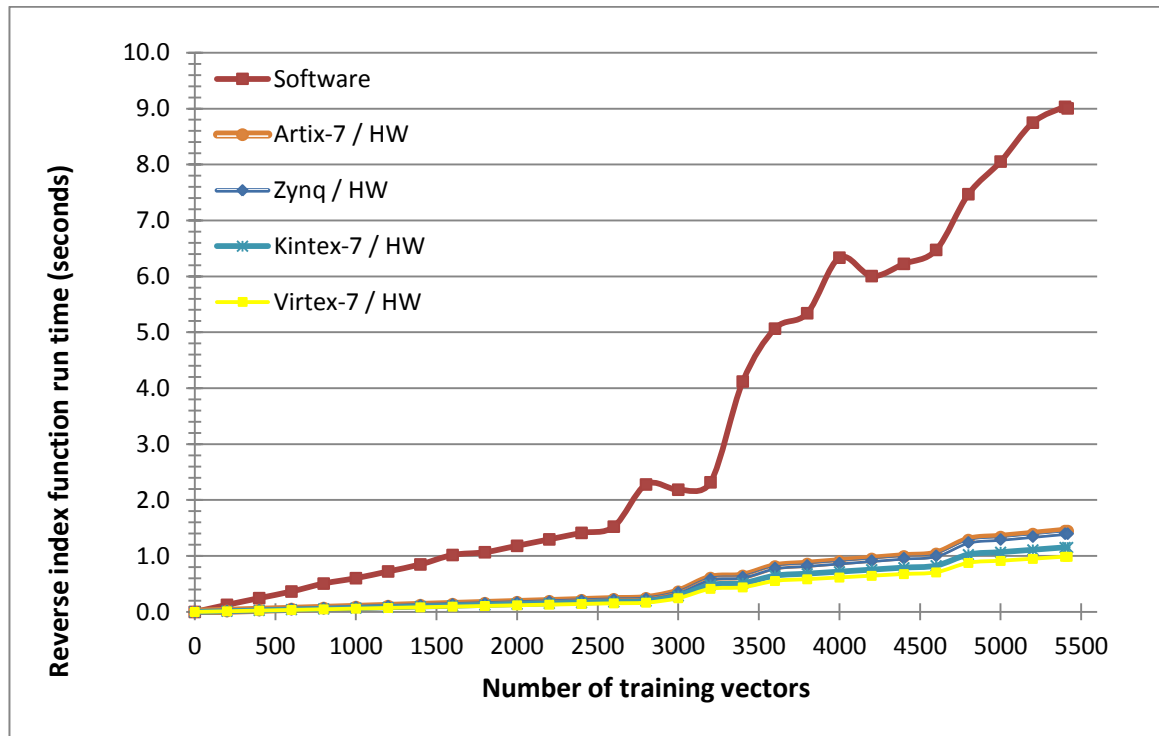


Figure 7.13: Hardware acceleration results of targeting a range of FPGAs for the agaricus-lepiota dataset.

Table 7.3 shows the hardware report of the number of slices, LUTs and FFs. The Kintex7 and Virtex7 achieved a similar clock frequency for the design, but both values were considerably higher than that of the Artix7. There was a small difference in the number of slices and LUTs used by the different FPGAs.

Table 7.3: Hardware design report using agaricus-lepiota dataset

FPGA FAMILY	SLICES	LUTs	FFs	Mean execution time reduction (%)	Clock frequency achieved
Virtex7	21	70	31	89.5	484 MHz
Kintex7	21	71	31	87.7	413 MHz
Zynq	21	69	31	85.1	341 MHz
Artix7	20	69	31	84.6	330 MHz

7.4.3 RESULTS FOR THE CHESS DATASET

The dataset of the chess problem has 36 attributes and two classes. Figure 7.14 shows the execution times of the hardware and software implementations of the *rev_index_func* for the chess dataset. Appendix C Table C.3 shows the numerical figures obtained for the simulation results.

The results of simulation show that the execution time achieved was almost two times shorter for the Artix7 implementation and around 3.4 times shorter for the Kintex7 solution. The maximum clock frequencies achieved are shown in Table 7.4 and ranged from 386 MHz for the Kintex7 to 217 MHz for the Artix7.

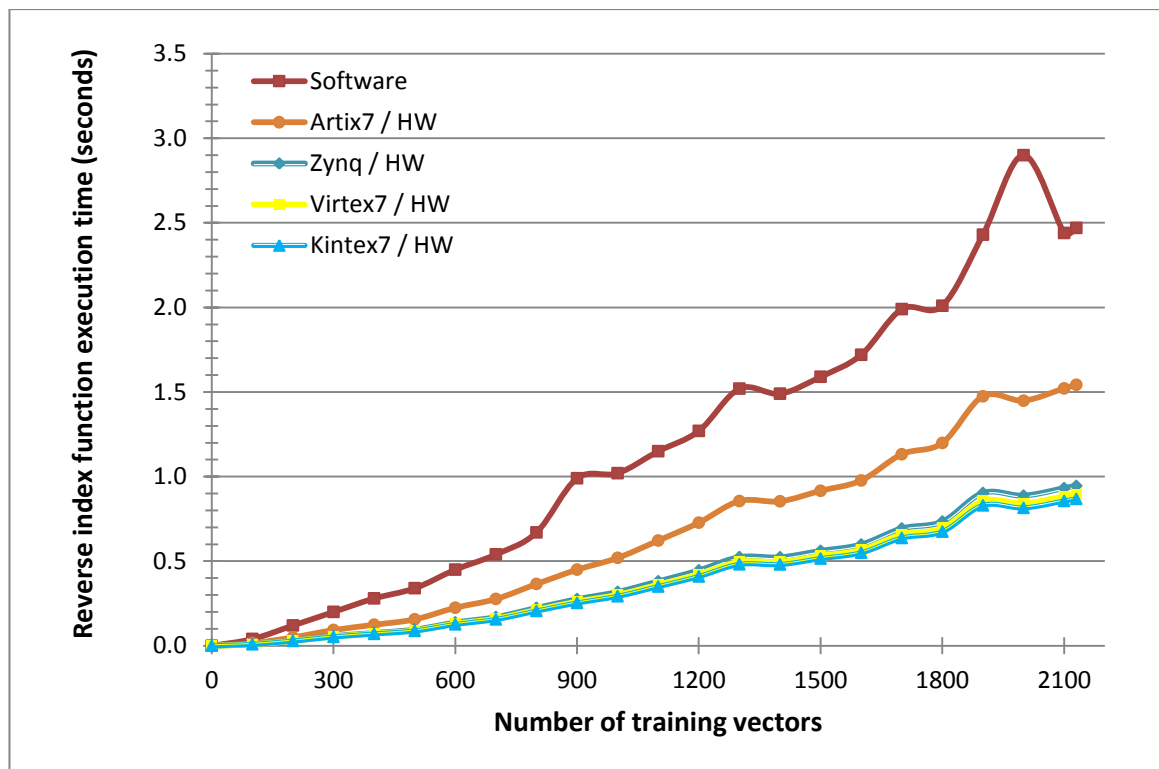


Figure 7.14: Hardware acceleration results of targeting a range of FPGAs for the chess dataset.

Table 7.4: Hardware design report using chess dataset

FPGA FAMILY	SLICES	LUTs	FFs	Mean execution time reduction (%)	Clock frequency achieved
Virtex7	39	136	54	69.4	374 MHz
Kintex7	42	136	54	70.3	386 MHz
Zynq	39	135	54	67.7	355 MHz
Artix7	39	134	54	47.3	217 MHz

7.4.4 RESULTS SUMMARY

Table 7.5 shows a summary for the implementation of the reverse index function showing the clock frequency and hardware acceleration achieved for the different FPGAs and for each of the datasets.

Table 7.5: Hardware simulation results summary

Dataset	FPGA Family	Frequency achieved (MHz)	Mean runtime acceleration	Latency	Initiation interval
Nursery	Virtex7	724	4.1x	4	5
	Kintex7	743	4.2x	4	5
	Artix7	521	3.0x	4	5
	Zynq	669	3.8x	4	5
Agaricus-lepiota	Virtex7	484	9.6x	11	12
	Kintex7	413	8.2x	11	12
	Artix7	330	6.5x	11	12
	Zynq	341	6.7x	11	12
Chess	Virtex7	374	3.3x	18	19
	Kintex7	386	3.4x	18	19
	Artix7	217	1.9x	18	19
	Zynq	355	3.1x	18	19

It can be seen from Table 7.5 that the FPGA implementations of the *rev_index_func* significantly improved the execution time performance when implemented for the test examples that exhibited a range of different numbers of attributes and attribute values.

The latency presented in Table 7.5 is the number of cycles needed to produce the output and the initiation interval is the number of clock cycles before new input can be applied. For all of the implementations, the latency reported by Vivado HLS tools is one cycle less than the initiation interval. Where the latency and initiation interval are so similar, it is clear that the design generated automatically by Vivado HLS has used no pipelining. The more the number of attributes required in the design, the greater the length of the array used to hold the output vector. Since Vivado HLS is known to map arrays to memory devices that have limited access capabilities, a solution results that has more hardware interdependencies and hence longer latency as the array length increases [98]. The Vivado HLS tools recommend considering array optimization techniques to allow more reads and writes in the same clock cycle.

Figures 7.15 to 7.17 show a comparison between the best hardware acceleration result with C4.5 and ITI. The result of Figure 7.15 for nursery dataset shows that hardware solution is 4.2 times faster than software, achieved similar performance to ITI and 6 times slower than C4.5. In Figure 7.16 the results for the agaricus-lepiota dataset show that the hardware solution reduced the execution time taken by the software by 9.6 times, where on average it is slower by 5.3 times compared to ITI and 15.6 times compared to C4.5. The results in Figure 7.17 for the chess dataset show that the hardware solution improved the software execution time by 3.4 times, which in hardware is slower by nine times and 18 times compared to ITI and C4.5 respectively.

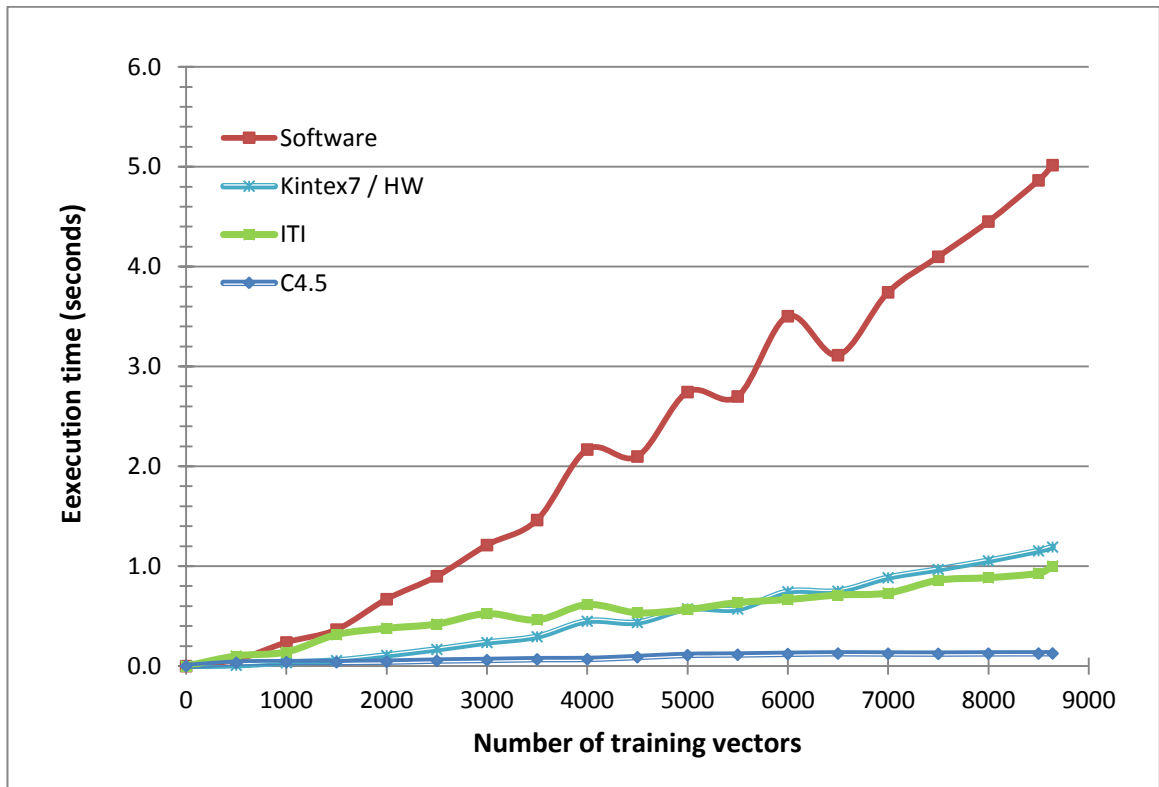


Figure 7.15: Hardware acceleration result compared with C4.5 and ITI for the nursery dataset

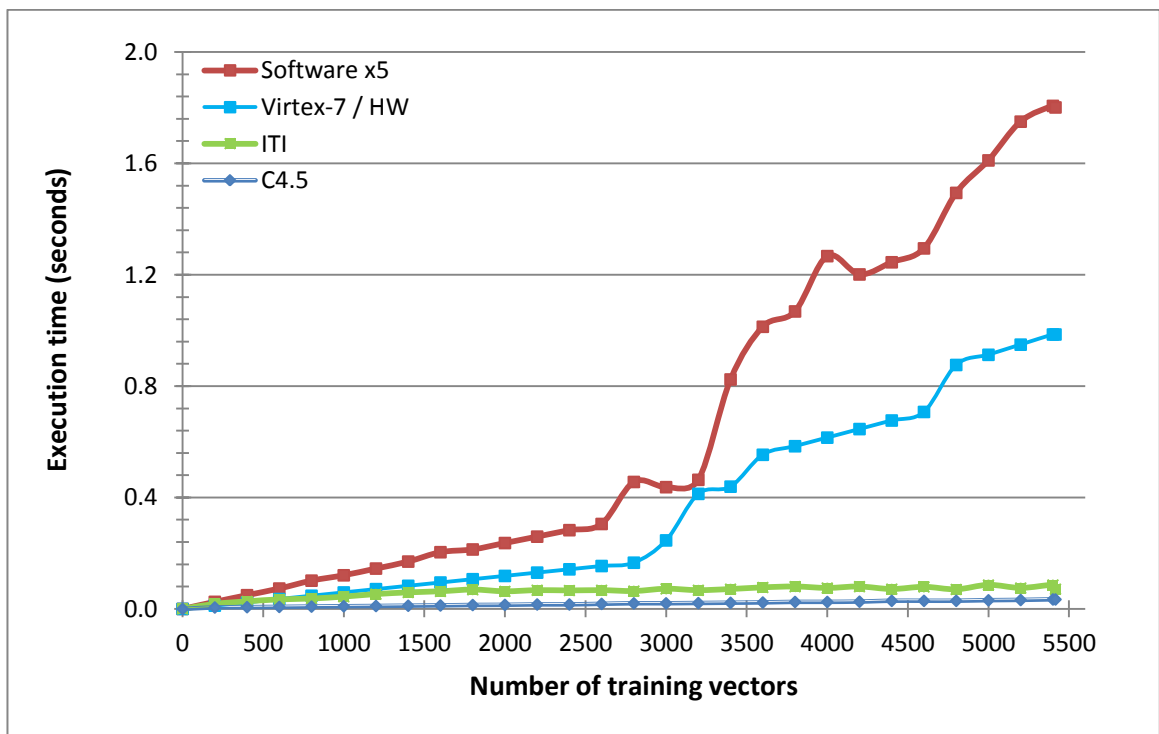


Figure 7.16: Hardware acceleration result compared with C4.5 and ITI for the agaricus-lepiota dataset

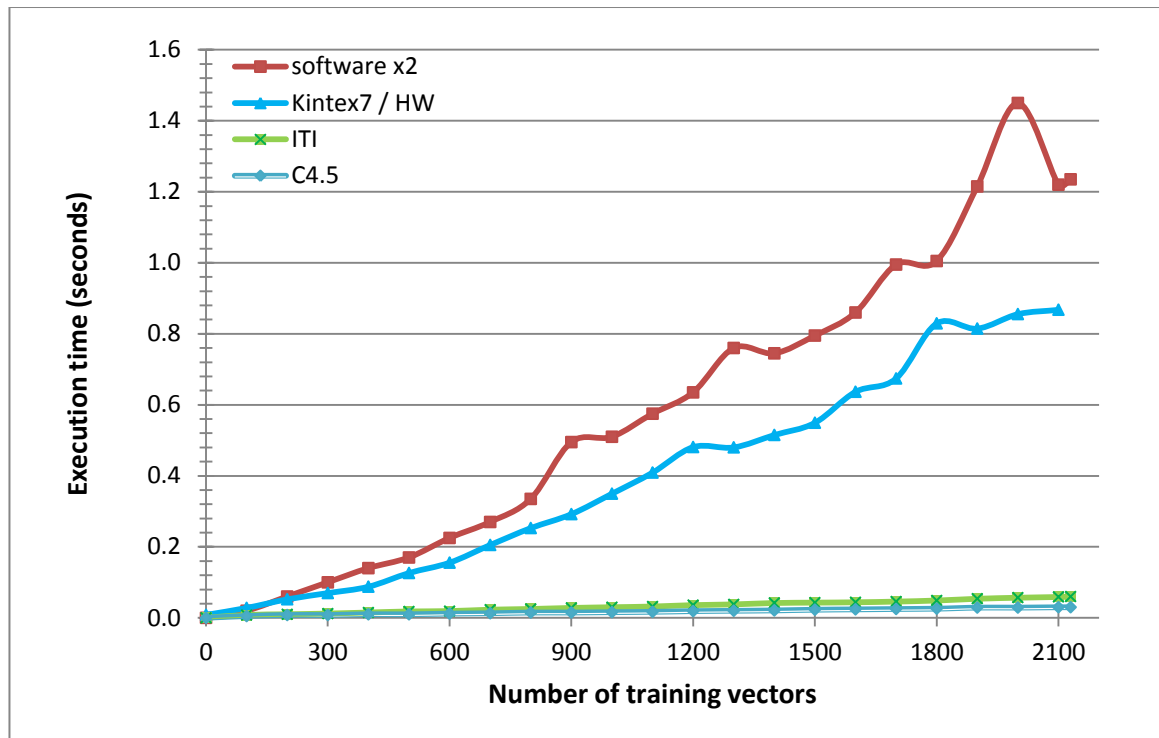


Figure 7.17: Hardware acceleration result compared with C4.5 and ITI for the chess dataset

7.5 EXTENSION OF THE PARALLEL IMPLEMENTATION

In this thesis, the hardware realisation has solely related to the supply of data for individual entropy calculations. The implementation of HFT has provided a novel method of storing and retrieving data vectors and only this part of the HFTDT approach has been accelerated; these operations being unique to this DT method.

The HFTDT algorithm implements incremental learning by storing the data in a HFT. The table is updated regularly with new data and each unique input data vector is stored with its own key. The way the data are stored is particularly suited to hardware implementation due to the manner in which the data are extracted. Since the data extraction is performed by cycling through the data stored in the HFT, the current entropy calculation that is being considered needs to wait for a specific data vector to become available before it can be used as part of the computation.

No attempt has been made to implement the DT node and entropy calculations in hardware, mainly because approaches already exist in the literature and so have little novelty value. In particular, where there is a set of open nodes to be investigated when building a DT, all of the nodes require a number of entropy calculations to be computed so that their children can be determined. However, as the method of data supply to HFTDT is quite different from that used in other DT approaches, it is interesting to consider the architecture that would be appropriate for parallel hardware implementations.

The hardware solution proposed here exploits two types of parallelism in the implementation of DTs. For the first type, high-level thread-level parallelism can be extracted when two or more nodes need to be expanded concurrently. For the second type, low-level instruction level parallelism inside each node of the DT can accelerate the sets of entropy calculations that need to be performed at each node.

The hardware approach proposes a parallel solution by using the information provided by the *rev_index_func* to feed multiple nodes (open nodes) simultaneously in order to reduce the overall time taken to perform entropy calculations. Knowing that the data goes only to the nodes that need to be investigated, Figure 7.18 shows how a set of n nodes could be considered in parallel, with all data fed simultaneously from the *rev_index_func*.

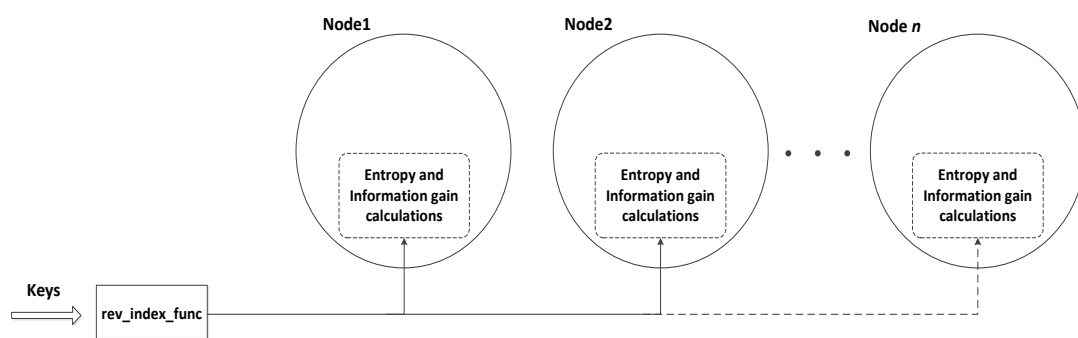


Figure 7.18: Implementation of *rev_index_func* to supply data to the open nodes in a DT

7.6 SUMMARY

HFTDT in its implementation of the nursery, chess and agaricus-lepiota datasets has been investigated using the GNU gprof profiler. gprof provides two types of profiling, a flat profile and a call graph. The results of the profiling showed that the reverse index function *rev_index_func* consumes most of the total execution time.

To improve the acceleration performance of the HFTDT code, a hardware design for *rev_index_func* was implemented. The hardware design was realised using Vivado HLS, the tool providing simulation, synthesis and verification of the *rev_index_func*. A C test bench file was created for verifying both the C code and the RTL design.

The advantage of the hardware implementation is that all data stored in the hash table are supplied in sequence and fed to the node under consideration so that it can perform its entropy calculations. This approach differs from existing approaches discussed in section 2.2.5 where data are partitioned into subsets. The hardware implementation improves the performance of HFTDT when compared to the software results obtained in section 6.5. The hardware simulation results have been generated for a range of FPGAs and gave improvements in runtime execution performance by a factor of up to 9.6 times. The hardware shows improvement in the calculation time when reached similar performance with ITI and six times slower than C4.5.

8 CONCLUSIONS AND FURTHER WORK

This research work has presented novel approaches, namely MDFT and HFTDT, that have been developed as real-time incremental learning methods targeting embedded systems. They were designed with the ability to generate application specific code for both training and classification purposes according to the requirements of the targeted application. This chapter discusses the conclusions of each method in terms of the objectives set out in chapter 1 and indicates potential future work.

8.1 CONCLUSIONS

The aim of this research work has been partially met by developing incremental learning methods that were able to achieve low memory usage suitable for meeting embedded system constraints. Nevertheless, memory usage was still shown to increase as more training vectors are assimilated. By identifying the maximum number of calculations that are needed to build a DT, it would be feasible to determine an upper bound for the time tree building would take, assuming it is possible to know the execution times needed for specific mathematical operations on the target platform.

Referring to the objectives, the literature for existing methods was reviewed, allowing the identification of an appropriate method for building an incremental system. Two solutions have been developed which are MDFT and HFTDT, whose outcomes are discussed in relation the objectives in the next two subsections.

8.1.1 MULTI-DIMENSIONAL FREQUENCY TABLE METHOD (MDFT)

The MDFT learning method introduced in this research work adopts a multi-dimensional array that acts as a frequency table. The frequency table technique was utilized to achieve incremental learning as it holds all the iterations of the

incoming data vectors, while keeping all correlations between the attributes and classes that are necessary to build the decision tree.

A deterministic time solution and memory requirements are issues of importance when targeting real-time embedded applications. The MDFT demonstrated in Chapter 4 has the ability to determine the time needed to perform the necessary calculations to build a DT and with a fixed memory usage. Despite the memory requirements of the MDFT being deterministic and can be known *a priori*, the approach has the limitation that a substantial demand on memory usage occurs when the number of dimensions of the frequency table is increased. This limits the MDFT method's suitability for embedded system applications where the memory is usually limited.

8.1.2 HASHED FREQUENCY TABLE DECISION TREE METHOD (HFTDT)

The HFTDT method was introduced in Chapter 5, and adopts a hashed frequency table technique to permit incremental learning. The method employs a compact version of the frequency table used by MDFT, where a two-dimensional hash table is utilized to save the non-zero elements of the MDFT frequency table. This approach considerably reduces the memory usage when compared to MDFT.

HFTDT is a real-time incremental learning method that targets embedded systems. The method has a deterministic calculation time, as in MDFT, where the number of calculations can be known *a priori*. The memory requirements of the HFTDT depend on the number of elements stored in the hash table, which can increase with the addition of new vectors. The upper limit of the memory requirement for the hash table is not possible to determine *a priori* if the DT is to be operated incrementally. In a practical system, the number of training vectors that can be acquired will need to be restricted, possibly through the implementation of a forgetting algorithm that removes older vectors from the HFT when it exceeds available memory capacity.

Experimental work carried out in Chapter 6 demonstrates good performance of the HFTDT in terms of the number of nodes in the induced DT, memory usage and

classification accuracy when compare to three widely used machine learning methods, such as kNN, C4.5 and ITI. The HFTDT performed well in classifying unseen datasets and when dealing with missing values, even though no pruning technique was used in producing the DTs.

The source code of HFTDT has been profiled and it was determined that the most time-consuming function was consistently *rev_index_func*, it being called many times to provide the necessary information to conduct the entropy and information gain calculations needed to build DTs. Hardware designs for the *rev_index_func* have been introduced using a range of FPGAs where the execution time was achieved by up to 10 times less than that of the software simulation.

8.2 FURTHER WORK

Further work in the areas of pruning and parallelism has been identified and these are discussed below.

Pruning

Improving classification accuracy and overall performance for the HFTDT using a suitable pruning technique can be considered in future work. As mentioned in Chapter 2, pruning aims to generalise DTs and improve classification accuracy. Pre-pruning methods often have a lower computational cost compared to post-pruning techniques, although the latter can achieve better classification accuracy. Pre-pruning can be more feasible for incremental learning than post-pruning since it adopts a set of stopping criteria that can be used to halt the growth of the DT.

HFTDT as an incremental method that has the ability to generate a new DT when any changes in the data occurs or whenever it is needed, since the FT can provide all the data needed to build a DT from scratch,. Although both types of pruning can be implemented in such a case, other incremental methods, such as ITI, adopt pre-pruning that affects the DT building at an early stage. These changes made during tree building cannot later be undone when more data are presented.

In addition to smaller DTs, pruning provides better classification accuracy, fewer nodes and a consequent reduction in memory usage. Execution time is also reduced as fewer calculations are needed in the building of a smaller tree. Consequently, further work is appropriate to identify a suitable pruning technique to achieve an equivalent performance to the C4.5 algorithm, which was able to produce a more generalized DT according to the evaluation results of Chapter 6.

Parallelism

Parallelism has been applied to a number of the decision tree methods found in the literature. The forms of parallelism which have been identified by other authors as in Section 2.2.4 include multi-threaded approaches, node based approaches and entropy based approaches. These methods are broadly applicable to HFTDT, as only the way in which the data are supplied to nodes is different. The data in most DTs are obtained as required by the algorithm, whereas in HFTDT the data is fed continuously from memory and it is the responsibility of the nodes to obtain the data they require as it is generated so that they can perform their entropy calculations. Consequently, the detailed implementation of parallelism in the HFTDT will have some differences from other DT methods, and further work is needed in order to realise this parallelism and to determine the advantages of this approach.

REFERENCES:

- [1] “Embedded System Market - Global Industry Analysis, Size, Share, Growth, Trends and Forecast, 2012- 2018,” *Transparency Market Research*, 2013. [Online]. Available: <http://www.transparencymarketresearch.com/embedded-system.html>. [Accessed: 01-Jan-2014].
- [2] E. Alpaydin, *Introduction to Machine Learning*. The MIT Press Cambridge, Massachusetts. London, England, 2004, p. 415.
- [3] Ryszard S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishing Company, 1983, p. 572.
- [4] *Oxford English Dictionary*, Draft Revi. Oxford University Press, 2010.
- [5] T. M. Mitchell, “The Discipline of Machine Learning,” Technical report, CMU-ML-06-108. Department of Machine Learning, Carnegie Mellon University, 2006.
- [6] C. R. Köpf, *Meta-learning: strategies, implementations, and evaluations for algorithm selection*. Akademische Verlagsgesellschaft Aka GmbH, Berlin, 2006.
- [7] E. L. Lamie, *Real-Time Embedded Multithreading Using ThreadX*, Second. Elsevier Inc., 2009.
- [8] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Third. Springer, 2011, p. 521.
- [9] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment,” *J. Assoc. Comput. Mach.*, vol. 20, no. 1, pp. 46–61, 1973.
- [10] J. C. Schlimmer and R. H. Granger, “Incremental learning from noisy data,” *Mach. Learn.*, vol. 1, no. 3, pp. 317–354, 1986.
- [11] E. Swere, “Machine Learning in Embedded Systems,” Loughborough University, 2008.
- [12] J. Gama, R. Rocha, and P. Medas, “Accurate Decision Trees for Mining High-speed Data Streams,” in *The ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 523–528.
- [13] S. B. Kotsiantis, “Supervised Machine Learning: A Review of Classification Techniques,” *Informatica*, vol. 31, pp. 249–268, 2007.

-
- [14] N. Grira, M. Crucianu, N. Boujemaa, and I. Rocquencourt, “Unsupervised and Semi-supervised Clustering: a Brief Survey,” in *A Review of Machine Learning Techniques for Processing Multimedia Content, Report of the MUSCLE European Network of Excellence (FP6) in*, 2005, pp. 1–12.
- [15] X. Zhu, “Semi-Supervised Learning Literature Survey Contents,” *Comput. Sci. Univ. Wisconsin-Madison*, 2008.
- [16] D. W. Aha, D. Kibler, and M. K. Albert, “Instance-based learning algorithms,” *Mach. Learn.*, vol. 6, no. 1, pp. 37–66, Jan. 1991.
- [17] M. Negnevitsky, *Artificial Intelligence: A Guide to Intelligent Systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001, p. 394.
- [18] D. Mange and M. Tomassini, *Artificial Neural Networks: Algorithms and Hardware Implementation*, 2.2 ed. PPUR Press, 1998, pp. 289–316.
- [19] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines*, First. Cambridge University Press, 2000.
- [20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press Cambridge, Massachusetts. London, England, 1998.
- [21] J. Bernardo and A. Smith, *Bayesian theory*. John Wiley & Sons, 2004.
- [22] K. S. Tang, K. F. Man, S. Kwong, and Q. He, “Genetic algorithms and their applications,” *IEEE Signal Process. Mag.*, vol. 13, no. November, pp. 22–37, 1996.
- [23] L. B. Booker, D. E. Goldberg, and J. H. Holland, “Classifier Systems and Genetic Algorithms,” *Artif. Intell.*, vol. 40, pp. 235–282, 1989.
- [24] J. R. Quinlan, “Induction of Decision Trees,” *Mach. Learn. J.*, vol. 1, no. 1, pp. 81–106, 1986.
- [25] P. Kokol, V. Podgorelec, M. Zorman, and M. M. S, “The Art of Building Decision Trees,” *J. Med. Syst.*, vol. 24, no. 1, pp. 43–52, 2000.
- [26] S. K. MURTHY, “Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey,” *Data Min. Knowl. Discov.*, vol. 2, no. 4, pp. 345–389, 1998.
- [27] J. R. Quinlan, *C4.5: Programs for machine learning*. California: Morgan Kaufmann publishers, inc., 1993.
- [28] J. R. Quinlan, “Data Mining Tools See5 and C5.0,” 2009. [Online]. Available: <http://www.rulequest.com/see5-info.html>. [Accessed: 11-Nov-2010].
- [29] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, “Classification and Regression Trees,” *Wadsworth Stat. Probab. Ser.*, vol. 19, 1984.

-
- [30] J. Shafer and R. Agrawal, "SPRINT: A Scalable Parallel Classifier for Data," in *Proceedings of 22nd international conference on very large data bases*, 1996, pp. 544–555.
- [31] S. Appavu and R. Rajaram, "Knowledge-Based Systems Knowledge-based system for text classification using ID6NB algorithm," *Knowledge-Based Syst.*, vol. 22, no. 1, pp. 1–7, 2009.
- [32] J. C. Schlimmer and D. Fisher, "A Case Study of Incremental concept induction," in *Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986, pp. 496–501.
- [33] P. E. Utgoff, "ID5: An Incremental ID3," in *Proceedings of the Fifth International Conference on Machine Learning*, 1988, pp. 107–120.
- [34] P. E. Utgoff, "Incremental Induction of Decision Trees," in *Machine learning 4*, 1989, no. C, pp. 161–186.
- [35] P. E. Utgoff, N. C. Berkman, and J. A. Clouse, "Decision Tree Induction Based on Efficient Tree Restructuring," *Mach. Learn. J.*, vol. 29, pp. 5–44, 1997.
- [36] E. Swere and D. J. Mulvaney, "Robot Navigation Using Decision Trees," *Electr. Eng.*, pp. 15–17, 2003.
- [37] E. Swere, D. Mulvaney, and I. Sillitoe, "A Fast Memory-Efficient Incremental Decision Tree Algorithm in its Application to Mobile Robot Navigation," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, 2006, pp. 645–650.
- [38] E. Swere, D. Mulvaney, and I. Sillitoe, "Efficient Incremental Decision Tree Generation for Embedded Applications," in *Proceedings of the IEEE international conference on cybernetics and intelligent systems*, 2005, pp. 1101–1106.
- [39] T. Carter, "An introduction to information theory and entropy." 2011.
- [40] T. ELOMAA and J. ROUSU, "General and Efficient Multisplitting of Numerical Attributes," *Mach. Learn. J.*, vol. 36, pp. 201–244, 1999.
- [41] K. P. Bennett, N. Cristianini, J. Shawe-Taylor, and D. Wu, "Enlarging the Margins in Perceptron Decision Trees," *Mach. Learn.*, vol. 41, no. 3, pp. 295–313, 2000.
- [42] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.*, vol. 27, no. July, pp. 379–423, 1948.
- [43] S. B. Kotsiantis, "Decision trees: a recent overview," *Artif. Intell. Rev.*, vol. 39, no. 4, pp. 261–283, Jun. 2011.
- [44] L. Rokach and O. Maimon, "Pruning Trees," in *data mining with decision trees: theory and applications*, World Scientific Publishing, Singapore, 2008, pp. 63–69.

-
- [45] N. J. Nilsson, “Overfitting and Evaluation,” in *INTRODUCTION TO MACHINE LEARNING*, 2005, pp. 80–85.
- [46] H. Kim and G. Koehler, “An investigation on the conditions of pruning an induced decision tree,” *Eur. J. Oper. Res.*, vol. 77, pp. 82–95, 1994.
- [47] L. A. Breslow and D. W. Aha, “Simplifying decision trees: A survey,” *Knowl. Eng. Rev.*, vol. 12, no. 1, pp. 1–40, 1997.
- [48] J. Martin, “An exact probability metric for decision tree splitting and stopping,” *Mach. Learn.*, vol. 291, pp. 257–291, 1997.
- [49] P. Clark and T. Niblett, “The CN2 induction algorithm,” *Mach. Learn.*, vol. 3, pp. 261–283, 1989.
- [50] J. Du, Z. Cai, and C. X. Ling, “Cost-sensitive decision trees with pre-pruning,” *Adv. Artif. Intell.*, pp. 171–179, 2007.
- [51] F. Esposito, D. Malerba, G. Semeraro, and J. Kay, “A comparative analysis of methods for pruning decision trees,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 19, no. 5, pp. 476–493, May 1997.
- [52] J. R. Quinlan, “Simplifying decision trees,” *International Journal of Man-Machine Studies*, vol. 27, pp. 221–234, 1987.
- [53] B. Cestnik and I. Bratko, “On estimating probabilities in tree pruning,” in *Machine Learning — EWSL-91 SE - 11*, vol. 482, Y. Kodratoff, Ed. Springer Berlin Heidelberg, 1991, pp. 138–150.
- [54] J. Mingers, “An empirical comparison of pruning methods for decision tree induction,” *Mach. Learn.*, vol. 4, pp. 227–243, 1989.
- [55] E. Frank, “Pruning decision trees and lists,” University of Waikato, 2000.
- [56] K. Steinhaeuser, N. V Chawla, and P. M. Kogge, “Exploiting Thread-Level Parallelism to Build Decision Trees,” in *International Workshop on Parallel Data Mining*, 2006.
- [57] R. Kufirin, “Decision trees on parallel processors,” *Parallel Process. Artif. Intell.*, vol. 3, 1995.
- [58] A. Srivastava, E. Han, V. Kumar, and V. Singh, “Parallel Formulations of Decision-Tree Classification Algorithms,” *Data Min. Knowl. Discov.*, vol. 3, pp. 237–261, 1999.
- [59] G. J. Narlikar, “A Parallel , Multithreaded Decision Tree Builder. Technical Report CMU-CS-98-184, School of Computer Science, Carnegie Mellon University,” 1998.

-
- [60] R. Jin and G. Agrawal, "Communication and Memory Efficient Parallel Decision Tree Construction," in *Proceedings of the 2003 SLAM International Conference on Data Mining*, 2003, pp. 119–129.
- [61] O. T. Yıldız and O. Dikmen, "Parallel univariate decision trees," *Pattern Recognit. Lett.*, vol. 28, no. 7, pp. 825–832, May 2007.
- [62] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "An FPGA Implementation of Decision Tree Classification," in *Proc. IEEE Int. Conf. Design, Automation and Test in Europe*, 2007, pp. 189–194.
- [63] M. J. Zaki and R. Agrawal, "Parallel classification for data mining on shared-memory multiprocessors," *Proc. 15th Int. Conf. Data Eng. (Cat. No.99CB36337)*, pp. 198–205, 1999.
- [64] H. Andrade, T. Kurc, A. Sussman, J. Saltz, and C. Park, "Decision Tree Construction for Data Mining on Cluster of Shared-Memory Multiprocessors," in *In HPDM: 6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining*, 2003.
- [65] P. T. P. Tang, "Table-Lookup Algorithms for Elementary Functions and Their Error Analysis," in *10th IEEE Symposium on Computer Arithmetic*, 1991, pp. 232–236.
- [66] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," *Proc. 12th Symp. Comput. Arith.*, pp. 10–16, 1995.
- [67] J. E. Stine and M. J. Schulte, "The Symmetric Table Addition Method for Accurate Function Approximation," *J. VLSI signal Process. Syst. signal, image video Technol.*, vol. 21, no. 2, pp. 167–177, 1999.
- [68] "GNU Mathematical Library." [Online]. Available: http://www.gnu.org/software/libc/manual/html_node/Mathematics.html. [Accessed: 20-Mar-2014].
- [69] J. R. Quinlan, "Learning Efficient Classification Procedures and Their Application to Chess End Games," in *Machine Learning SE - 15*, R. Michalski, J. Carbonell, and T. Mitchell, Eds. Springer Berlin Heidelberg, 1983, pp. 463–482.
- [70] Y. Wang, D. Mulvaney, I. Sillitoe, and E. Swere, "Robot Navigation by Waypoints," *J. Intell. Robot. Syst.*, pp. 175–207, 2008.
- [71] H. Zhe, Z. Jun, and L. Xiling, "Design and Realization of Efficient Memory Management for Embedded Real-Time Application," in *2006 6th International Conference on ITS Telecommunications Proceedings*, 2006, pp. 174–177.
- [72] J. R. Alcob, "Incremental Learning of Tree Augmented Naive Bayes Classifiers," in *In: Series lecture notes in computer science: Advances in artificial intelligence - IBERAMLA*, Springer Berlin/Heidelberg publisher, 2002, pp. 32–41.

-
- [73] R. Wan, I. Takigawa, and H. Mamitsuka, "Applying Gaussian Distribution-Dependent Criteria to Decision Trees for High-Dimensional Microarray Data," in *Lecture Notes in Bioinformatics*, Springer-Verlag Berlin Heidelberg, 2006, pp. 40–49.
- [74] Canonical Ltd, "Ubuntu 12.04." [Online]. Available: www.ubuntu.com. [Accessed: 01-Apr-2014].
- [75] ORACLE, "VirtualBox." [Online]. Available: <https://www.virtualbox.org>. [Accessed: 01-Apr-2014].
- [76] I. Free Software Foundation, "GCC, the GNU Compiler Collection." [Online]. Available: <http://gcc.gnu.org/>. [Accessed: 01-Mar-2014].
- [77] "Matlab," 2014. [Online]. Available: <http://www.mathworks.co.uk>. [Accessed: 01-Mar-2014].
- [78] H. Jiawei and M. Kamber, *Data mining: concepts and techniques*, 2nd ed. Morgan Kaufmann Publishers, 2006.
- [79] A. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 50–60, 2005.
- [80] T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Commun. Surv. Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.
- [81] J. Ostlund, P. Komarek, T. Liu, and A. Moore, "Dense K nearest neighbour.," 2005. [Online]. Available: <http://www.autonlab.org/autonweb/downloads/software.html>. [Accessed: 01-Dec-2013].
- [82] J. R. Quinlan, "Quinlan, J. R., Ross Quinlan's personal homepage," 1992. [Online]. Available: <http://www.rulequest.com/Personal>. [Accessed: 25-Jan-2014].
- [83] P. E. Utgoff, "Incremental Decision Tree Induction," 2001. [Online]. Available: <http://www-ml.cs.umass.edu/iti/>. [Accessed: 25-Feb-2014].
- [84] Y. Yang and X. Liu, "A re-examination of text categorization methods," *Proc. 22nd Annu. Int. ACM SIGIR Conf. Res. Dev. Inf. Retr. - SIGIR '99*, pp. 42–49, 1999.
- [85] K. Bache and M. Lichman, "UCI Machine Learning Repository," *University of California, Irvine, School of Information and Computer Sciences*, 2013. [Online]. Available: <http://archive.ics.uci.edu/ml>.
- [86] A. Van Den Bosch and K. Van Der Sloot, "Superlinear Parallelization of k - Nearest Neighbor Retrieval," 2007.

-
- [87] I. H. Witten, E. Frank, and M. A. Hall, “Applications and Beyond,” in *DATA MINING Practical Machine Learning Tools and Techniques*, Third., Morgan Kaufmann Publishers, 2011, pp. 375–381.
- [88] D. Wettschereck and T. G. Dietterich, “An experimental comparison of the nearest-neighbor and nearest-hyperrectangle algorithms,” *Mach. Learn.*, vol. 19, no. 1, pp. 5–27, Apr. 1995.
- [89] H. Liang and Y. Yan, “Improve Decision Trees for Probability-Based Ranking by Lazy Learners,” *2006 18th IEEE Int. Conf. Tools with Artif. Intell.*, pp. 427–435, Nov. 2006.
- [90] E. Acuna and C. Rodriguez, “The treatment of missing values and its effect on classifier accuracy,” *Classif. Clust. Data Min. Appl.*, pp. 639–647, 2004.
- [91] P. E. Utgoff, “An improved algorithm for incremental induction of decision trees,” in *Proceedings of the Eleventh International Conference on Machine Learning*, 1994, pp. 318–325.
- [92] J. Grzymala-Busse and M. Hu, “A comparison of several approaches to missing attribute values in data mining,” *Rough sets Curr. trends Comput.*, pp. 378–385, 2001.
- [93] G. E. a. P. a. Batista and M. C. Monard, “An analysis of four missing data treatment methods for supervised learning,” *Appl. Artif. Intell.*, vol. 17, no. 5–6, pp. 519–533, May 2003.
- [94] J. Fenlason, “GNU gprof manual.” [Online]. Available: <https://sourceware.org/binutils/docs-2.17/gprof/>. [Accessed: 01-Feb-2014].
- [95] G. Martin, G. Smith, D. Tho-, M. Barbacci, and A. Parker, “High-Level Synthesis: Past , Present , and Future,” *IEEE Des. Test Comput.*, pp. 18–25, 2009.
- [96] P. Coussy, D. D. Gajski, M. Meredith, and a. Takach, “An Introduction to High-Level Synthesis,” *IEEE Des. Test Comput.*, vol. 26, no. 4, pp. 8–17, Jul. 2009.
- [97] D. Ivošević and V. Struk, “Unified flow of custom processor design and FPGA implementation,” *EUROCON, 2013 IEEE*, pp. 1721–1727, 2013.
- [98] *Vivado Design Suite User Guide*, UG902 ed. Xilinx Inc., 2013.
- [99] M. Fingeroff, *High-level synthesis blue book*. Xlibris, Philadelphia, 2010.
- [100] Cadence, “Cadence C-to-Silicon Compiler,” 2008. [Online]. Available: http://www.cadence.com/rl/resources/technical_papers/c_to_silicon_tp.pdf. [Accessed: 15-Apr-2014].
- [101] Compaan_Design, “Compaan.” [Online]. Available: <http://www.compaandesign.com/>. [Accessed: 15-Apr-2014].

- [102] NEC, “CyberWorkBench.” [Online]. Available: <http://www.cyberworkbench.com>. [Accessed: 13-Apr-2014].
- [103] Synopsys, “Synphony C.” [Online]. Available: <http://www.synopsys.com/>. [Accessed: 15-Apr-2014].
- [104] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, “An overview of today’s high-level synthesis tools,” *Des. Autom. Embed. Syst.*, vol. 16, no. 3, pp. 31–51, Aug. 2012.
- [105] BDTI, “An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool (White paper),” 2010. [Online]. Available: <http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot>. [Accessed: 01-Mar-2014].
- [106] *Introduction to FPGA Design with Vivado High-Level Synthesis*, UG998 ed. Xilinx Inc., 2013.
- [107] “7 Series FPGAs Overview,” vol. 180. XILINX Inc., pp. 1–16, 2013.
- [108] “Zynq-7000 All Programmable SoC Overview.” Xilinx Inc., pp. 1–21, 2013.
- [109] Xilinx Inc., “Xilinx 7-series product brief,” 2012. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/7-Series-Product-Brief.pdf. [Accessed: 18-Feb-2014].
- [110] T. Feist, “Vivado Design Suite, White Paper,” Xilinx Inc., 2012.

APPENDIX A

AN ILLUSTRATIVE EXAMPLE FOR HFTDT METHOD

The weather problem given in Table 5.2 consists of four attributes ($N_a = 4$) and two classes ($N_c = 2$). The HFTDT method uses an HFT that holds the keys and the corresponding values of each input vector.

After the HFT has been populated, the decision tree can be generated. The splitting criterion used in generating the decision tree is information gain.

The following shows the total entropy calculation needed for the root node, using Equation 2.2.

$$\begin{aligned} Q_{c_0}^1 &= \sum \text{Values in HFT where the class} = (\text{Don't Play}) \\ &= f(0) + f(2) + f(8) + f(58) + f(70) = 5 \end{aligned} \quad (\text{A.1})$$

$$\begin{aligned} Q_{c_1}^1 &= \sum \text{Values in HFT where the class} = (\text{Don't Play}) \\ &= (f(9) + f(15) + f(21) + f(25) + f(29) + f(35) + f(47) + f(57) \\ &\quad + f(61) + f(69)) = 11 \end{aligned} \quad (\text{A.2})$$

$$Q_{c_T}^1 = \sum \text{Values in HFT of all classes} = Q_{c_0}^1 + Q_{c_1}^1 = 16 \quad (\text{A.3})$$

The total entropy at the root node is thus

$$\begin{aligned} E_T^1 &= \left(-\frac{Q_{c_0}^1}{Q_{c_T}^1} \log_2 \frac{Q_{c_0}^1}{Q_{c_T}^1} \right) - \left(\frac{Q_{c_1}^1}{Q_{c_T}^1} \log_2 \frac{Q_{c_1}^1}{Q_{c_T}^1} \right) \\ &= \left(-\frac{5}{16} \log_2 \frac{5}{16} \right) - \left(\frac{11}{16} \log_2 \frac{11}{16} \right) = 0.896038 \end{aligned} \quad (\text{A.4})$$

Choosing the attribute at the root node

In this section, the entropy values will be computed for each of the attributes for the root node.

1. Entropy for 'outlook'.

For attribute value 'sunny'

$$\begin{aligned} Q_{C_0^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (0, 2, 8)} \\ &= 3 \end{aligned} \tag{A.5}$$

$$\begin{aligned} Q_{C_1^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (9, 15, 21)} \\ &= 4 \end{aligned} \tag{A.6}$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 7 \tag{A.7}$$

For attribute value 'overcast'

$$Q_{C_0^1}^1 = \text{No entries available for class (don't play)} \tag{A.8}$$

$$\begin{aligned} Q_{C_1^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (25, 29, 35, 47)} \\ &= 4 \end{aligned} \tag{A.9}$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 4 \tag{A.10}$$

For attribute value 'rainy'.

$$\begin{aligned} Q_{C_0^2}^1 &= \sum \text{Values of the HFT that correspond to the keys (58, 70)} \\ &= 2 \end{aligned} \tag{A.11}$$

$$\begin{aligned} Q_{C_1^2}^1 &= \sum \text{Values of the HFT that correspond to the keys (57, 61, 69)} \\ &= 3 \end{aligned} \tag{A.12}$$

$$Q_{C_T^2}^1 = Q_{C_0^2}^1 + Q_{C_1^2}^1 = 5 \tag{A.13}$$

By using Equation 2.4, the entropy for attribute 'outlook' can be found from:

$$\begin{aligned}
 E^1(A_1) = & \frac{Q_{C_T}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0}^1}{Q_{C_T}^1} \log_2 \frac{Q_{C_0}^1}{Q_{C_T}^1} \right) - \left(\frac{Q_{C_1}^1}{Q_{C_T}^1} \log_2 \frac{Q_{C_1}^1}{Q_{C_T}^1} \right) \right] \\
 & + \frac{Q_{C_T}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0}^1}{Q_{C_T}^1} \log_2 \frac{Q_{C_0}^1}{Q_{C_T}^1} \right) - \left(\frac{Q_{C_1}^1}{Q_{C_T}^1} \log_2 \frac{Q_{C_1}^1}{Q_{C_T}^1} \right) \right] \\
 & + \frac{Q_{C_T}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0}^2}{Q_{C_T}^1} \log_2 \frac{Q_{C_0}^2}{Q_{C_T}^1} \right) - \left(\frac{Q_{C_1}^2}{Q_{C_T}^1} \log_2 \frac{Q_{C_1}^2}{Q_{C_T}^1} \right) \right] = 0.734459
 \end{aligned} \tag{A.14}$$

By using Equation 2.5, the information gain for attribute 'outlook' is given by:

$$\begin{aligned}
 IG^1(A_1) &= E_T^1 - E^1(A_1) \\
 &= 0.896038 - 0.734459 = 0.161579 \\
 &\text{(This is the largest gain at this node)}
 \end{aligned} \tag{A.15}$$

2. Entropy for 'temperature':

For attribute value 'hot'.

$$\begin{aligned}
 Q_{C_0}^1 &= \sum \text{Values of the HFT that correspond to the keys (0, 2)} \\
 &= 2
 \end{aligned}$$

$$\begin{aligned}
 Q_{C_1}^1 &= \sum \text{Values of the HFT that correspond to the keys (25, 29)} \\
 &= 2
 \end{aligned}$$

$$Q_{C_T}^1 = Q_{C_0}^1 + Q_{C_1}^1 = 4$$

For attribute value 'mild'.

$$\begin{aligned}
 Q_{C_0}^1 &= \sum \text{Values of the HFT that correspond to the keys (8, 58)} \\
 &= 2
 \end{aligned}$$

$$\begin{aligned}
 Q_{C_1}^1 &= \sum \text{Values of the HFT that correspond to the keys (9, 15, 35, 57, 61)} \\
 &= 6
 \end{aligned}$$

$$Q_{C_T}^1 = Q_{C_0}^1 + Q_{C_1}^1 = 8$$

For attribute value 'cool'.

$$\begin{aligned} Q_{C_0^1}^1 &= \sum \text{Value of the HFT that corresponds to the key (70)} \\ &= 1 \end{aligned}$$

$$\begin{aligned} Q_{C_1^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (21, 47, 69)} \\ &= 3 \end{aligned}$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 4$$

By using Equation 2.4, the entropy for attribute 'temperature' can be found from

$$\begin{aligned} E^1(A_2) &= \frac{Q_{C_T^1}^1}{Q_{C_T^1}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T^1}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T^1}^1} \left[\left(-\frac{Q_{C_2^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_2^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] = 0.858459 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'temperature' is

$$\begin{aligned} IG^1(A_2) &= E_T^1 - E^1(A_2) \\ &= 0.896038 - 0.858459 = 0.03758 \end{aligned}$$

(This is not the largest gain at this node)

3. Entropy for attribute 'humidity'

For attribute value 'high'

$$\begin{aligned} Q_{C_0^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (0, 2, 8, 58)} \\ &= 4 \end{aligned}$$

$$\begin{aligned} Q_{C_1^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (25, 9, 57, 35)} \\ &= 5 \end{aligned}$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 9$$

For attribute value normal

$$\begin{aligned} Q_{C_0^1}^1 &= \sum \text{Value of the HFT that corresponds to the key (70)} \\ &= 1 \end{aligned}$$

$$\begin{aligned} Q_{C_1^1}^1 &= \sum \text{Value of the HFT that corresponds to the key (15, 21, 29, 47, 61, 69)} \\ &= 6 \end{aligned}$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 7$$

By using Equation 2.4, the entropy for attribute 'humidity' can be found from

$$\begin{aligned} E^1(A_3) &= \frac{Q_{C_T^1}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_T^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_T^1}^1} \right) \right] = 0.816337 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'humidity' is given by

$$\begin{aligned} IG^1(A_3) &= E_T^1 - E^1(A_3) \\ &= 0.896038 - 0.816337 = 0.079701 \end{aligned}$$

(this is not the largest gain at this node)

4. Entropy for attribute 'wind'

For attribute value 'weak'

$$\begin{aligned} Q_{C_0^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (0, 8)} \\ &= 2 \end{aligned}$$

$$\begin{aligned} Q_{C_1^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (9,21,25,29,57,61,69)} \\ &= 8 \end{aligned}$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 10$$

For attribute value 'strong'

$$\begin{aligned} Q_{C_0^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (2, 58, 70)} \\ &= 3 \end{aligned}$$

$$\begin{aligned} Q_{C_1^1}^1 &= \sum \text{Values of the HFT that correspond to the keys (15, 35, 47)} \\ &= 3 \end{aligned}$$

$$Q_{C_T^1}^1 = Q_{C_0^1}^1 + Q_{C_1^1}^1 = 6$$

By using Equation 2.4, the entropy for attribute 'wind' can be found from

$$\begin{aligned} E^1(A_4) &= \frac{Q_{C_T^0}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^0}^1}{Q_{C_0^0}^1} \log_2 \frac{Q_{C_0^0}^1}{Q_{C_0^0}^1} \right) - \left(\frac{Q_{C_1^0}^1}{Q_{C_1^0}^1} \log_2 \frac{Q_{C_1^0}^1}{Q_{C_1^0}^1} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^1}{Q_{C_T}^1} \left[\left(-\frac{Q_{C_0^1}^1}{Q_{C_0^1}^1} \log_2 \frac{Q_{C_0^1}^1}{Q_{C_0^1}^1} \right) - \left(\frac{Q_{C_1^1}^1}{Q_{C_1^1}^1} \log_2 \frac{Q_{C_1^1}^1}{Q_{C_1^1}^1} \right) \right] = 0.826205 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'wind' is given by

$$\begin{aligned} IG^1(A_4) &= E_T^1 - E^1(A_4) \\ &= 0.896038 - 0.826205 = 0.069833 \end{aligned}$$

(this is not the largest gain at this node)

The DT progress beyond this point is shown in Figure 4.5. For the root node, the attribute 'outlook' is chosen as it has the largest information gain value. There are three attribute values for 'outlook' sunny, overcast and rainy. Therefore the root node has three children as follows.

Sunny --> node 2

Overcast --> node 3

Rainy --> node 4

Calculations of node 2

After choosing 'outlook' for node 1 according to the entropy and information gain calculations given by Equations A.14 and A.15, the following calculations are for node 2 to choose the next attribute from the remaining attributes following similar procedure.

Three main relations will be considered in this stage.

1. Outlook-->Temperature
2. Outlook-->'humidity'
3. Outlook-->Wind

For node 2 where value 'sunny' presented, the number of vectors is 7 as given by Equation A.7. In the following, the calculations for node 2 to choose the next attribute.

Total entropy for node 2

For first child node 2 for root node with attribute value 'sunny', the total node entropy can be calculated using the following parameters by referring to Equations A.5, A.6 and A.7.

$$Q_{C_0}^2 = Q_{C_0^1}^1 = 3$$

$$Q_{C_1}^2 = Q_{C_1^1}^1 = 4$$

$$Q_{C_T}^2 = Q_{C_T^1}^1 = 7$$

From Equation 2.2, the entropy, for all vectors within attribute value 'sunny' can be found from

$$E_T^2 = \left(-\frac{Q_{C_0}^2}{Q_{C_T}^2} \log_2 \frac{Q_{C_0}^2}{Q_{C_T}^2} \right) - \left(\frac{Q_{C_1}^2}{Q_{C_T}^2} \log_2 \frac{Q_{C_1}^2}{Q_{C_T}^2} \right) = \left(-\frac{3}{7} \log_2 \frac{3}{7} \right) - \left(\frac{4}{7} \log_2 \frac{4}{7} \right) \quad (\text{A.16})$$

$$= 0.985228$$

Attribute entropy for node 2

1. The entropy for attribute 'temperature' when 'outlook' is 'sunny' can be found as follows.

For attribute value 'hot'

$$Q_{C_0}^2 = \sum \text{Values of the HFT that correspond to the keys (0, 2)} \quad (\text{A.17})$$

$$= 2$$

$$Q_{C_1}^2 = \text{No entries available for class (play)} \quad (\text{A.18})$$

$$Q_{C_T}^2 = Q_{C_0}^2 + Q_{C_1}^2 = 2 \quad (\text{A.19})$$

For attribute value 'mild'

$$Q_{C_0}^2 = \sum \text{Value of the HFT that corresponds to the key (8)} \quad (\text{A.20})$$

$$= 1$$

$$Q_{C_1}^2 = \sum \text{Values of the HFT that correspond to the keys (9, 15)} \quad (\text{A.21})$$

$$= 3$$

$$Q_{C_T}^2 = Q_{C_0}^2 + Q_{C_1}^2 = 4 \quad (\text{A.22})$$

For attribute value cool

$$Q_{C_0^2}^2 = \text{No entries available for class (don't play)} \quad (\text{A.23})$$

$$\begin{aligned} Q_{C_1^2}^2 &= \sum \text{Value of the HFT that corresponds to the key (21)} \\ &= 1 \end{aligned} \quad (\text{A.24})$$

$$Q_{C_T^2}^2 = Q_{C_0^2}^2 + Q_{C_1^2}^2 = 1 \quad (\text{A.25})$$

By using Equation 2.4, the entropy for attribute 'temperature' can be found from

$$\begin{aligned} E^2(A_2) &= \frac{Q_{C_T^0}^2}{Q_{C_T}^2} \left[\left(-\frac{Q_{C_0^0}^2}{Q_{C_T^0}^2} \log_2 \frac{Q_{C_0^0}^2}{Q_{C_T^0}^2} \right) - \left(\frac{Q_{C_1^0}^2}{Q_{C_T^0}^2} \log_2 \frac{Q_{C_1^0}^2}{Q_{C_T^0}^2} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^2}{Q_{C_T}^2} \left[\left(-\frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \right) - \left(\frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \right) \right] \\ &\quad + \frac{Q_{C_T^2}^2}{Q_{C_T}^2} \left[\left(-\frac{Q_{C_0^2}^2}{Q_{C_T^2}^2} \log_2 \frac{Q_{C_0^2}^2}{Q_{C_T^2}^2} \right) - \left(\frac{Q_{C_1^2}^2}{Q_{C_T^2}^2} \log_2 \frac{Q_{C_1^2}^2}{Q_{C_T^2}^2} \right) \right] = 0.463587 \end{aligned} \quad (\text{A.26})$$

By using Equation 2.5, the information gain for attribute 'temperature' is given by

$$\begin{aligned} IG^2(A_2) &= E_T^2 - E^2(A_2) \\ &= 0.985228 - 0.463587 = 0.521641 \\ &\quad (\text{this is the largest gain at this node}) \end{aligned} \quad (\text{A.27})$$

2. Entropy for 'humidity' when 'outlook' is 'sunny'

For attribute value 'high'

$$\begin{aligned} Q_{C_0^0}^2 &= \sum \text{Values of the HFT that correspond to the keys (0, 2, 8)} \\ &= 3 \end{aligned}$$

$$\begin{aligned} Q_{C_1^0}^2 &= \sum \text{Values of the HFT that correspond to the keys (9)} \\ &= 2 \end{aligned}$$

$$Q_{C_T^0}^2 = Q_{C_0^0}^2 + Q_{C_1^0}^2 = 5$$

For attribute value 'normal'

$$Q_{C_0}^2 = \text{No entries available for class(don't play)}$$

$$\begin{aligned} Q_{C_1}^2 &= \sum \text{Values of the HFT that correspond to the keys (15, 21)} \\ &= 2 \end{aligned}$$

$$Q_{C_T}^2 = Q_{C_0}^2 + Q_{C_1}^2 = 2$$

By using Equation 2.4, the entropy for attribute 'humidity' can be found from

$$\begin{aligned} E^2(A_3) &= \frac{Q_{C_T}^2}{Q^2} \left[\left(-\frac{Q_{C_0}^2}{Q_{C_T}^2} \log_2 \frac{Q_{C_0}^2}{Q_{C_T}^2} \right) - \left(\frac{Q_{C_1}^2}{Q_{C_T}^2} \log_2 \frac{Q_{C_1}^2}{Q_{C_T}^2} \right) \right] \\ &\quad + \frac{Q_{C_T}^2}{Q^2} \left[\left(-\frac{Q_{C_0}^2}{Q_{C_T}^2} \log_2 \frac{Q_{C_0}^2}{Q_{C_T}^2} \right) - \left(\frac{Q_{C_1}^2}{Q_{C_T}^2} \log_2 \frac{Q_{C_1}^2}{Q_{C_T}^2} \right) \right] = 0.693536 \end{aligned}$$

By using Equation 2.5, the information gain for 'humidity' is given by

$$\begin{aligned} IG^2(A_3) &= E_T^2 - E^2(A_3) \\ &= 0.985228 - 0.693536 = 0.291692 \\ &\text{(this is not the largest gain for this node)} \end{aligned}$$

3. Entropy for 'wind' when 'outlook' is 'sunny'

For attribute value 'weak'

$$\begin{aligned} Q_{C_0}^2 &= \sum \text{Values of the HFT that correspond to the keys (0, 8)} \\ &= 2 \end{aligned}$$

$$\begin{aligned} Q_{C_1}^2 &= \sum \text{Values of the HFT that correspond to the keys (9, 21)} \\ &= 3 \end{aligned}$$

$$Q_{C_T}^2 = Q_{C_0}^2 + Q_{C_1}^2 = 5$$

For attribute value 'strong'

$$Q_{C_0^2}^2 = \sum \text{Values of the HFT that correspond to the keys (2)}$$

$$= 1$$

$$Q_{C_1^2}^2 = \sum \text{Values of the HFT that correspond to the keys (15)}$$

$$= 1$$

$$Q_{C_T^2}^2 = Q_{C_0^2}^2 + Q_{C_1^2}^2 = 2$$

By using Equation 2.4, the entropy for 'wind' can be found from

$$E^2(A_4) = \frac{Q_{C_T^2}^2}{Q_{C_T^2}^2} \left[\left(-\frac{Q_{C_0^2}^2}{Q_{C_T^2}^2} \log_2 \frac{Q_{C_0^2}^2}{Q_{C_T^2}^2} \right) - \left(\frac{Q_{C_1^2}^2}{Q_{C_T^2}^2} \log_2 \frac{Q_{C_1^2}^2}{Q_{C_T^2}^2} \right) \right]$$

$$+ \frac{Q_{C_T^2}^2}{Q_{C_T^2}^2} \left[\left(-\frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_0^1}^2}{Q_{C_T^1}^2} \right) - \left(\frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \log_2 \frac{Q_{C_1^1}^2}{Q_{C_T^1}^2} \right) \right] = 0.97925$$

By using Equation 2.5, the information gain for 'wind' is given by

$$IG^2(A_4) = E_T^2 - E^2(A_4)$$

$$= 0.985228 - 0.97925 = 0.005978$$

(this is not the largest gain at this node)

The DT progress is shown in Figure 4.6, where attribute 'temperature' is chosen as it has the largest information gain value as given by Equation A.27.

Calculations of node 3

Next node is node 3 where attribute value 'overcast' presented, the number of vectors is 4 as given by Equation A.10. The calculations for the entropy and the information gain will be repeated on this node as follows.

Total entropy for node 3

For second child node 3 for root node with value 'overcast', the total node entropy can be calculated using the following parameters by referring to Equations A.8, A.9 and A.10.

$$Q_{C_0}^3 = Q_{C_0^1}^1 = 0$$

$$Q_{C_1}^3 = Q_{C_1^1}^1 = 4$$

$$Q_{C_T}^3 = Q_{C_T^1}^1 = 4$$

From Equation 2.2 the entropy, for all vectors within value 'overcast' is given by

$$E_T^3 = \left(-\frac{Q_{C_0}^3}{Q_{C_T}^3} \log_2 \frac{Q_{C_0}^3}{Q_{C_T}^3} \right) - \left(\frac{Q_{C_1}^3}{Q_{C_T}^3} \log_2 \frac{Q_{C_1}^3}{Q_{C_T}^3} \right) = \left(-\frac{0}{4} \log_2 \frac{0}{4} \right) - \left(\frac{4}{4} \log_2 \frac{4}{4} \right) = 0 \quad (\text{A.28})$$

According to the entropy calculation of node 3 is a leaf node as all the vectors only to one class. The DT progress beyond this point is shown in Figure 4.7.

Calculations of node4

For node 4 where value 'rainy' presented, the number of vectors is 5 as given by Equation A.13. In the following, the calculations for node 4 to choose the next attribute.

Total entropy for node 4

For third child node 4 for root node with attribute value 'rainy', the total node entropy can be calculated using the following parameters by referring to Equations A.11, A.12 and A.13.

$$Q_{C_0}^4 = Q_{C_0^2}^1 = 2$$

$$Q_{C_1}^4 = Q_{C_1^2}^1 = 3$$

$$Q_{C_T}^4 = Q_{C_T^2}^1 = 5$$

From Equation 2.2, the entropy for all vectors within value 'rainy' can be found from

$$E_T^4 = \left(-\frac{Q_{C_0}^4}{Q_{C_T}^4} \log_2 \frac{Q_{C_0}^4}{Q_{C_T}^4} \right) - \left(\frac{Q_{C_1}^4}{Q_{C_T}^4} \log_2 \frac{Q_{C_1}^4}{Q_{C_T}^4} \right) = \left(-\frac{2}{5} \log_2 \frac{2}{5} \right) - \left(\frac{3}{5} \log_2 \frac{3}{5} \right)$$

$$= 0.970951$$

Attribute entropy for node 4

1. Entropy for attribute 'temperature' when 'outlook' is 'rainy'

For attribute value 'hot'

$$Q_{C_0}^4 = \text{No entries available for class (don't play)}$$

$$Q_{C_1}^4 = \text{No entries available for class (play)}$$

$$Q_{C_T}^4 = Q_{C_0}^4 + Q_{C_1}^4 = 0$$

For attribute value 'mild'

$$Q_{C_0}^4 = \sum \text{Values of the HFT that correspond to the keys (58)}$$

$$= 1$$

$$Q_{C_1}^4 = \sum \text{Values of the HFT that correspond to the keys (57, 61)}$$

$$= 2$$

$$Q_{C_T}^4 = Q_{C_0}^4 + Q_{C_1}^4 = 3$$

For attribute value 'cool'

$$\begin{aligned} Q_{C_0^2}^4 &= \sum \text{Value of the HFT that corresponds to the key (70)} \\ &= 1 \end{aligned}$$

$$\begin{aligned} Q_{C_1^2}^4 &= \sum \text{Value of the HFT that corresponds to the key (69)} \\ &= 1 \end{aligned}$$

$$Q_{C_T^2}^4 = Q_{C_0^2}^4 + Q_{C_1^2}^4 = 2$$

By using Equation 2.4, the entropy for attribute 'temperature' can be found from

$$\begin{aligned} E^4(A_2) &= \frac{Q_{C_T^0}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^0}^4}{Q_{C_T^0}^4} \log_2 \frac{Q_{C_0^0}^4}{Q_{C_T^0}^4} \right) - \left(\frac{Q_{C_1^0}^4}{Q_{C_T^0}^4} \log_2 \frac{Q_{C_1^0}^4}{Q_{C_T^0}^4} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^1}^4}{Q_{C_T^1}^4} \log_2 \frac{Q_{C_0^1}^4}{Q_{C_T^1}^4} \right) - \left(\frac{Q_{C_1^1}^4}{Q_{C_T^1}^4} \log_2 \frac{Q_{C_1^1}^4}{Q_{C_T^1}^4} \right) \right] \\ &\quad + \frac{Q_{C_T^2}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^2}^4}{Q_{C_T^2}^4} \log_2 \frac{Q_{C_0^2}^4}{Q_{C_T^2}^4} \right) - \left(\frac{Q_{C_1^2}^4}{Q_{C_T^2}^4} \log_2 \frac{Q_{C_1^2}^4}{Q_{C_T^2}^4} \right) \right] = 0.950978 \end{aligned} \quad (\text{A.29})$$

By using Equation 2.5, the information gain for attribute 'temperature' is given by

$$\begin{aligned} IG^4(A_2) &= E_T^4 - E^4(A_2) \\ &= 0.970951 - 0.950978 = 0.019973 \\ &\quad (\text{this is the largest gain at this node}) \end{aligned} \quad (\text{A.30})$$

2. Entropy for 'humidity' when 'outlook' is 'rainy'

For attribute value 'high'

$$\begin{aligned} Q_{C_0^0}^4 &= \sum \text{Value of the HFT that corresponds to the key (58)} \\ &= 1 \end{aligned}$$

$$\begin{aligned} Q_{C_1^0}^4 &= \sum \text{Value of the HFT that corresponds to the key (57)} \\ &= 1 \end{aligned}$$

$$Q_{C_T^0}^4 = Q_{C_0^0}^4 + Q_{C_1^0}^4 = 2$$

For attribute value 'normal'

$$\begin{aligned} Q_{C_0^4}^4 &= \sum \text{Value of the HFT that corresponds to the key (70)} \\ &= 1 \end{aligned}$$

$$\begin{aligned} Q_{C_1^4}^4 &= \sum \text{Values of the HFT that correspond to the keys (61, 69)} \\ &= 2 \end{aligned}$$

$$Q_{C_T^4}^4 = Q_{C_0^4}^4 + Q_{C_1^4}^4 = 3$$

By using Equation 2.4, the entropy for attribute 'humidity' can be found from

$$\begin{aligned} E^4(A_3) &= \frac{Q_{C_T^0}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^0}^4}{Q_{C_T^0}^4} \log_2 \frac{Q_{C_0^0}^4}{Q_{C_T^0}^4} \right) - \left(\frac{Q_{C_1^0}^4}{Q_{C_T^0}^4} \log_2 \frac{Q_{C_1^0}^4}{Q_{C_T^0}^4} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^1}^4}{Q_{C_T^1}^4} \log_2 \frac{Q_{C_0^1}^4}{Q_{C_T^1}^4} \right) - \left(\frac{Q_{C_1^1}^4}{Q_{C_T^1}^4} \log_2 \frac{Q_{C_1^1}^4}{Q_{C_T^1}^4} \right) \right] = 0.950978 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'humidity' is

$$\begin{aligned} IG^4(A_3) &= E_T^4 - E^4(A_3) \\ &= 0.970951 - 0.950978 = 0.019973 \end{aligned}$$

(this is not the largest gain at this node)

3. Entropy for attribute 'wind' when 'outlook' is 'rainy'

For attribute value 'weak'

$$Q_{C_0^4}^4 = \text{No entries available for class (don't play)} \quad (\text{A.31})$$

$$\begin{aligned} Q_{C_1^4}^4 &= \sum \text{Values of the HFT that correspond to the keys (57, 61, 69)} \\ &= 3 \end{aligned} \quad (\text{A.32})$$

$$Q_{C_T^4}^4 = Q_{C_0^4}^4 + Q_{C_1^4}^4 = 3 \quad (\text{A.33})$$

For attribute value 'strong'

$$\begin{aligned} Q_{C_0^4}^4 &= \sum \text{Values of the HFT that correspond to the keys (58, 70)} \\ &= 2 \end{aligned} \quad (\text{A.34})$$

$$Q_{C_1^4}^4 = \text{No entries available for class (play)} \quad (\text{A.35})$$

$$Q_{C_T^4}^4 = Q_{C_0^4}^4 + Q_{C_1^4}^4 = 2 \quad (\text{A.36})$$

By using Equation 2.4, the entropy for 'wind' can be found from

$$\begin{aligned} E^4(A_4) &= \frac{Q_{C_T^0}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^0}^4}{Q_{C_T^0}^4} \log_2 \frac{Q_{C_0^0}^4}{Q_{C_T^0}^4} \right) - \left(\frac{Q_{C_1^0}^4}{Q_{C_T^0}^4} \log_2 \frac{Q_{C_1^0}^4}{Q_{C_T^0}^4} \right) \right] \\ &\quad + \frac{Q_{C_T^1}^4}{Q_{C_T}^4} \left[\left(-\frac{Q_{C_0^1}^4}{Q_{C_T^1}^4} \log_2 \frac{Q_{C_0^1}^4}{Q_{C_T^1}^4} \right) - \left(\frac{Q_{C_1^1}^4}{Q_{C_T^1}^4} \log_2 \frac{Q_{C_1^1}^4}{Q_{C_T^1}^4} \right) \right] = 0 \end{aligned}$$

By using Equation 2.5, the information gain for 'wind' is given by

$$\begin{aligned} IG^4(A_4) &= E_T^4 - E^4(A_4) \\ &= 0.970951 - 0 = 0.970951 \\ &\quad (\text{this is the largest gain at this node}) \end{aligned}$$

The DT progress is shown in Figure 4.8, shows that attribute 'wind' is chosen as it has the largest information gain value. There are three values for 'temperature' hot, mild and cool. Therefore node 2 has three children as follows.

Hot --> node 5

Mild --> node 6

Cool --> node 7

Calculations of node 5

Next node is node 5 where its parent node 2 chooses attribute 'temperature' as given by Equation A.27. The following calculations are for node 5 to choose the next attribute from the remaining two attributes.

Two main relations to be considered at this stage are

1. Outlook-->Temperature-->'humidity'
2. Outlook--> Temperature-->Wind

For node 5 where value 'hot' is presented, the number of vectors is 2 as given by Equation A.19. In the following, the calculations for node 5 to choose the next attribute.

Total entropy for node 5

For first child node 5 to parent node 2 with attribute value 'hot', the total node entropy can be calculated using the following parameters that are given by Equations A.17, A.18 and A.19.

$$Q_{C_0}^5 = Q_{C_0^0}^1 = 2$$

$$Q_{C_1}^5 = Q_{C_1^0}^1 = 0$$

$$Q_{C_T}^5 = Q_{C_T^0}^1 = 2$$

From Equation 2.2 the total entropy can be found from

$$E_T^5 = \left(-\frac{Q_{C_0}^5}{Q_{C_T}^5} \log_2 \frac{Q_{C_0}^5}{Q_{C_T}^5} \right) - \left(\frac{Q_{C_1}^5}{Q_{C_T}^5} \log_2 \frac{Q_{C_1}^5}{Q_{C_T}^5} \right) = 0$$

According to the entropy calculation, node 5 is a leaf node as all the vectors belongs to class 0. The DT progress beyond this point has been updated as shown in Figure 4.9.

Calculations of node 6

For node 6 where value 'mild' is presented, the number of vectors is 4 as given by Equation A.22. In the following, the calculations of node 6 to choose the next attribute.

Total entropy for node 6

For second child node 6 to parent node 2 with value 'mild', the total entropy can be calculated using the following parameters that are given by Equations A.20, A.21 and A.22.

$$Q_{C_0}^6 = Q_{C_0^1}^2 = 1$$

$$Q_{C_1}^6 = Q_{C_1^1}^2 = 3$$

$$Q_{C_T}^6 = Q_{C_T^1}^2 = 4$$

From Equation 2.2 the total entropy can be found from

$$E_T^6 = \left(-\frac{Q_{C_0}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_0}^6}{Q_{C_T}^6} \right) - \left(\frac{Q_{C_1}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_1}^6}{Q_{C_T}^6} \right) = 0.811278$$

Attribute entropy for node 6

1. Entropy for attribute 'humidity'

For attribute value high

$$Q_{C_0^0}^6 = \sum_{= 1} \text{Value of the HFT that corresponds to the key (8)} \quad (\text{A.37})$$

$$Q_{C_1^0}^6 = \sum_{= 2} \text{Value of the HFT that corresponds to the key (9)} \quad (\text{A.38})$$

$$Q_{C_T^0}^6 = Q_{C_0^0}^6 + Q_{C_1^0}^6 = 3 \quad (\text{A.39})$$

For attribute value 'normal'

$$Q_{C_0^6}^6 = \text{No entries available for class (don't play)} \quad (\text{A.40})$$

$$\begin{aligned} Q_{C_1^6}^6 &= \sum \text{Value of the HFT that corresponds to the key (15)} \\ &= 1 \end{aligned} \quad (\text{A.41})$$

$$Q_{C_T^6}^6 = Q_{C_0^6}^6 + Q_{C_1^6}^6 = 1 \quad (\text{A.42})$$

By using Equation 2.4, the entropy for attribute 'humidity' can be found from

$$\begin{aligned} E^6(A_3) &= \frac{Q_{C_T^6}^6}{Q_{C_T^6}^6} \left[\left(-\frac{Q_{C_0^6}^6}{Q_{C_T^6}^6} \log_2 \frac{Q_{C_0^6}^6}{Q_{C_T^6}^6} \right) - \left(\frac{Q_{C_1^6}^6}{Q_{C_T^6}^6} \log_2 \frac{Q_{C_1^6}^6}{Q_{C_T^6}^6} \right) \right] \\ &\quad + \frac{Q_{C_T^6}^6}{Q_{C_T^6}^6} \left[\left(-\frac{Q_{C_0^6}^6}{Q_{C_T^6}^6} \log_2 \frac{Q_{C_0^6}^6}{Q_{C_T^6}^6} \right) - \left(\frac{Q_{C_1^6}^6}{Q_{C_T^6}^6} \log_2 \frac{Q_{C_1^6}^6}{Q_{C_T^6}^6} \right) \right] = 0.688722 \end{aligned}$$

By using Equation 2.5, the information gain for attribute 'humidity' is given by

$$\begin{aligned} IG^6(A_3) &= E_T^6 - E^6(A_3) \\ &= 0.811278 - 0.688722 = 0.122556 \\ &\quad (\text{this is the largest gain at this node}) \end{aligned} \quad (\text{A.43})$$

2. Entropy for attribute 'wind'

For attribute value 'weak'

$$\begin{aligned} Q_{C_0^6}^6 &= \sum \text{Value of the HFT that corresponds to the key (8)} \\ &= 1 \end{aligned}$$

$$\begin{aligned} Q_{C_1^6}^6 &= \sum \text{Value of the HFT that corresponds to the key (9)} \\ &= 2 \end{aligned}$$

$$Q_{C_T^6}^6 = Q_{C_0^6}^6 + Q_{C_1^6}^6 = 3$$

For attribute value 'strong'

$$Q_{C_0}^6 = \text{No entries available for class(don't play)}$$

$$Q_{C_1}^6 = \sum \text{Values of the HFT that correspond to the keys (15)} \\ = 1$$

$$Q_{C_T}^6 = Q_{C_0}^6 + Q_{C_1}^6 = 1$$

By using Equation 2.4, the entropy for attribute 'wind' can be found from

$$E^6(A_4) = \frac{Q_{C_T}^6}{Q_{C_T}^6} \left[\left(-\frac{Q_{C_0}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_0}^6}{Q_{C_T}^6} \right) - \left(\frac{Q_{C_1}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_1}^6}{Q_{C_T}^6} \right) \right] \\ + \frac{Q_{C_T}^6}{Q_{C_T}^6} \left[\left(-\frac{Q_{C_0}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_0}^6}{Q_{C_T}^6} \right) - \left(\frac{Q_{C_1}^6}{Q_{C_T}^6} \log_2 \frac{Q_{C_1}^6}{Q_{C_T}^6} \right) \right] = 0.688722$$

By using Equation 2.5, the information gain for attribute 'wind' is given by

$$IG^6(A_4) = E_T^6 - E^6(A_4) \tag{A.44} \\ = 0.811278 - 0.688722 = 0.122556 \\ (\text{this is not the largest gain at this node})$$

As both attributes 'humidity' and 'wind' have similar information gain as given by Equations A.42 and A.43, an arbitrary decision [73] made to choose the attribute 'humidity'. The DT has been updated as shown in Figure 4.10, where attribute 'humidity' is chosen for node 6.

Calculations of node 7

For node 7 where the value is cool, the number of vectors is 1 as given by Equation A.25. In the following, the calculations for node 7 to choose the next attribute.

Total entropy for node 7

For the third child node 7 to the parent node 2 with attribute value 'cool', the total node entropy can be calculated using the following parameters that are given by Equations A.23, A.24 and A.25.

$$Q_{c_0}^7 = Q_{c_0^2}^2 = 0$$

$$Q_{c_1}^7 = Q_{c_1^2}^2 = 1$$

$$Q_{c_T}^7 = Q_{c_T^2}^2 = 1$$

From Equation 2.2 the total entropy can be found from

$$E_T^7 = \left(-\frac{Q_{c_0}^7}{Q_{c_T}^5} \log_2 \frac{Q_{c_0}^7}{Q_{c_T}^7} \right) - \left(\frac{Q_{c_1}^7}{Q_{c_T}^7} \log_2 \frac{Q_{c_1}^7}{Q_{c_T}^7} \right) = 0$$

The entropy calculation shows that node 7 is a leaf node as one vector of class 1 is left. The DT beyond this point is shown in Figure 4.11, where the DT is updated with node 7 is a leaf node.

There are two values for 'wind' weak and strong. Therefore there are two children for node 4 as follows.

Weak --> node 8

Strong --> node 9

Two main relations to be considered at this stage

1. Outlook-->Wind-->Humidity
2. Outlook-->Wind--> Temperature

Calculations of node 8

The next node is 8 where its parent node 4 has chosen attribute 'wind' as given by Equation A.30. The following calculations are for node 8 to choose the next attribute from the remaining two attributes.

For node 8 where the value is 'weak', the number of vectors is 3 as given by Equation A.33. In the following, the calculations to choose the next attribute.

Total entropy for node 8

For first child node 8 to parent node 4 with attribute value 'weak', the total node entropy can be calculated using the following parameters that are given by Equations A.31, A.32 and A.33.

$$Q_{C_0}^8 = Q_{C_0^0}^4 = 0$$

$$Q_{C_1}^8 = Q_{C_1^0}^4 = 3$$

$$Q_{C_T}^8 = Q_{C_T^0}^4 = 3$$

From Equation 2.2 the total entropy can be found from

$$E_T^8 = \left(-\frac{Q_{C_0}^8}{Q_{C_T}^8} \log_2 \frac{Q_{C_0}^8}{Q_{C_T}^8} \right) - \left(\frac{Q_{C_1}^8}{Q_{C_T}^8} \log_2 \frac{Q_{C_1}^8}{Q_{C_T}^8} \right) = 0$$

The entropy calculation shows that node 8 is a leaf node as all the vectors of class 1. The DT has been updated as shown in Figure 4.12.

Calculations of node 9

For second child node 9 to parent node 4 with attribute value 'strong', the total node entropy can be calculated using the following parameters that are given by Equations A.34, A.35 and A.36.

$$Q_{C_0}^9 = Q_{C_0^1}^4 = 0$$

$$Q_{C_1}^9 = Q_{C_1^1}^4 = 3$$

$$Q_{C_T}^9 = Q_{C_T^1}^4 = 3$$

From Equation 2.2 the total entropy is equal to

$$E_T^8 = \left(-\frac{Q_{C_0}^9}{Q_{C_T}^9} \log_2 \frac{Q_{C_0}^9}{Q_{C_T}^9} \right) - \left(\frac{Q_{C_1}^9}{Q_{C_T}^9} \log_2 \frac{Q_{C_1}^9}{Q_{C_T}^9} \right) = 0$$

The DT has been updated as shown in Figure 4.13, where the entropy calculation shows that node 9 is a leaf node as all vectors are of class 1.

Node10

Next node is node 10 where its parent node 6 has chosen attribute 'humidity' as given by Equation A.43. The relation considered at this stage is

Outlook-->Temperature-->Humidity-->Wind

For node 10 where the value is high, the number of vectors is 3 as given by Equation A.39. The only attribute left is 'wind' and will be chosen for node 10. The DT beyond this point has been updated as shown in Figure 4.14.

NODE 11

For node 11 where the value is 'normal', there is one vector as given by Equation A.42. This node is a leaf node as the remaining vector is for class 1 and therefore no further calculations needed. The DT has been updated as shown in Figure 4.15.

NODE 12

For node 12, it is considered as leaf node. The value 'weak' on this node has three vectors distributed as follows

$$Q_{C_0}^{12} = 1$$

$$Q_{C_1}^{12} = 2$$

Referring to the values of class *don't play* (0) as in $Q_{C_0}^{12}$ and *play* (1) as in $Q_{C_1}^{12}$ the probability of class *play* is twice the occurrence of class *don't play*. In this case the decision is choosing the class with higher probability of occurrence. The complete decision tree is shown in Figure 4.16, where node 12 is updated as a leaf node.

The classification Rules generated by the HFTDT method are summarised as shown in Table A.1.

Table A.1 Classification (IF-THEN) rules model obtained by the HFTDT method for the completed DT shown in Figure 4.16.

1	Outlook "Sunny" AND Temperature "Hot" \xrightarrow{THEN} Class "Play"
2	Outlook "Sunny" AND Temperature "Mild" AND Humidity "High" AND Wind "Weak" \xrightarrow{THEN} Class "Play"
3	Outlook "Sunny" AND Temperature "Mild" AND Humidity "Normal" \xrightarrow{THEN} Class "Play"
4	Outlook "Sunny" AND Temperature "Cool" \xrightarrow{THEN} Class "Play"
5	Outlook "Overcast" \xrightarrow{THEN} Class "Play"
6	Outlook "Rainy" AND Wind "Weak" \xrightarrow{THEN} Class "Play"
7	Outlook "Rainy" AND Wind "Strong" \xrightarrow{THEN} Class "Don't play"

APPENDIX B

SOURCE CODE FUNCTIONS

B.1 HFTDT ALGORITHM SOURCE CODE FILES

The source code organised in three main folders

(1) src this folder contains the following three subfolders

- convert, contains convert.c
- header, contains global.h
- testbench, contains tb.c and struct.c

(2) data: this folder has four subfolders

- converted_files, contains the converted data and names files
- example, contains the dataset to be tested
- param, contains the generated parameters.c, parameters.h and logarithm table
- results, contains the results files

(3) bin: this folder contains the executable files

The following is a list of the source code functions.

B.2 SOURCE CODE FUNCTIONS LIST

Convert.c functions

1.	Function: read_word
	Purpose: read attributes and class of the input data vectors one by one
	Arguments: (1) Input data vectors (2) Pass a word
	Returns:

2.	Function: read_to_EOL
	Purpose: Used for the conversion to numerical data
	Argument:
	Returns:

3.	Function: Createlog
	Purpose: Create a lookup table to store the logarithm table
	Arguments: Number of classes
	Returns: Nothing

Parameters.c functions

1.	Function: Index_func
	Purpose: Generate key from input vector
	Argument: Input data vector
	Returns: A key value

2.	Function: updateHashTable
	Purpose: Store the generated keys in hash table and update its corresponding counter
	Arguments: Key from index function
	Returns: Nothing

3.	Function:	rev_index_func
	Purpose:	Generate output vector from ceratin key
	Argument:	(1) Key value from hash table (2) Output vector from key
	Returns:	Nothing
4.	Function:	ExpandNextNode
	Purpose:	Find the attribute needed in the next node in the list
	Argument:	The number of the attribute found to have the largest gain
	Returns:	True if an attribute has been found at this node
5.	Function:	AttributeEntropy
	Purpose:	Calculate the entropy for an attribute
	Arguments:	(1) start of the FT region for each attribute (2) end of the FT region for each attribute (3) the numer of the attribute to be considered (4) the number of vectors to be considered
	Returns:	The entropy of the data for the attribute
6.	Function:	NodeEntropy
	Purpose:	Calculate the entropy of all data being considered at a node
	Arguments:	(1) start of the FT region for each attribute (2) end of the FT region for each attribute (3) The number of vectors to be considered at this node (4) The entropy of the data being considered at this node
	Returns:	Nothing
7.	Function:	FindAttributeInstances
	Purpose:	Find the number of vectors for a certain pattern of attributes
	Arguments:	(1) start attribute value (for each attribute) (2) end attribute value (for each attribute) (3) Number of classes
	Returns:	The number of vectors

struct.c functions

1.	Function:	function_struct
	Purpose:	Build the decision tree
	Argument:	Numerical notation of input vector
	Returns:	Nothing

2.	Function:	function_test_data
	Purpose:	Classify new data
	Argument:	Input test vector
	Returns:	(6) 1 for correct classification (7) 0 for incorrect classification

APPENDIX C

TABLES RESULTS

C.1 DATA SETS SIMULATION RESULTS

(1) Nursery dataset simulation results compared to the software results.

Table C.1: Hardware simulation results for nursery dataset compared with software results.

Samples	Reverse index function run time (seconds)				
	Software	Virtex7	Kintex7	Artix7	Zynq
500	0.061919	0.007832	0.007628	0.010878	0.008479
1000	0.236946	0.035107	0.034192	0.048759	0.038005
1500	0.365639	0.057207	0.055716	0.079452	0.06193
2000	1.361605	0.110442	0.107563	0.153388	0.119559
2500	0.899293	0.172258	0.167767	0.23924	0.186478
3000	1.211905	0.242236	0.235921	0.336429	0.262232
3500	1.462808	0.302306	0.294425	0.419857	0.327261
4000	2.168851	0.461182	0.44916	0.640512	0.499252
4500	2.099037	0.447915	0.436239	0.622087	0.48489
5000	2.7444	0.588485	0.573144	0.817316	0.637063
5500	2.700256	0.586407	0.571121	0.814431	0.634815
6000	3.504098	0.76516	0.745213	1.062691	0.828323
6500	3.113556	0.770296	0.750215	1.069824	0.833883
7000	3.743051	0.909235	0.885533	1.262789	0.984291
7500	4.099405	0.993459	0.967562	1.379765	1.075468
8000	4.45182	1.083032	1.054799	1.504167	1.172435
8500	4.86446	1.184965	1.154075	1.645737	1.282783
8640	5.016686	1.223684	1.191785	1.699512	1.324698

(2) Agaricus-lepiota dataset simulation results compared to the software results

Table C.2: Hardware simulation results for agaricus-lepiota dataset compared with software results.

Samples	Reverse index function run time (seconds)				
	Software	Virtex7	Kintex7	Artix7	Zynq
200	0.13	0.011803	0.013803	0.017316	0.016768
400	0.25	0.023666	0.027675	0.034720	0.033620
600	0.37	0.035528	0.041547	0.052123	0.050472
800	0.51	0.047391	0.055419	0.069526	0.067324
1000	0.60	0.059253	0.069291	0.086930	0.084176
1200	0.72	0.071116	0.083163	0.104333	0.101028
1400	0.85	0.082978	0.097036	0.121736	0.117881
1600	1.02	0.094841	0.110908	0.139140	0.134733
1800	1.07	0.106703	0.124780	0.156543	0.151585
2000	1.18	0.118566	0.138652	0.173946	0.168437
2200	1.30	0.130428	0.152524	0.191350	0.185289
2400	1.41	0.142291	0.166396	0.208753	0.202141
2600	1.53	0.154153	0.180268	0.226156	0.218993
2800	2.28	0.166016	0.194140	0.243560	0.235845
3000	2.19	0.246097	0.287789	0.361046	0.349611
3200	2.32	0.413313	0.483332	0.606365	0.587160
3400	4.12	0.439153	0.513549	0.644275	0.623869
3600	5.07	0.553683	0.647482	0.812300	0.786572
3800	5.34	0.584451	0.683463	0.857441	0.830283
4000	6.34	0.615220	0.719444	0.902581	0.873994
4200	6.01	0.645989	0.755425	0.947721	0.917704
4400	6.23	0.676757	0.791407	0.992862	0.961415
4600	6.47	0.707526	0.827388	1.038002	1.005125
4800	7.47	0.876186	1.024621	1.285441	1.244728
5000	8.05	0.912702	1.067322	1.339012	1.296602
5200	8.75	0.949217	1.110024	1.392583	1.348477
5400	9.03	0.985733	1.152725	1.446155	1.400351
5416	9.01	0.985733	1.152725	1.446155	1.400351

(3) Chess dataset simulation results compared to the software results

Table C.3: Hardware simulation results for chess dataset compared with software results.

Samples	Reverse index function run time (seconds)				
	Software	Virtex7	Kintex7	Artix7	Zynq
100	0.04	0.008897	0.008617	0.015317	0.009379
200	0.12	0.029106	0.028191	0.05011	0.030685
300	0.2	0.054137	0.052436	0.093205	0.057073
400	0.28	0.072243	0.069972	0.124377	0.076162
500	0.34	0.090757	0.087905	0.156253	0.09568
600	0.45	0.130526	0.126424	0.224721	0.137606
700	0.54	0.160615	0.155568	0.276525	0.169328
800	0.67	0.212052	0.205388	0.365082	0.223555
900	0.99	0.261227	0.253018	0.449745	0.275398
1000	1.02	0.301737	0.292254	0.519488	0.318105
1100	1.15	0.361186	0.349836	0.621841	0.380779
1200	1.27	0.422522	0.409244	0.72744	0.445442
1300	1.52	0.496745	0.481135	0.855226	0.523691
1400	1.49	0.495865	0.480282	0.853712	0.522764
1500	1.59	0.531923	0.515207	0.915791	0.560778
1600	1.72	0.567408	0.549577	0.976885	0.598188
1700	1.99	0.657497	0.636834	1.131986	0.693163
1800	2.01	0.696196	0.674317	1.198613	0.733961
1900	2.43	0.856373	0.829461	1.474384	0.902828
2000	2.9	0.841316	0.814878	1.448461	0.886954
2100	2.44	0.883403	0.855642	1.520921	0.931324
2130	2.47	0.896029	0.867871	1.542659	0.944635

APPENDIX D

RESOURCES INCLUDED ON THE DVD

The resources included in the DVD are as follow:

(1) HFTDT source code

(2) Datasets: Include the three datasets nursery, agaricus-lepiota and chess.