Embedded Machine-Learning for Variable-Rate Fertiliser Systems: A Model-Driven approach to Precision Agriculture.

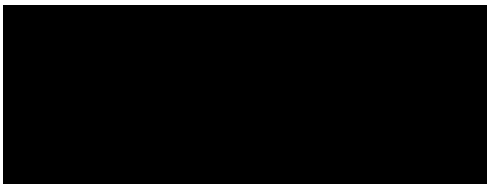
By
Joshua Stover

A thesis submitted for the degree of
Master of Science (Precision Agriculture)
of The University of New England
May 2019

DECLARATION

I certify that the substance of this thesis has not already been submitted for any degree and is not currently being submitted for any other degree.

I certify that to the best of my knowledge, any help received in preparing this thesis, and all sources used, have been acknowledged in this thesis.

....  .....

Acknowledgements

I would like to primarily thank Dr. Gregory Falzon for the valued support and supervision during the course of this work. Thank you to my supervisors Dr. Gregory Falzon and Prof. David Lamb. I would also thank the entire PARG research team, in particular Derek Schneider, James Bishop, Richard Crabbe, Arachchige Surantha Salgadoe, Jasmine Muir, and Andrew Robson for their support and specialised knowledge, much of the credit for this work belongs to them. Special acknowledgements to my entire family: Ann, Mark, Lucas, Mikaela, Kirsty, Hendrix and Kenji Stover for their continued support. I acknowledge the support I have received for this research through the provision of a Regional University Network Precision Agriculture Flagship (RUN PAF) scholarship.

Abstract

Efficient use of fertilisers, in particular the use of Nitrogen (N), is one of the rate-limiting factors in meeting global food production requirements. While N is a key driver in increasing crop yields, overuse can also lead to negative environmental and health impacts. It has been suggested that Variable-Rate Fertiliser (VRF) techniques may help to reduce excessive N applications. VRF seeks to spatially vary fertiliser input based on estimated crop requirements, however a major challenge in the operational deployment of VRF systems is the automated processing of large amounts of sensor data in real-time. Machine Learning (ML) algorithms have shown promise in their ability to process these large, high-velocity data streams, and to produce accurate predictions. The newly developed Fuzzy Boxes (FB) algorithm has been designed with VRF applications in mind, however no publicly available software implementation currently exists. Therefore, development of a prototype implementation of FB forms a component of this work. This thesis will also employ a Hardware-in-the-Loop (HWIL) testing methodology using a potential target device in order to simulate a real-world VRF deployment environment. By using this environment simulation, two existing ML algorithms (Artificial Neural Network (ANN) and Support Vector Machine (SVM)) can be compared against the prototype implementation of FB for applicability to VRF applications. It will be shown that all tested algorithms could potentially be suitable for high-speed VRF when measured on prediction time and various accuracy metrics. All algorithms achieved higher than 84.5% accuracy, with FB_{20} reaching 87.21%. Prediction times were highly varied; the fastest average predictor was an ANN (16.64 µs), while the slowest was FB_{20} (502.77 µs). All average prediction times were fast enough to achieve a spatial resolution of 31 mm when operating at 60 m/s, making all tested algorithms fast enough predictors for VRF applications.

List of Publications

- **Stover, J.**, Falzon, G., Jensen, T., Schroeder, B., Lamb, D. W. (2017). Hardware and Embedded Algorithms for Real Time Variable Rate Fertiliser Applications. *Presented at the International Tri-Conference for Precision Agriculture 2017*, Hamilton, New Zealand.

Contents

List of Tables

List of Figures

List of Equations

Chapter 1

Introduction

1.1 Overview

As the global human population grows, increases in food production capacity are inherently required to follow. Nitrogen fertiliser usage is a key driver in increasing crop efficiency and pasture yields in order to match demand (Erisman, Mark A. Sutton, Klimont and Winiwarter, 2008). Efficient usage of this diminishing and costly resource will be a top industry priority into the future, as well as one of the rate limiting factors in meeting food production requirements (Cavigelli, 2005). In addition, inefficient fertiliser usage has negative impacts on both the environment and human health (Townsend et al., 2003). Excess runoff from Nitrogen-based fertilisers is a major threat to the Great Barrier Reef (Schwartz, 2016), and has been linked to vast aquatic dead zones in the Gulf of Mexico (Biello, 2008). This has prompted the Australian government to set an 80% Nitrogen runoff reduction target as part of a long-term sustainability plan for the reef, which has already seen a reduction from the 2009 baseline, but requires continued innovation if it is to meet this target (Australian Government and Queensland Government, 2015).

Volatilisation of Nitrogen-based fertilisers also leads to the production of Nitrous Oxide (N_2O), a highly potent greenhouse gas (Ravishankara, Daniel and Portmann, 2009). In addition, excessive Nitrate levels in drinking water increases the risk of methaemoglobinaemia in infants less than three months old (Department of Health, 2011; McCasland, Trautmann and Wagenet, 1985) and at very high levels, may be associated with a increased risk of cancer (Espejo-Herrera et al., 2015; Ward et al., 2005). Reducing Nitrate levels in existing drinking water supplies can be a costly exercise for communities (Harter and Lund, 2012) but may be necessary to ensure public safety.

Precision Agriculture (PA) promises to address this challenge via the use of Variable-Rate Fertiliser (VRF) application strategies, along with the development

of smart sensors to exactly determine crop nutrient requirements. The combination of Remote Sensing (RS), Geographical Information Systems (GIS), and Machine Learning (ML) offers the possibility of intelligently applying variable rates of Nitrogen (N)-fertilisers "on-the-go", by using algorithms which can automatically determine the exact amount of nutrients required for each position in the crop.

A recent innovation in the field is the development of the Dynamic Aerial Survey (DAS) algorithm (Falzon, Lamb and Schneider, 2012), which was designed to batch process sensor measurement data from optical biomass sensors such as the Raptor ACS-225LR (Holland Scientific, 2011) mounted on low-flying aircraft.

DAS demonstrated that it is possible to segment a cropping area into variable-rate management zones and update biomass and nutrient-requirement estimates while the aircraft is in-flight and still collecting data. The estimated cost savings using the DAS approach was approximately AUD8000 compared to a uniform application method over a 110 ha field of wheat (Falzon et al., 2012). Such cost and resource savings are quite significant across vast areas of land under crop production, such as the estimated 22 348 000 ha dedicated to winter crop production alone in Australia over the 2016–17 season (Australian Bureau of Agricultural and Resource Economics, 2016).

Whilst the DAS algorithm was shown to be a useful contribution towards the realisation of a fully automated VRF system, there is still room for improvement in the regression component. This component has been designed to allow different ML classification and regression algorithms to be used depending on the task and site-specific conditions, however a comparison of algorithms for suitability to this application has not been performed (Falzon et al., 2012). One of the limitations in the testing methodology of DAS was the lack of a direct implementation of the described software model, which makes swapping software components more difficult, and therefore harder to test and compare. This is not uncommon in sectors such as agriculture, having a working prototype early is more meaningful to growers than a system designed for correctness.

A newly developed classification algorithm designed with real-time VRF applications in mind has been presented by Cohen as the Fuzzy Boxes (FB) algorithm (Cohen, 2016). This algorithm has the potential to be highly amenable to embedded systems, as it could easily be modified to provide a tuning parameter to balance prediction speed and prediction accuracy, and was shown in the original work to be capable of processing thousands of multi-band records per second. One major drawback to the FB algorithm is the memory complexity, particularly during the training phase. Memory space required grows exponentially relative to the number of input features. This makes FB suitable only for a low number of input

streams.

Both DAS and FB have been demonstrated to have separate beneficial uses within PA environments where fast and accurate prediction of crop requirements is desired. However, both algorithms have practical disadvantages which inhibit on-farm deployment. The current methodology of DAS requires a scale of computational resources only available in desktop machines, which is currently unrealistic for low-powered deployments such as on board light aircraft.

The FB algorithm has no available software implementation, meaning that in order to test and compare it with established ML algorithms, it needs to first be developed. This work makes use of techniques and processes taken from Model-Driven Software Development (MDSD) methodology principles for both the implementation of software and testing of algorithms.

1.2 Aims & Objectives

This thesis explores the suitability of multiple ML classification algorithms to the task of real-time estimation of crop N requirements from high-velocity input data streams like those produced by typical agricultural sensors.

Two main research questions will be explored: a Can ML classification algorithms provide accurate prediction of N fertiliser requirements?, andb Can MDSD techniques be useful for integrating ML into current PA applications? In order to address these questions and work towards realising a fully practical solution to automated VRF application, a dedicated and optimised hardware device is required which can perform accurate ML prediction and classification tasks in a timely manner using high-velocity data streams.

However, developing such a device requires timing and error metrics in order to assert the suitability of the core classification algorithm to the task of VRF.

The primary objective of this project is to benchmark and compare a selection of ML algorithms currently in use in the PA field. These algorithms will be compared on timing and accuracy performance within the scope of VRF applications which target low-powered computing systems. A recently published ML algorithm dubbed Fuzzy Boxes was implemented in the Python language for the first time and compared against two existing algorithms (Artificial Neural Network (ANN) and Support Vector Machine (SVM)) which have already seen success in PA.

In order to compare all of these algorithms, a secondary aim of implementing the FB algorithm as a programming library was required. The *fuzzyboxes* package was created using the Python language to meet this aim, and was subsequently modified, exposing the *depth* parameter in order to balance prediction time and

accuracy. This implementation was then compared against the existing Fast Artificial Neural Network (FANN)[1] and LIBSVM[2] libraries, which provide ANN and SVM models respectively, and have benefited from significant development effort over time. One of the primary objectives of this comparison is to show whether the FB algorithm has the ability to predict crop input requirements from sensor data streams in a timely manner.

Incorporating modern PA technologies such as RS, ML, and GIS with widely-used MDSD and Hardware-in-the-Loop (HWIL) techniques in order to develop an efficient, real-time VRF controller can pave the way for a model-driven approach to designing more robust and performant embedded PA hardware. The future development of dedicated VRF hardware controllers has the potential to provide environmental, health, and financial benefits to a range of stakeholders, and will be an attractive investment opportunity for the agricultural industry.

[1] https://web.archive.org/web/20190122061120/http://leenissen.dk/fann/wp/

[2] https://web.archive.org/web/20190106170055/https://www.csie.ntu.edu.tw/~cjlin/libsvm/

Chapter 2

Literature Review

In order to meet the agricultural demands of a growing population, Nitrogen (N) based fertilisers have been extensively used as a key driver in increasing the yields and production efficiency of pastoral crops (Erisman et al., 2008).

Efficient use of this diminishing and costly resource will be a top industry priority into the future, as well as one of the rate-limiting factors in meeting our global food production requirements (Cavigelli, 2005).

Precision Agriculture (PA) promises to address this challenge via the use of Variable-Rate Fertiliser (VRF) application strategies, along with the development of Remote Sensing (RS) technologies to accurately determine the nutrient requirements of the crop.

The combination of RS, Geographical Information Systems (GIS), Machine Learning (ML), and Model-Driven Software Development (MDSD) techniques offers the possibility of intelligently applying variable rates of N fertiliser to different points in a crop from a real time, "on-the-go" system.

2.1 Variable-Rate Technology

Adjusting the rate at which crop inputs are applied has been a consideration for at least 20 years (Clark and McGuckin, 1996; Kitchen, Hughes, Sudduth and Birrell, 1995), and patents have existed for variable rate fertiliser application systems since 1992 (Monson and Bauer, 1995).

Variable-Rate Technology (VRT) seeks to change the application rate of crop inputs "in response to spatially variable factors that affect the optimum application rate" (Sawyer, 1994), and has been investigated for use in multiple areas of PA.

A study conducted by Gonzalez-Dugo, Goldhamer, Zarco-Tejada and Fereres (2015) indicated that by adopting a variable-rate irrigation system, "over 15% of irrigation costs could be saved", and Kim, Evans and Iversen (2008) showed proof

of concept of a distributed wireless sensor network in order to continuously monitor and adjust irrigation levels.

By using a camera sensor to detect weed density and adjust output rate, herbicide savings of 30% and above were observed when compared to conventional uniform spraying of carrots (Dammer, 2016). Fungicide savings of up to 32.6% were also shown in cereal fields by combining VRT with an appropriate decision support system (in this case the online proPlant expert system) (Dammer, Karl-Heinz, Thöle, Volk and Hau, 2009).

VRT is currently being investigated for use in sugarcane fertiliser applications and has already been identified by Sugar Research Australia as one of the "promising opportunities" for improving Nitrogen use efficiency (Bell, 2014).

The literature surrounding the precision management of N in wheat crops has been thoroughly reviewed by Diacono, Rubino and Montemurro (2013), who found that the "improvements in yields, profitability or environmental quality still appear questionable". However, it was also concluded in the same review that it may be possible to increase profitability and reduce environmental impact by "integrating the real-time crop N status acquisition methods [...] with the soil and yield maps".

2.1.1 Variable-Rate Nitrogen

Nitrogen is a key macro-nutrient required for chlorophyll production in plants; crop efficiency and yields are directly affected by N status (Muñoz-Huerta et al., 2013). Growers will generally over-apply N fertiliser as a rational reaction to the uncertainty of soil nutrients and the potential impact that inadequate N has on yield and profitability (Thorburn, Park and Biggs, 2003). Excess application of N can result in runoff into waterways and into the Great Barrier Reef, where it is believed to be a major cause of crown-of-thorns starfish outbreaks (Fraser, Rohde and Silburn, 2017).

Variable-rate N has been shown to provide various economic and environmental benefits to growers by increasing use efficiency of the nutrient. Sensor-based Variable-Rate Application (VRA) outperformed producer-based N management for corn crops across 55 farms in the USA, where an increased partial profit of US$42/ha and reduced N use of 16 kg/ha was seen (Scharf et al., 2011).

In a review of precision N management in wheat, real-time sensing and fertiliser application were shown to increase profits by AU$(5–50) /ha compared to uniform applications, and an increase in N use efficiency by up to 368% compared to standard grower practices (Diacono et al., 2013).

Koch, Khosla, Frasier, Westfall and Inman (2004) also reported savings of US$(18.21–29.57) /ha and a reduced N fertiliser use of 6 %–46 % on corn by using VRA and site-specific management zones.

Variable-rate N has not always necessarily resulted in an improvement to crop quality or yield, as not all fields contain significantly different N management zones (Sawyer, 1994). Jørgensen and Jørgensen (2007) did not observe significant differences in protein content or crop yield between uniform and sensor based variable rate N application in wheat cropping using the Hydro-Precise N-Sensor System. A study involving 10 sites over 2 years in the Mid North and Yorke Peninsula of South Australia only found a mean increase in wheat grain yield of 0.8% by using variable-rate N (Mayfield and Trengove, 2009). The economic benefits of VRA in the Australian grain industry were studied and found to be overall positive, however some paddocks had losses of up to AU$28 /ha/yr, which was largely attributed to fertiliser management practises and within-yield variation (Robertson, Carberry and Brennan, 2009).

In order to automatically apply a varying rate of N fertiliser to a crop, it is first required that the current N levels and requirements are estimated.

2.1.2 N Requirement Estimation

Finding the ideal amount of N fertiliser to apply to a particular area of a crop or pasture is extremely difficult due to the many factors which influence the nutrient's efficiency. Some of these factors include: losses from $N \rightarrow NH_3$ volatilisation due to the breakdown of widely used urea fertiliser, natural leaching of water-soluble $N \rightarrow NO_3$ into soil, inter-annual and seasonal variability, and spatial variability in soil organic matter (Bragagnolo et al., 2013).

Methods currently used to obtain N requirements are primarily soil sampling and tissue testing, however these methods are time consuming and costly. Tissue testing is generally a destructive process, and incomplete combustion of the tissue sample can result in N loss and therefore incorrect measurements. This was shown by Unkovich et al. (2008) and was presented alongside other manual sampling based methods for estimating N fixation.

Queensland currently has mandatory testing of soil N (Queensland Government, 1994), with the goal of increasing the number of growers who base their fertiliser application levels on the results of regular N status testing.

Other N management strategies exist besides the traditional use of averaged regional levels, some from the Queensland sugar industry include the SIX EASY STEPS (Schroeder, Wood, Moody and Panitz, 2005; Wood, Schroeder, Hurney,

Salter and Panitz, 2008) and N replacement (Thorburn, Webster, Biggs et al., 2007; Thorburn, Webster and Biggs, 2008). Being manual processes, these strategies naturally require grower time and effort in order to be effective.

Technological alternatives for gathering N requirement data include the use of active optical reflectance sensors mounted on vehicles or low-level aircraft (Lamb, Schneider, Trotter, Schaefer and Yule, 2011; Lamb, Schneider and Stanley, 2014; Trotter, Lamb, Donald and Schneider, 2010), as well as the use of satellite imagery, aerial multi-spectral, and ground based field spectroscopy (Robson et al., 2016).

There is a strong relationship between chlorophyll and N content levels (Katz, Dougherty and Boucher, 1966), meaning that chlorophyll can be used as a proxy measurement for N level estimation. Jones et al. (2007) concluded that "Correlations of NDVI to biomass were approximately the same as the correlation of NDVI to chlorophyll yield", showing that remotely sensed Normalised Difference Vegetation Index (NDVI) data can be used as an indicator of N requirements.

Other indices which have been shown to be indicative of chlorophyll content, such as Red-Edge Normalised Difference Vegetation Index (RENDVI) and Red-Edge Position (REP) have also shown promise in estimating foliar N and other biochemical contents (Mutanga and Skidmore, 2007). For wider geographical areas, multispectral optical spaceborne platforms such as the Sentinel-2[1] and Landsat-8[2] satellites provide freely available data sets which can be used to generate the various optical indices.

Remote sensing devices targeting soil can also be used as proxy measurements for N requirements. One such device is the Geonics EM38-MK2 which measures apparent electrical conductivity (EC_a) (Geonics, Ltd., 2013) using conductivity and magnetic susceptibility components. A statistical analyses conducted by Eigenberg, Nienaber, Woodbury and Ferguson (2006) "supports an association of EC_a and Nitrogen Nitrate ($NO_3 - N$)", therefore it is possible to make use of EC_a as an additional data feature in the detection of N requirement.

In order to successfully develop a hardware device such as a dedicated VRF controller, it would be ideal to use small form factor and low powered computing hardware due to power consumption and physical space concerns. Such practical concerns might include issues such as unreliable power supplies in remote geographical areas, and inconvenient mounting locations (*e.g.* when mounted on a spray boom or aircraft undercarriage).

[1] `https://web.archive.org/web/20190111195127/https://sentinel.esa.int/web/sentinel/missions/sentinel-2`

[2] `https://web.archive.org/web/20190326184128/https://www.usgs.gov/land-resources/nli/landsat/landsat-8`

2.2 Low-powered Computing

This section will provide a review of some of the low-powered hardware technology which is currently available, including Field-Programmable Gate Array (FPGA) devices, Single-Board Computers (SBCs), and many-core devices. It will end with a review on the current state of running real-time systems under the Linux kernel as well as alternative operating systems.

There is a large range of computing hardware available today which has been designed with low power consumption and small form-factor in mind. Technology such as FPGA has eased the time and development effort required to create truly specialised hardware and embedded circuits, while the widespread availability and low cost of micro controllers such as the Atmel AVR[3] series has not only enabled hobbyists and beginners, but has also spawned the creation of low-cost SBCs and development boards such as the BeagleBone[4] and Arduino[5] series. The computational power available on a SBC has advanced even further since a team from University of Cambridge created the Raspberry Pi[6], which was designed as a low-cost platform to help teach programming to younger students (Ortmeyer, 2014).

2.2.1 FPGA Devices

FPGAs devices are a good choice for development of dedicated hardware and integrated circuits due to the ability to produce truly hardware-only implementations. When designing systems where verifiable run-time behaviour and low power usage are desired, FPGA based development may be beneficial. However, writing firmware for an FPGA target requires a working understanding of the hardware involved, and familiarity with one of the logic-based Hardware Description Languages (HDLs) such as VeriLog or VHSIC Hardware Description Language (VHDL). This difference in approach from the more common procedural languages such as C can cause development time and cost to be increased when developing for FPGA targets. This cost may be dramatically increased when the hardware is required to integrate with various peripherals such as volatile (*e.g.* RAM) and non-volatile (*e.g.* HDD/SD Card) storage, graphics, network and inter-board communications interfaces (*e.g.* USB, I2C, SPI).

[3] `https://web.archive.org/web/20190122185203/https://www.microchip.com/design-centers/8-bit/avr-mcus/`

[4] `https://web.archive.org/web/20190108044655/http://beagleboard.org/bone`

[5] `https://web.archive.org/web/20190110201933/https://www.arduino.cc/en/Main/Products`

[6] `https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md`

2.2.2 Single-Board Computers

One potential solution to speed up development time is to use off-the-shelf devices which integrate a microprocessor with commonly used peripheral devices onto a single board.

The idea behind integrating computing hardware onto a single PCB or development board has been around for over 40 years. In 1976, E & L Instruments manufactured the "Dyna-micro", which was later renamed to "Mini-Micro Designer", as the first true SBC (Romin, 2016; Hobson, 1978). This board was designed around a 2 MHz Intel C8080A microprocessor and 2 kB Random-Access Memory (RAM).

In July 2008, the non-profit organisation BeagleBoard.org was formed with the goal of creating a "low cost, open source community supported development platform". The result of which was the BeagleBoard SBC based around a Texas Instruments ARM Cortex-A8 System-on-Chip (SoC).

SBC development has only progressed since this time, and it is now easy to obtain a cheap, credit-card sized development board which integrates a SoC clocked at over 1 GHz, over 1 GB of RAM, and multiple I/O interfaces.

Current SBC development boards include the current BeagleBoard.org boards such as the BeagleBone Black/Blue/Green, Technologic Systems TS-7xxx series, and the Raspberry Pi series. Due to its popularity owing to its low cost and open source design, the Raspberry Pi SBCs have spawned the creation of many unofficial variants such as OrangePi[7] and BananaPi[8].

The Raspberry Pi SBC has been used as the base station in Wireless Sensor Networks (WSNs) for environmental monitoring (Ferdoush and Li, 2014), video signal processing for traffic monitoring (Kochlán, Hodo, echovi, Kapitulk and Jureka, 2014), motion-detection enabled video surveillance (Nguyen, Loan, Mao and Huh, 2015), and face recognition in noisy environments (Fernandes and Bala, 2015).

Barry et al. (2019) also used a Raspberry Pi 3 SBC for their Balloon Launched Imaging and Monitoring Platform (BLIMP) aerial monitoring system. This was a battery-powered design launched from a seaborne vessel, and was therefore constrained in terms of power requirements. This challenge is especially prominent with Unmanned Aerial Vehicle (UAV) based systems (Göktoan and Sukkarieh,

[7] `https://web.archive.org/web/20181005001841/http://www.orangepi.org/Docs/mainpage.html`

[8] `https://web.archive.org/web/20190317053813/http://wiki.banana-pi.org/Main_Page`

2015) due to weight limitations adding an additional constraint. A major challenge in using low-powered computing hardware is that it is generally very limited in computational resources such as memory and CPU power when compared with modern desktop and laptop hardware (Ibrahim, 2006). However, it has been shown that current low-powered hardware devices are capable of executing the often computationally intensive algorithms found in ML, and can do so at speeds which are acceptable for PA practices (Stover, Falzon, Jensen and Schroeder, 2017).

2.2.3 Microprocessors and micro-controllers

For applications requiring less computational power, such as dedicated actuator controllers or input processors, micro-controllers such as the Atmel AVR series have been a popular choice, and form the core of the Arduino product line (D'Ausilio, 2012).

Over the previous 15 years, the popularity of directly programming and using micro-controllers such as PIC and AVR has been declining in favour of the Arduino platform, which has seen a surge in popularity over the last decade due to its hobbyist-friendly approach (Figure 2.1). This has led to Arduino being a very common choice for prototyping low-powered systems both within academia and also the commercial sector.

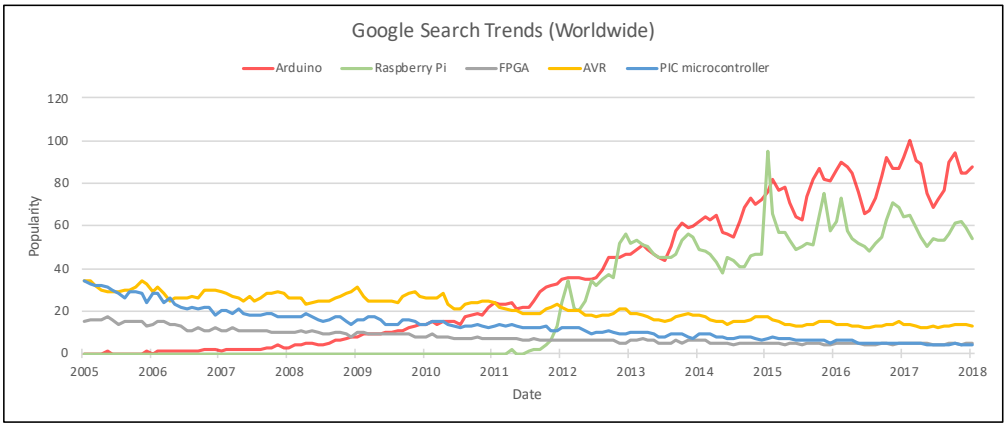


Figure 2.1: Google search trends for various development systems (trends.google.com, 2018)

Other popular low-powered micro controllers include the Texas Instruments MSP430 range, though these have not seen the widespread success that the Atmel micro controllers have, most likely due to the ease of development found in the modern Arduino platform.

DSP processors are specialised micro controllers designed for processing digital signal streams. They will often be built around a pipelined architecture and integrate components such as a Floating-point unit (FPU) to facilitate mathematical

calculations at low latency. A specialised hardware VRF controller which makes use of ML will most likely perform faster when used on a DSP processor instead of a general-purpose micro controller due to the many floating-point operations required.

Being specialised hardware means that software development effort is increased, as developers will need to learn the hardware architecture and software libraries for the target processor. This makes DSP processors a good choice for optimised hardware, but a poor choice for rapid prototyping.

2.2.4 Parallel and Many-core Devices

Parallelisation of algorithms has the potential to increase time performance dramatically (US Air Force Research Laboratory, 2015). Multicore CPU's are now extremely common consumer hardware, and many modern computing systems are equipped with dedicated graphics accelerators. Today's GPU's are designed around a Single Instruction Multiple Data (SIMD) architecture with a large number of individual cores (Nvidia Tesla V100 contains 5120 (NVIDIA Corporation, 2018)).

Since "Multicore processing units have become the dominant elements of modern computing systems" (Agathos, Papadogiannakis and Dimakopoulos, 2015), the usefulness of generalised parallel programming libraries has increased. One such library, OpenMP, was first released in October 1997 and targeted multiprocessing paradigms using the FORTRAN language. Since then, hardware support for various parallel and distributed systems has been added to OpenMP, including support for SIMD architectures as well as the recently developed Epiphany processors from Adapteva (Agathos et al., 2015).

The Parallella board is a credit-card sized SBC also developed by Adapteva, and is powered by two SoC IC's (Olofsson, Nordström and Ul-Abdin, 2014). The central SoC is a Xilinx Zynq-7000 AP SoC, which itself is an integration of a dual-core ARM Cortex-A9 microprocessor, programmable FPGA logic area, and various I/O peripherals and interfaces such as USB, Ethernet, and I^2C. An Epiphany III co-processor provides either 16 (Parallella-16) or 64 (Parallella-64) cores with a 32-bit Reduced instruction set computer (RISC) architecture. This co-processor enables the Parallella to perform parallel execution on many cores at low cost. A 1024-core version of the Epiphany architecture (Epiphany V) has also been developed, although it is not yet commercially available (Olofsson, Trogan, Raikhman and Adapteva, 2011; Olofsson, 2016).

2.2.5 Operating System (RTOS vs. Linux)

Low-powered development systems can be configured to operate as either a Real-Time Operating System (RTOS), or by making use of a general-purpose kernel such as Linux or one of the BSD kernels. These kernels provide a high level of support for various subsystems, devices and resources such as storage devices, network interfaces, and video output.

When the Linux kernel is running specific parts of its internal code, it may not be preempted (interrupted) by user processes. Due to this limitation, a so-called *soft* real-time environment is available for Linux, but not a *hard* real-time environment in which timing constraints can be guaranteed (Abeni, Goel, Krasic, Snow and Walpole, 2002). This makes the current mainline Linux unsuitable for timing-critical systems, however there is a patch set which aims to make the Linux kernel fully preemptible (Dietrich and Walker, 2005), and work by Sousa, Pereira and Tovar (2012) has been done to replace the First-In-First-Out (`SCHED_FIFO`) and Round Robin (`SCHED_RR`) real-time scheduling policies with a more advanced and performant policy (`SCHED_NPF_F`).

The NetBSD kernel has supported in-kernel preemption since version 5.0 was released in April 2009, which makes it a promising candidate for developing embedded, real-time systems (NetBSD Foundation, 2009).

An alternative to using a general-purpose operating system (OS) kernel is to use a RTOS, which generally consists of a minimal set of library functions allowing task scheduling, interrupt handling and process synchronisation, while still guaranteeing strict timing constraints where required.

FreeRTOS is one popular RTOS implementation and has been verified by Ferreira, Gherghina, He, Qin and Chin (2014) for "memory safety and functional correctness properties" using the HIP/SLEEK verification system.

While a high degree of timing precision is more easily attainable from a RTOS, a Linux kernel is still more likely to be used in most embedded hardware products due to the ease of development, deployment, and future upgrades. This is especially true when dealing with internet-connected and IoT devices.

During the early stages of embedded system development, a kernel such as Linux has the clear benefit of not requiring much "plumbing code" in order to achieve a minimal working system, and allows for rapid prototyping of embedded devices. This allows developers to take advantage of much of the current range of SBC devices without intimate knowledge of the board's components.

2.3 Machine Learning

ML algorithms have been widely discussed and used in PA over the last 20 years (Dimitriadis and Goumopoulos, 2008), and a wide variety are currently in use, such as Artificial Neural Networks (ANNs), and Support Vector Machines (SVMs). This section will present a review of ML algorithm use both in the context of embedded systems, and VRF development.

ANNs are a class of ML algorithms which have gained traction among researchers, industry professionals, and other stakeholders for use in precision agriculture since at least 1998 (Drummond, Joshi and Sudduth, 1998). They are inspired by the perceived "way that the brain performs computations", and use a series of interconnected artificial neurons as the basic building blocks of the algorithm, which are then organised into *layers* (Hagan, Demuth and Beale, 1996). These artificial neurons operate in a similar way to biological neurons, whereby weighted input signals are summed, and the output is activated once the sum exceeds a threshold. The transfer function of each neuron in an ANN is called the *activation function* and can be chosen to suit specific problems. A commonly used activation function for continuous input data is the sigmoid function (Equation 2.1) "since the function and its derivative are continuous" (Livingstone, 2008). In order to create a binary classifier from an ANN, a nonlinear function such as *binary step* (Equation 2.2) may be used on the last (output) layer neuron.

Training of an ANN involves varying the weights of each neuron in order to minimise the error between the ANN output and the labelled training data. Consequently, ANNs perform very well at tasks such as "pattern recognition, perception, and motor control" (Haykin, 1999), similar to the human brain.

$$f(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{2.1}$$

$$f(x) = \begin{cases} 0 & \text{if } x < 0.5 \\ 1 & \text{if } x \geq 0.5 \end{cases} \tag{2.2}$$

Recently, Deep Learning (DL) strategies consisting of multi-layer neural-networks have also seen operational successes in areas where large data sets are common such as audio and image classification (LeCun, Bengio and Hinton, 2015).

Although accurate models have been created for speech recognition (Yu and Deng, 2015; Hinton et al., 2012), and image classification (Simonyan and Zisserman, 2014), Deep Learning (DL) techniques require a high level of computational

resources not found in typical low-powered development platforms. Most DL models are trained on one or multiple GPU's (Chilimbi, Trishul M., Suzue, Yutaka, Apacible, Johnson and Kalyanaraman, Karthik, 2014) in order to reduce the training time.

ANNs have been shown to be successfully applicable to various PA applications such as automated irrigation control (Capraro, Patino, Tosetti and Schugurensky, 2008), inspection and grading of agricultural product (Brosnan and Sun, 2002), and wheat yield prediction (Pantazi, Moshou, Alexandridis, Whetton and Mouazen, 2016). Drummond et al. (1998) evaluated accuracy and generalisation ability of a number of feed-forward networks in 1998, and Yang, Prasher and Mehuys (1997) developed an ANN model in 1997 to estimate soil temperature which "executes faster than a comparable conceptual simulation model by several orders of magnitude".

SVM models have have also been used for multiple PA uses including real-time classifications. This ML algorithm involves mapping "the input space into a high-dimensional feature space" and attempting to construct an "Optimal separating hyperplane" between the input instances of different classes (Vapnik, 2013). This hyperplane is then used to predict the binary classification of unknown input values. Since instances can only lie on one of two sides of the hyperplane, SVMs are binary classifiers by default. A *kernel* based SVM is one which uses a *similarity function* (kernel) to help map the decision boundary. One of the most commonly used kernels is the Radial Basis Function (RBF).

Estimation of N content in the leaves of rice plants was demonstrated by Du et al. (2016) using SVM for accurate predictions. Qureshi et al. (2017) implemented an SVM based prediction model for automated fruit counting of mango; it was shown that ML models such as SVM could perform accurately and in real-time when incorporated into agricultural robotics systems.

With the current level of computing power available, it is now viable to perform classification of video and image data streams using ML techniques in real time (Nishihara et al., 2017; Shi et al., 2016). Weed and other pasture objects were able to be detected successfully and in real-time by using a novel ANN implementation (Sadgrove, Falzon, Miron and Lamb, 2017). More common convolutional neural networks have also been used for real-time discrimination between weeds and beets (Milioto, Lottes and Stachniss, 2017).

Studies have since shown that such real time classification can be of great use in the agricultural sector for field segmentation and zone mapping, as demonstrated in the Dynamic Aerial Survey (DAS) study which used an SVM model (Falzon et al., 2012).

2.3.1 Machine Learning in VRF

Use of ML models for VRF applications require fast performance of the predictor due to the potential high-speed of applicator vehicles such as aircraft. A VRF prediction model also needs to produce highly accurate predictions since the chemicals involved are hazardous to human health.

Poor algorithm performance potentially incurs economic and ecological losses, particularly for airborne delivery methods due to the high speed of aircraft, which is typically 55–65 m/s for fertiliser application with the widely used Air Tractor AT-502B (Air Tractor, 2017). Murray and Yule (2007) performed some early analysis of VRA techniques over six different production scenarios, and found there was a potential to increase annual cash surplus by AU$487/ha, 26% higher than blanket fertiliser applications.

Falzon et al. (2012) have presented the DAS algorithm which has been shown to be suitable for generating variable-rate prescription maps while in flight and in between collecting batches of data. One of the primary limitations with DAS is this reliance on batch processing, where an aircraft is required to complete a pass over the field to collect a batch of data. When aligning for the next data collection pass, the current data batch is processed using a regression or classification algorithm, and the fertiliser prescription map is then updated. Reliance on batch processing limits the applicability of this methodology to situations where the required data is either collected all at once (such as from a satellite platform), or is collected in stages broken up by gaps for processing time. This tends to rule out any delivery mechanisms besides aircraft, and becomes a major disadvantage for smaller scale applications and ground-based vehicles.

The architecture of DAS provides the ability to choose from a number of algorithms for the estimation, such as SVM and ANN models, as well as the ability to change the field segmentation algorithm (linear cutting, clustering, etc.) depending on the suitability to the crop type.

2.3.2 Machine Learning in Embedded Systems

Currently, there are a large number of existing software libraries implementing the different ML algorithms. Although there a a couple of independently created libraries in the hobbyist area and on GitHub, there are no thoroughly-tested and production-ready ML libraries for micro-controllers available. There is however, a range of ML libraries which are suitable for low-powered SBC platforms such as Raspberry Pi and BeagleBone Black. Some of these libraries, such as CaffePresso, were designed to run on accelerator-based platforms which utilise GPU, DSP, or

FPGA acceleration (Hegde, Siddhartha, Ramasamy and Kapre, 2016) which is generally unavailable on SBCs.

Other libraries such as Fast Artificial Neural Network (FANN) were designed to be cross-platform and have low training and prediction times (Nissen, 2003). A larger list of ML algorithm packages suitable for low powered systems is shown below.

- CaffePresso — An optimised library for deep learning on embedded accelerator-based platforms (Hegde et al., 2016)
- FANN — Implementation of a fast artificial neural network library (Nissen, 2003)
- LIBSVM — A library for support vector machines (Chang and Lin, 2011)
- tiny-dnn — header only, dependency-free deep learning framework in C++14 (Nomi, 2012)
- PyBrain — the Python Machine Learning Library (Schaul et al., 2010)
- scikit-learn — Machine Learning in Python (Pedregosa et al., 2011)
- Python-ELM — Extreme Learning Machine implementation in Python (Lambert, 2013)
- ELM-C-- — Extreme Learning Machine C++ library (Vladislavs, 2013)
- Fido — A lightweight C++ machine learning library for embedded electronics and robotics (Truell and Gruenstein, 2016)

2.4 Software Methodology

This section will discuss the use and limitations of software methodologies currently being used in the agricultural field, including robotics and hardware development. It will introduce software methodologies which have seen success in other fields, such as MDSD and Finite-State Machine (FSM) based modelling. Finally, a methodology for testing hardware systems by simulating environmental inputs is presented in 2.4.6.

2.4.1 Agricultural Robotics

It has been shown by Pedersen, Fountas, Have and Blackmore (2006) that the use of robotics and automated systems in agriculture "may reduce labour costs and restrictions on the number of daily working hours significantly".

Commercial products are already available which perform specific tasks such as the *RowBot* for automated seeding and N application (Rowbot Systems, 2017), and the *ASTERIX* project for automated herbicide application, which aims to

reduce herbicide usage by approximately 95% (The European Commision, 2017).

Small scale robotics have even made it into the home-garden and amateur farming market, with products such as FarmBot providing an open-source, web-integrated, automated farming machine which could potentially be built to support large areas (Aronson, 2013).

Other companies such as SwarmFarm (SWARM FARM, 2018) claim to be creating smaller robotic devices which are "smarter" than the current large-form field machines. As is typical for commercial enterprises, research methodology is not revealed to the public, therefore most of the claims made by these organisations are unverifiable short of purchasing the product and conducting manual tests. This makes it extremely difficult to locate technical information about the designs, and therefore makes analysing, verifying, and validating these products much more difficult, if not impossible.

Since the source code for these products is generally proprietary and not publicly available, any new entrant into the agricultural robotics industry will also need to build their own software from the ground up instead of relying on open-source agricultural robotics libraries. This slows down progress in furthering the state of the art in PA robotics.

Software development for agricultural robotics often involves a simulation of the hardware in order to verify designs and to observe automata behaviour. Shamshiri et al. (2018) has reviewed many of the current simulation packages such as Webots, V-REP, Gazebo, Actin, and ARGoS, concluding that V-REP is the most fully-featured while Gazebo contained a smaller set of features but provided a simpler interface.

2.4.2 Limitations in current development techniques

The traditional view of designing embedded systems was to treat it as a hardware-software co-design problem (Chiodo et al., 1994; Wolf, 1994), where the hardware and software are designed together in a process known as "Co-synthesis" (Ernst, Henkel and Benner, 1993). This was generally performed using FSM designs and calculating the exact computational costs of all code paths (Chiodo et al., 1994). This used to be a more viable approach; the limited availability of computational power in microprocessors meant that the task complexity was lower and the programming was primarily done in assembly (Ernst et al., 1993), however as computational power grew, more code was being written in higher level languages such as C. This, along with increasing design complexity, functionality demands, and time-to-market pressures in the embedded space, meant that changes in design and

development methodologies were required in order to stay economically feasible. In particular, a drive towards a methodology which will "work at all abstraction levels ... with guaranteed correctness", and "favour reuse" (Sangiovanni-Vincentelli and Martin, 2001) was desired.

As the uptake of robotics and automata grows in precision agriculture and an increasing number of developer-hours is spent in writing the corresponding software, it becomes increasingly desirable for the existence of an open, widely available, and easy to use software framework. Each unique problem in PA has so far required specific software solutions and are often designed from the ground up. Both hardware and software requirements are re-designed for each solution, and there is very little, if any, code re-use between research teams. To increase the robustness and design time of PA solutions, it would be ideal to have a common software framework which PA solutions can be built from.

2.4.3 Model-Driven Software Development

Just as programming languages and paradigms are abstractions of a physical computing environment, models can be created which are instead abstractions of the software behaviour. MDSD is a software development framework which originated as a reframing by Maes, Dedene et al. (1996) of the information systems architecture described by Zachman (1987), and seeks to separate "domain knowledge specifications from functionality specifications" (Snoeck and Dedene, 1997). This approach is field-agnostic and therefore works within any problem domain, though it has seen the most success in the automobile industry in recent years due to the industry's focus on safety and critical-failure mitigation.

This is sometimes worded as "abstractions of the solution space, rather than abstractions of the problem space", and has led to the development of Computer-Assisted Software Engineering (CASE) tools. These are applications which aid in the development of software by providing a visual representation of complex data structures, behaviour states, and data flow of a system (Schmidt, 2006).

CASE tools are currently being used in the development of robotics software, such as in the annual RoboCup competition for soccer-playing automata where rapid changes to agent behaviour may be required *e.g.* "during half-time because of some unexpected opponent strategy" (Topalidou-Kyniazopoulou, Spanoudakis and Lagoudakis, 2013).

By making use of these abstractions and CASE tools as drivers in the development process, a MDSD methodology provides the ability to perform formal verification of program behaviour, as demonstrated by Estivill-Castro, Hexel and

Rosenblueth (2012), as well as automatic translation from a modelling language such as UPPAAL to executable code (Stover, 2015).

2.4.4 Finite-State Machines

Booth (1967) describes in the book *Sequential Machines and Automata Theory*, a system for modelling sequential automata as state diagrams which has become useful for designing and representing behaviour models for robotic applications.

There have been various extensions to this model, but since *Statecharts* were described by Harel's seminal work (Harel, 1987), FSMs have "become ubiquitous models of system behaviour" (Estivill-Castro and Rosenblueth, 2011).

FSM designs are often used when employing a MDSD methodology, as they provide a high-level of system abstraction (Wood, Akehurst, Uzenkov, Howells and McDonald-Maier, 2008). It has been suggested that future software for the rapidly increasing number of Internet-of-Things (IoT) devices is likely to be based on state machines (Bryce and Kuhn, 2014).

One of the drawbacks to traditional UML state machine designs is that transitions between states are defined as *events* which may come into the system in a non-deterministic order, hindering the ability to reason about the behaviour over the system. The *run-to-completion* execution model described by Selic (1996) can be used to somewhat alleviate the issue by forcing the state to finish execution before transitioning.

2.4.5 Logic-Labelled Finite-State Machines

Software quality is a major concern for embedded systems as a lack of quality can cause serious failures and security risks in critical hardware (Sametinger, Rozenblit, Lysecky and Ott, 2015). This has led to an increase in academic interest in run-time verification and health management of embedded software (Heffernan, MacNamee and Fogarty, 2014; Srivastava and Schumann, 2013).

The Logic-Labelled Finite-State Machine (LLSFM) model was developed in order to minimise the chances of human error when implementing software based on a FSM model, whilst also introducing a more complete and formal reasoning about FSM models (Billington, Estivill-Castro, Hexel and Rock, 2010a).

When used in conjunction with an object-oriented shared memory component such as `gusimplewhiteboard`, multiple LLSFMs can communicate with each other and the external environment in a safe a predictable way (Estivill-Castro and Hexel, 2014). The operation behind LLSFMs is based on the theory of Plausible Logic (PL) (Billington, Estivill-Castro, Hexel and Rock, 2010b).

A LLSFM is a variant of FSM which allows a programmer to produce behaviour models which are easily compiled, have deterministic execution, and are easy to understand for existing programmers. The internal sections of states can contain any valid C or C++ code, however control structures such as `for`, `while`, and `if` should never be used, as the control flow should be provided by the LLSFM structure.

The major differences between a LLSFM and Harel's model is that machines are not waiting in a state for events, but instead the machines drive the execution by actively checking the outgoing transition conditions. This is in contrast to the *run-to-completion* execution strategy (Estivill-Castro, Hexel and Regalado, 2016). These conditions are generally standard C/C++ Boolean expressions or "queries to an expert system" (Estivill-Castro and Hexel, 2014).

Formally, an LLSFM consists of a set `S` of states, an initial state `s0 ∈ S`, and a transition table `T : S x E → S`. The set `E` consists of logic expressions, whereas the UML approach to FSMs would have events in the transition table (Estivill-Castro and Hexel, 2014). An example taken from a pacemaker controller is shown in Figure 2.2.

For multi-machine systems, `clfsm` is used as an execution scheduler and whiteboard client. It can run machines saved in the `.machine` format, and execution control (start/stop/pause) is done through the whiteboard messaging system.

UPPAAL is a "suite for automatic verification of safety and bounded liveness properties of real-time systems modelled as networks of timed automata" (Bengtsson, Larsen, Larsson, Pettersson and Yi, 1996).

Stover (2015) has shown that the individual components of a UPPAAL model can be directly translated into an executable LLSFM, opening the possibility to fully automatic translation from verifiable software models to executable code.

By combining a set of LLSFMs with the `gusimplewhiteboard` library, a developer can create a fully executable, model-based design where all inter-machine communications happen through the common `gusimplewhiteboard` instance.

2.4.6 Hardware-in-the-loop testing

The architecture of LLSFMs encourages breaking down the software into subsystem components, which in turn allows for the creation of simulated input signals in order to verify the behaviour of each of the models at run-time.

This method of simulating environmental inputs is called Hardware-in-the-Loop (HWIL) testing and is done by creating "tester" programs to generate the

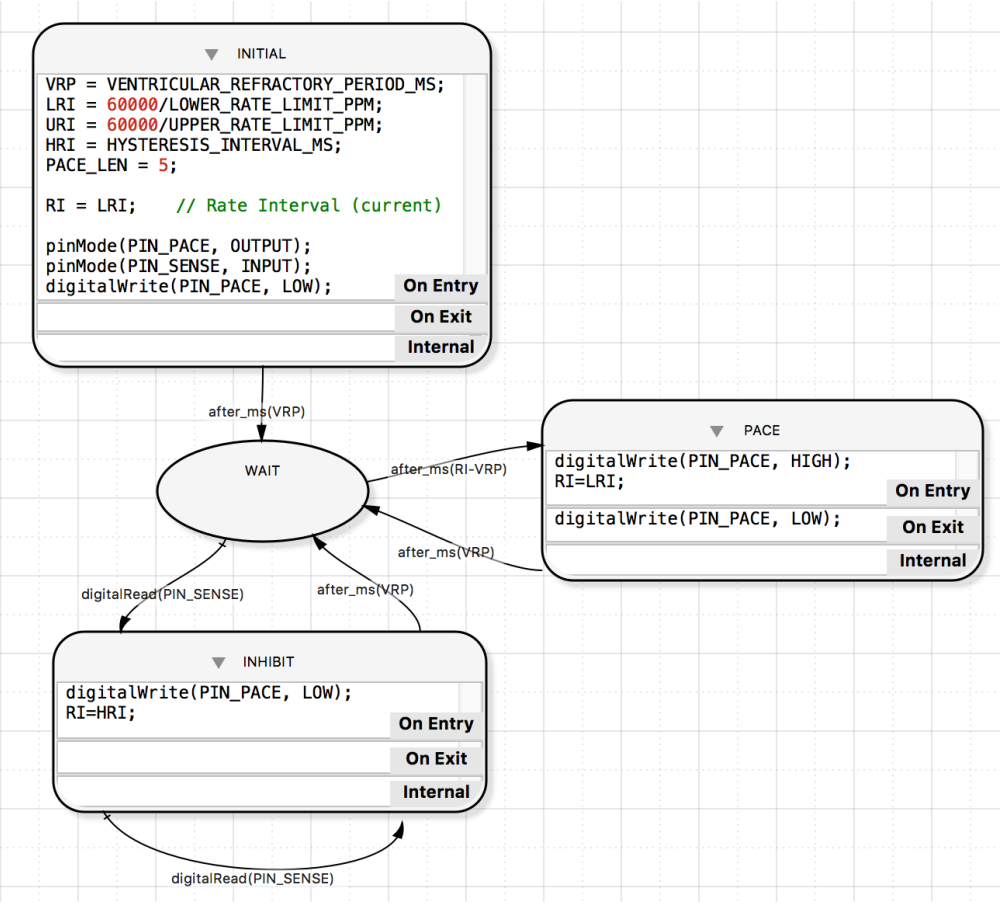


Figure 2.2: Pacemaker VVI controller as LLFSM

inputs and verify the output behaviour of the System Under Test (SUT) (Figure 2.3,2.4). This process is made much easier by making use of LLSFMs and the `gusimplewhiteboard` communications library.

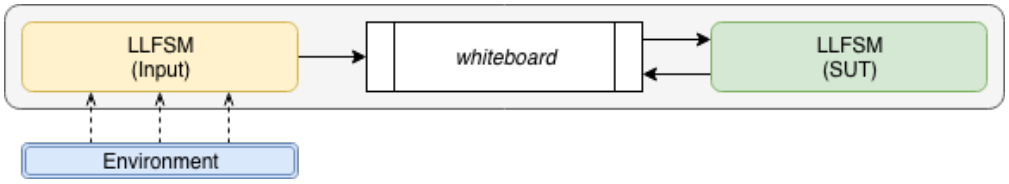


Figure 2.3: Typical LLFSM/whiteboard usage

In this scenario, the SUT has no knowledge that it is in a simulated environment, and will behave as if it was interfaced to the real environment. When these tests are derived from the software requirements, they can form the groundwork for a Test-Driven Development (TDD) methodology (Estivill-Castro and Hexel, 2017; Estivill-Castro, Hexel and Stover, 2015a; Estivill-Castro, Hexel and Stover, 2015b). Such testing software can be incorporated into Continuous Integration (CI) environments, providing immediate feedback to developers about the results of behaviour tests. It has been demonstrated by Estivill-Castro and Hexel (2016) that LLSFMs are "very fruitful for model-driven software development as they

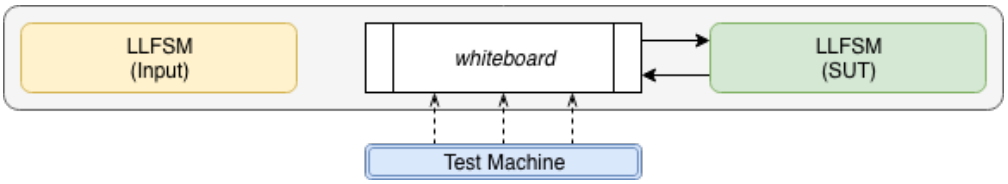


Figure 2.4: Testing LLFSM with simulated inputs

constitute executable models".

Chapter 3

Methodology

The work presented in this thesis builds off the work done by Falzon et al. (2012), which showed that the Dynamic Aerial Survey (DAS) algorithm is suitable for applying varying levels of crop inputs such as Nitrogen (N) based on remote sensing data streams.

Adapting an algorithm such as DAS to operate in a real-time, dedicated hardware environment contains some challenges that must be overcome. Resource limitations such as power consumption, computational power, and memory resources are all common in the embedded hardware space. Practical limitations such as on-vehicle deployment and access to sensor streams is another challenge that would need to be overcome in order to fully realise a dedicated hardware Variable-Rate Fertiliser (VRF) controller.

A number of segmentation algorithms are compared in the original paper, however to further test the DAS algorithm for its suitability to the task of VRF application, a number of different classification models also need to be tested.

Besides prediction accuracy metrics, it is also important to test and compare the timing behaviour of these models in order to eliminate any models which require too much time to predict.

Training performance was also measured for this work. Although this metric is less relevant for *application* of a model, it is a useful measurement for any future work which attempts to combine stream-learning techniques with DAS.

3.1 Data Set

For this thesis, a real-world data set was used for model training and testing. This data was collected by Derek Schneider and the University of New England Precision Agriculture Research Group team for the initial development and analysis of the Fuzzy Boxes (FB) algorithm (Cohen, 2016).

The collected data consists of measurements taken from a quad bike platform fitted with two commercially available sensors as part of trials to asses wheat (*Triticum aestivum var. Gregory*) vigour. The wheat field consisted of 18Ha of Z30 stage wheat. Applying N fertiliser at this stage has been shown to increase biomass yield (Bowden et al., 2008).

Red Normalised Difference Vegetation Index (RNDVI) (Equation 3.1) data measures the difference between red light (780 nm) reflectance and near-infrared (647 nm) light reflectance (Rouse, Haas, Schell and Deering, 1974), and was collected by a Holland Scientific Crop Circle™ ACS-210 active canopy sensor (*Crop Circle ACS-210 Product Sheet* 2004). Apparent soil conductivity (EC_a) (Rhoades, Raats and Prather, 1976) was also measured using a Geonics EM38-MK2 operating in vertical dipole mode (Geonics, Ltd., 2013). As such, the data is labelled as RNDVI and EM38 within the data set and on many of the generated output graphics. All data instances were georeferenced using a Trimble TSCe Ranger data logger with attached Differential Global Positioning System (DGPS).

$$\mathbf{RNDVI} = \frac{NIR_{(780)} - R_{(647)}}{NIR_{(780)} + R_{(647)}} \tag{3.1}$$

An expert agronomist was on-site during data collection, and reviewed sensor records to determine the thresholds at which fertiliser should be applied to the crop. These thresholds were then used to develop a binary classification for each recorded point in the field. A total of 2399 measurements were recorded and classified; a statistical summary of the data is provided in Table 3.1. Once the data had been reviewed, there was found to be an imbalance in the classifications labels. This was expected as the crop was operating as usual and was not controlled to produce balanced data. 699 locations were classified as requiring fertiliser (positive class), and 1700 were classified as not requiring fertiliser (negative class). As can be seen from Figure 3.1, the classifications are spread fairly evenly along the RNDVI feature, but were somewhat clustered along the range of EM38 measurements, where a higher EC_a value indicates a lower probability of requiring fertiliser. This clustering around a single sensor's threshold shows potential for fast and computationally cheap prediction of fertiliser requirements by using Machine Learning (ML) methods.

An up-sampling strategy is often used to balance the number of instances in the positive and negative classes by reusing instances from the smallest set where needed (Ganganwar, 2012). Since the positive and negative classes of this data set were unbalanced (699:1700), up-sampling was employed during ML classification testing. All values were also min-max normalised to the [-1:1] range in order to

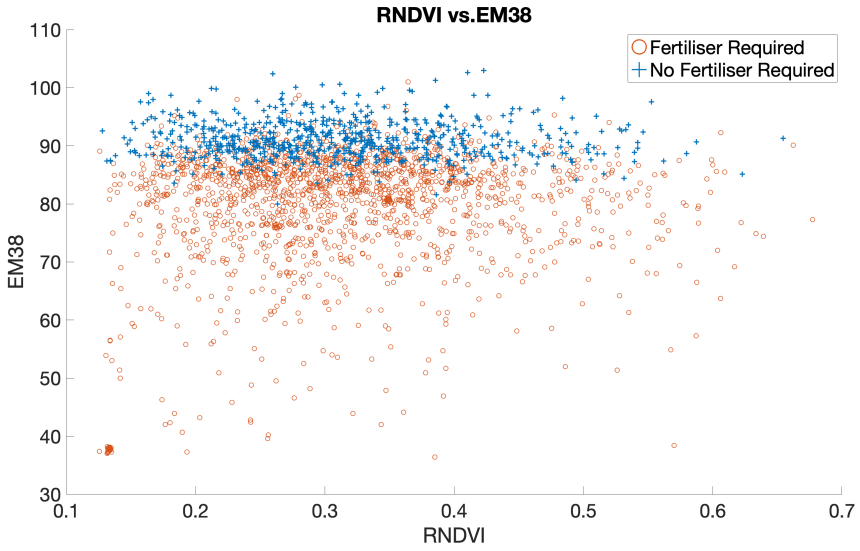


Figure 3.1: Visualisation of the data set

	RNDVI	EM38
Minimum	0.1256	36.40
1st Quartile	0.2450	79.90
Median	0.3044	85.50
Mean	0.3138	83.11
3rd Quartile	0.3718	89.40
Maximum	0.7118	103.00

Table 3.1: Statistical summary of the data set

"neutralise the effect of different quantitative features being measured on different scales" (Flach, 2012).

3.2 Existing Algorithms

Many applications of Precision Agriculture (PA) have made use of ML algorithms in order to provide accurate predictions from high-bandwidth data streams.

Two existing families of ML algorithms which are currently in use for PA are Support Vector Machines (SVMs), and Artificial Neural Networks (ANNs). A newly described algorithm called FB (Cohen, 2016) was implemented in Python, and then compared against these existing algorithms.

One of the common ML algorithms currently being used is the SVM. A particular type of SVM called C-Support Vector Classifier (C-SVC) is a Radial Basis Function (RBF) kernel based classifier which exposes the C and γ kernel parameters. The model used for this work is a C-SVC implemented in the LIBSVM library.

The parameters used for this model were found by conducting a 10-fold Cross Validation (10CV) grid search over the hyperparameter (C & γ) space. Parameter search results for SVM are shown in Table 4.2.

Another widely used class of ML algorithms are ANNs. There are many varieties of ANN depending on the number of hidden layers, number of neurons per layer, and which neurons are connected. The ANN model used in this thesis is a fully-connected back-propagation neural network implemented using the Fast Artificial Neural Network (FANN) library. The parameters for this model can be seen in 4.3 and were also found via a 10CV grid search. The hidden layer neurons used a standard sigmoid activation function (Equation 2.1), while the output layer used a binary step activation function at 0.5 (Equation 2.2).

3.3 Fuzzy Boxes Algorithm

The Fuzzy Boxes (FB) algorithm is a binary classification algorithm described in 2016 by Cohen (Cohen, 2016).

It has been designed to assist in easing the challenges of a low-power and low-resource computing environment, and the original work specifically addressed VRF applications. Since there was no publicly available implementation of FB, one was created in the Python language as part of this thesis (Appendix A).

FB models are trained by reading data from each input stream (feature) and dividing the value range into bins. Split points for each bin are calculated in one of two ways: either based on the percentile of instances which fall into each bin, or by evenly splitting the value range. The split-point calculation algorithm is described in Algorithm 1. This has been called *percentile* and *even* splitting respectively, and is controlled in the code with the `method` parameter. The splitting process is repeated for each number of desired bins from $S_{\max}$ to $S_{\min}$, where S is the number of splits along each feature. The total ratio of positive to negative training instances is saved into the model and can be accessed as the `prior_ratio` property if desired.

Once all split points have been found, the model will consist of $S_{\max} - S_{\min} + 1$ layers each containing n-dimensional boxes, where n is the number of input features. Stored within each bin is a rational number to keep track of how many training instances fell into the box, and what proportion where classified as positive.

Figure 3.2 shows examples of a 2-dimensional FB model at $S = 10$ with both even and percentile splitting methods using the data described in (Section 3.1). The darker boxes are cells in the feature space where a higher percentage of training

values were in the positive class, whereas the lighter boxes are cells with a lower percentage of positive-classed instances. A full FB model consists of one of these grid layers for each value of S.

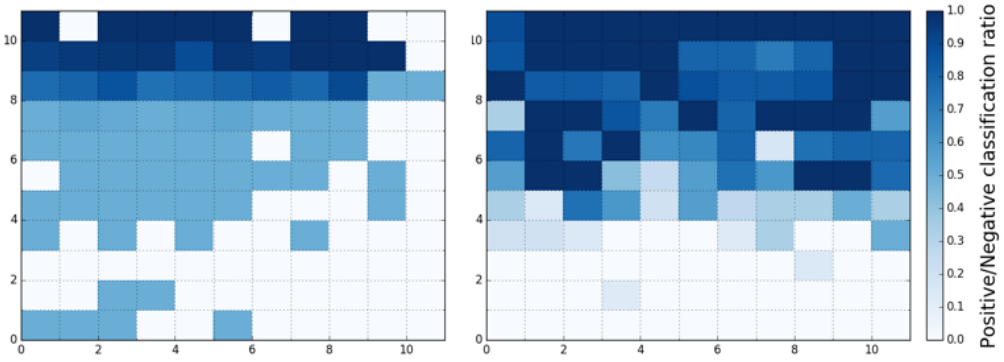


Figure 3.2: Fuzzy Boxes model showing even (left) and percentile (right) splitting methods

Once all of the training data has been placed into a bin and the positive/negative ratio for each bin has been determined, the model is considered to be trained. In order to save and load the model, the ratio within each box needs to be recorded.

Since each value of S makes up one "layer", one possible improvement to this algorithm would be to "flatten" all the layers down into a single n-dimensional array. This would provide faster prediction times since the "box position" of each incoming instance would only need to be found once. Currently, this process needs to be repeated for every value of S.

3.4 Fuzzy Boxes Implementation

The development of FB was broken down into separate modules for the main model, input features and instance values. Modules were also created for the graphical outputs and testing results in order to keep the main implementation separate from the testing and visualisation code.

The classes which are considered central to the FB algorithm's operation are described in Table 3.2. Classes developed for testing the algorithm are listed in Table 3.3, and any extra modules which are not part of the library, but were used during developed are listed in Table 3.4.

In the future, an optimised and highly portable software library may be desired for FB. The low-level nature of the C language means that embedding C libraries into other language code bases is often much easier than attempting the opposite. At the expense of greater development effort, a C library also provides greater potential for optimisation of run-time performance. This may allow for

Algorithm 1 Fuzzy Boxes split-point algorithm

Require: F, List of features
Require: $S_{\min}$, Minimum number of split points
Require: $S_{\max}$, Maximum number of split points
Require: M, Method of calculating split points

for all $f \in$ F **do**
  $S \leftarrow S_{\min}$
  while $S \leq S_{\max}$ **do**
    $i \leftarrow 1$
    while $i \leq S + 1$ **do**
      if $M == even$ **then**
        $f.\text{splits}[S][i] = f.\min + (i+1) \times (f.\max - f.\min)/(S+1)$
      else if $M == percentile$ **then**
        $f.\text{splits}[S][i] = \text{numpy.percentile}(f.\text{values}, (i+1)/(S+1) * 100)$
      end if
      $i \leftarrow i + 1$
      $S \leftarrow S + 1$
    end while
  end while
end for

ML algorithms such as FB to work on devices which may not have the resources to run an interpreted language like Python.

Since it was still unknown as to whether FB would produce timing results comparable to the existing ANN and SVM algorithms, it was decided that the initial code for FB would be written in the Python language. This helped to greatly decrease the development time of both the FB library and the code used to compare to other algorithms, while maintaining portability between most operating systems.

Loose coupling between modules was desired in order to ease transition from Python to C in the future, as there are many useful high-level abstractions available in Python which must be implemented manually in C. This is the primary reason that FuzzyBoxInstanceValue inherits from `float` and why custom classes were created for FuzzyBoxInstance and FuzzyBoxRatio (*i.e.* these classes will need to be implemented as `struct`'s in C and it will make the re-implementation easier if they are already loosely-coupled). The coupling which resulted is shown in Figure 3.3.

The high-level modules were then designed as UML (Figure 3.4) before finally being implemented in Python. Basic regression tests were also designed at this stage to help detect errors around loading data from comma-separated values (CSV) files and manipulating data internally (Figure 3.5). As the algorithm has no other implementation to test against, a small (10 instance) set of synthetic data

Fuzzy Boxes core classes	
FuzzyBoxModel	Main model class. This is responsible for saving/loading the model and also contains the train/test functions.
FuzzyBoxFeature	Object containing the split points for each box N of a single input feature. Also may contain a list of FuzzyBoxInstance's holding the training data.
FuzzyBoxRatio	Simple object describing ratio of positive / negative class ratio for a single box.
FuzzyBoxInstance	Simple object containing values for each feature and a classification (training data only). Operates similar to a dataframe row in R.
FuzzyBoxInstanceValue	Derived from float and so can be treated in code as such. Also stores a box position array (one entry for each value of S).

Table 3.2: Description of the core classes making up the *fuzzyboxes* Python library

Fuzzy Boxes testing classes	
Results	Error statistics for model classification testing, as well as custom timer objects for performance testing.

Table 3.3: Description of classes related to *fuzzyboxes*

was created and the algorithm was worked through by hand up to $S = 3$ to create results which were compared to the model values. These regression tests were primarily used to detect unexpected changes in the output due to programming errors, and were found to be very useful over the course of this work.

3.5 Algorithm Testing

Each of the ML algorithms were tested and compared on multiple error metrics: *Accuracy* (Equation 3.2), *Precision* (Equation 3.3), $F1_{score}$ (Equation 3.4), and G_{mean} (Equation 3.5). Note that the formulae presented here are for analysing binary classifiers, while the implementation of the `Results` class was created to be compatible with multiclass testing. Implementation can be found in the file `results.py` (Appendix A). *Accuracy* and *Precision* were included since they are standard error metrics when comparing ML algorithms, while $F1_{score}$ and G_{mean} was included in order to consider both precision and recall metrics. Other metrics recorded by `Results` include *true-positive rate* (TPR), *true-negative rate* (TNR), *false-positive rate* (FPR), and *false-negative rate* (FNR). TPR and TNR measurements were collected for calculation of the G_{mean} value. FPR and FNR were

Additional modules	
graphics	Additional functions for generating scatter plots and heat-maps from 2D data sets.

Table 3.4: Description of any extra modules related to development but not part of the *fuzzyboxes* library

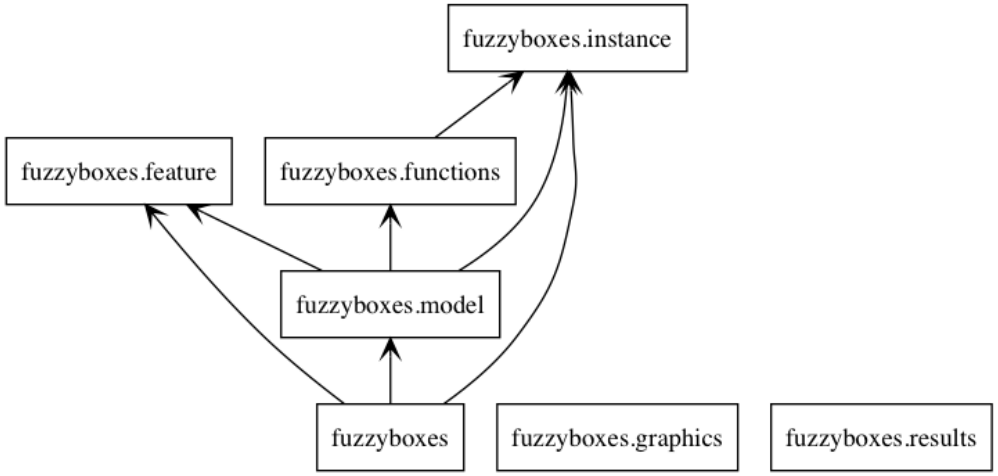


Figure 3.3: Fuzzy Boxes Python modules

not used, but were collected regardless due to ease of implementation. Since this testing was done with an automated delivery system for potentially hazardous chemicals in mind, it was important to consider the *true negative* class (doesn't need fertiliser) as important as the *true positive* class, if not more so. For this reason, the geometric mean of the recall statistic for every classification was used. When the classifier under test has binary classes, this is called *gmean1*.

$$Accuracy = \frac{TP + TN}{P + N} \tag{3.2}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

$$F_1 - Score = \frac{2TP}{2TP + FP + FN} \tag{3.4}$$

$$G_{mean} = \sqrt{TPR \times TNR} \tag{3.5}$$

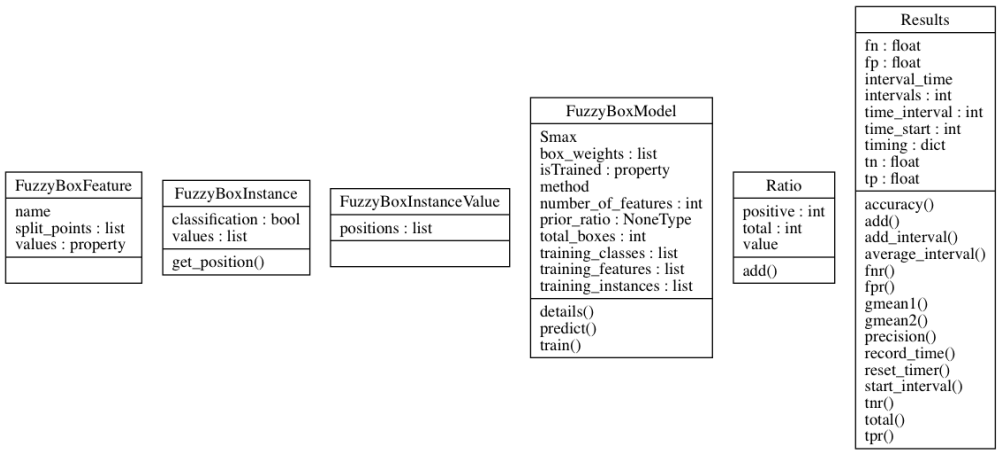


Figure 3.4: Fuzzy Boxes main classes

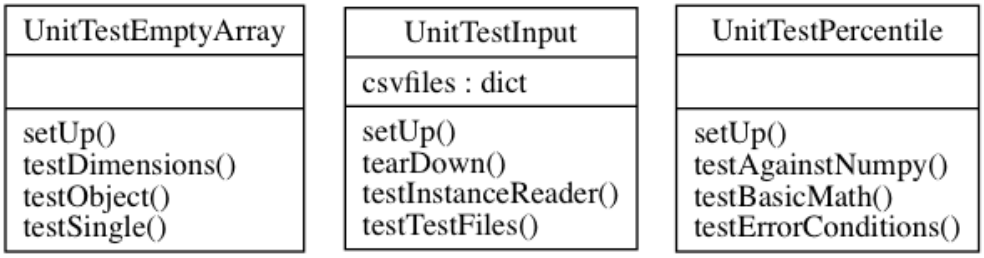


Figure 3.5: Fuzzy Boxes test classes

Offline timing tests of all algorithms was performed via the use of a custom timer class in Python. This class can be found in the file `interval_timer.py` (Appendix A) and has methods to make usage familiar to users of the `tick` and `tock` functions in MATLAB.

Although FANN and LIBSVM are primarily written in C and C++ respectively, both libraries come packaged with Python bindings, making a test harness easy to develop. The Python bindings add a computational cost to the algorithm, primarily when loading and starting the Python interpreter. Since timing measurements are made from within the Python code, the startup time of the interpreter does not affect the results presented here.

One thing to note with this environment, is that FANN and LIBSVM still contain a small amount of overhead by running C modules from Python, though this should not be a significant factor considering the C/C++ modules are precompiled and CPython (the primary Python implementation and also written in C) supports this convention.

A more accurate way to measure timing performance would be to use logic probes either directly on the communications bus, or by signalling different stages via the use of general-purpose input/output (GPIO) pins. For example, one could

use one pin to signal "read value from sensor", and another for "prediction returned from algorithm". In order to automate the process and to avoid setup complexity, this was not performed. Instead, the stopwatch tick/tock calls were placed immediately before and after the calls to the ML prediction function. The same technique was also used to measure training time for each algorithm.

3.6 HWIL Testing

A simple model describing many software and hardware solutions is the Input-Processing-Output (IPO) model (Figure 3.6) which describes three major components. For a real-time automated VRF controller, Remote Sensing (RS) data would form the primary input, in this case the data consists of RNDVI and EC_a measurements, but it is possible other sensor inputs could be viable for this. The output stage would consist of at least one actuator, in the case of binary on/off applicators, or a voltage level which controls the amount of fertiliser being applied. For a real-world deployment of VRF systems, the output stage would need to apply smoothing when using binary classification techniques as in this thesis. Failure to do so could result in damage to the variable-rate application nozzles due to rapid on/off transitions. The *processing* stage is the main focus of this thesis, where ML techniques are tested for suitability to the task of rapidly reading input data, and affecting the output stage in a timely manner. Both the IPO model and Hardware-in-the-Loop (HWIL) testing are used extensively in the automotive computing industry, but have not yet seen widespread adoption within PA (Goel, 2010).

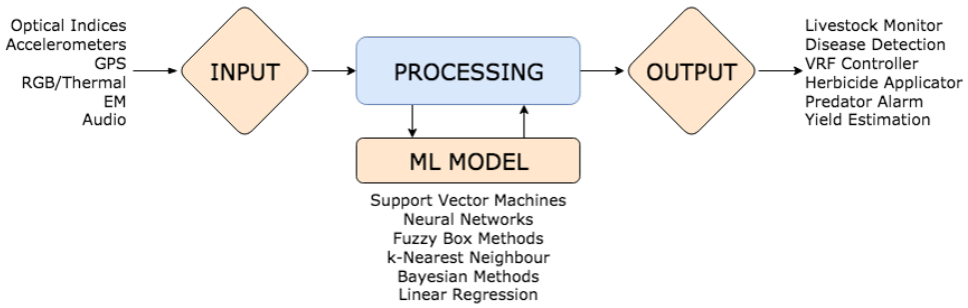
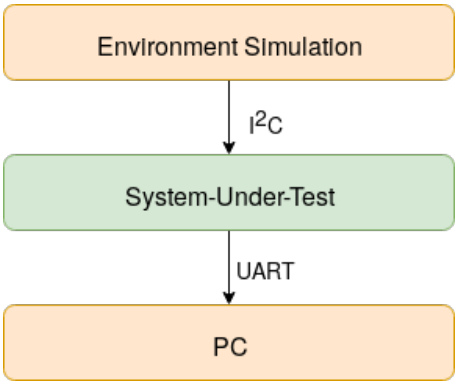


Figure 3.6: Input-Processing-Output model can be used for common PA applications

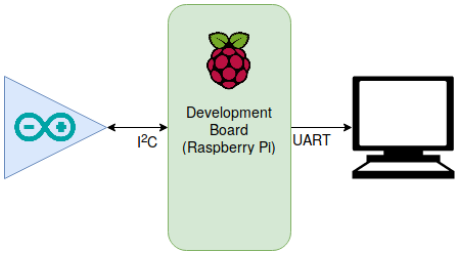
In order to verify the applicability of this model and timing constraints of ML algorithm to VRF tasks, HWIL testing was performed by simulating the input and output stages (Figure 3.7a).

The HWIL simulation was created using an ATmega328P micro controller on

the Arduino platform which acted as an imitation sensor, connected to a Raspberry Pi development board via the I^2C [1] protocol. The output stage was simulated by connecting the Raspberry Pi's serial port to a desktop PC and communicating via Universal asynchronous receiver-transmitter (UART). The Raspberry Pi was the System Under Test (SUT) and executed a Logic-Labelled Finite-State Machine (LLSFM) model which verified the timing constraints within a real, albeit simulated, environment. This is shown in Figure 3.7b.



(a) HWIL testing block diagram



(b) HWIL testing with hardware devices

Figure 3.7: Simulating environmental inputs and checking output against expected behaviour

The micro controller was programmed with a subset of the original data set, keeping the same ratio of positive and negative class instances. Due to the extremely limited memory found on micro controllers, only 180 instances were loaded. Since the HWIL simulation was only looking at verifying timing constraints, the limited number of instances did not affect the outcome of the simulation. The Raspberry Pi board was connected to the PC via a UART serial connection operating at a baud rate of 115 200 bit/s. The data was read from the Arduino device using the I^2C bus, this was chosen due to the high bandwidth ceiling and ease of connection. I^2C only requires two signal lines and no voltage level shifting when connecting two devices, even when those devices operate at different voltage levels (3.3v Raspberry Pi, and 5v Arduino). The imitation sensor code is very simple; it works by receiving a single command byte from the host and returning either an NDVI value or an EC_a value, both represented as 4-byte floating point numbers.

The SUT needs to send 1-byte to the Arduino in order to receive each of the two 4-byte features, and another byte to advance the sample counter (3 total). When the I^2C bus is operating in fast mode plus, the maximum bandwidth is 1 Mbit/s, thus the maximum completed predictions per second when using I^2C in this mode

[1] `https://web.archive.org/web/20190107182009/https://www.nxp.com/docs/en/user-guide/UM10204.pdf`

is 11363 (Equation 3.6). This is $\approx 88\,\mu s$ per prediction.

$$\textbf{Predictions/second} = \frac{1000000/8}{3+4+4} = 11363.\overline{63} \tag{3.6}$$

The code running on the Raspberry Pi was an LLSFM being executed by the `clfsm` scheduler. Although when designing real-world systems using LLSFMs, care should be taken to eliminate all control structures used within a machine's states, in this case we are only verifying timing results and so can go against this advice. The ML algorithms all have a `predict` function which contains many loops, conditionals, and other control structures. This function was not rewritten as an LLSFM as it would effectively require the algorithms to be completely re-implemented using this paradigm. This means that one of the benefits of using an LLSFM (*i.e.* formally verifiable behaviour) will be lost within that state which calls `predict`, but will be retained for other states.

The tester LLSFM is shown in Figure A.1 and includes the `python_interface` C module which was written for this work (Appendix A). This modules embeds a Python interpreter into the C code so that the startup overhead of the interpreter is only applied once within the `SETUP` state. The Python file `python_interface.py` defines the `python_predict` function, which was modified once for each algorithm being tested in order to use a pre-trained model corresponding to that algorithm. These models were either loaded from disk or trained during the call to `python_setup`.

The `after_us(n)` function is part of the `gusimplewhiteboard` module and is not publicly available, however this function simply returns `true` if it was been longer than n microseconds since the current state was entered.

Variables

Type	Name
int	i2c_bus
int	i2c_dev
int	bytes_written
char[4]	bytes_read
unsigned...	feature_id
int	pystatus
float[2]	values
int	sample_id

(a) LLFSM variables

Includes

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>
#include "python_interface.h"
#include "gusimplewhiteboard.h"

#define I2C_ADDRESS 0x42
#define I2C_DEVICE "/dev/i2c-0"
#define DATASET_SIZE 180
#define N_FEATURES 2
```

(b) LLFSM include paths

Figure 3.8: Environment configuration for PredictionTesterLLFSM

3.7 Development Environment

The primary development environment was Neovim[2] running on both a mid-2015 MacBook Pro, and a desktop PC with an Intel Core i7 4770, both containing 16 GB Random-Access Memory (RAM). A mixture of operating systems were used over the course of this work, including Windows, macOS, Debian and Fedora on the high powered machines, and Raspbian on the Raspberry Pi. The `virtualenv` Python package was extremely useful for managing multiple development environments and compatibility testing for Python 2 and Python 3. The PredictionTester LLSFM was created using the `MiCASE` software created by MiPAL[3] specifically for LLSFM development. A Continuous Integration (CI) server was set up running the Jenkins[4] software package. This server was intended to be used as a central point for the latest copy of the code, as well as to generate build logs and regression testing reports. However, it was found to not be very useful as the code for this work was under continuous change and there were no *releases* or existing code base to have integration problems with.

[2] `https://web.archive.org/web/20190109035231/https://neovim.io/`
[3] `https://web.archive.org/web/20190227044813/http://mipal.net.au/`
[4] `https://web.archive.org/web/20181229082713/https://jenkins.io/doc/`

Chapter 4

Results and Discussion

Presented in this section are the results of the analysis and testing of the Machine Learning (ML) algorithms as previously described. An analysis of the Fuzzy Boxes (FB) algorithm in terms of both time and memory complexity is first shown in order to provide context for how FB performs compared to existing algorithms. Model parameters for the Support Vector Machine (SVM) and Artificial Neural Network (ANN) algorithm were found via 10-fold Cross Validation (10CV) and are presented in 4.2. Accuracy and timing results, which form the primary comparison points between these algorithms, are then presented in 4.3 and 4.4.

4.1 Fuzzy Boxes Analysis

A basic complexity analysis of the training time and memory usage of the FB algorithm was performed in order to compare the scalability in relation to the SVM and ANN algorithms. Results are shown in Table 4.1. Complexity of training time and memory usage domains is already known in the case of the SVM (Tsang, Kwok and Cheung, 2005) and ANN (Owechko and Shams, 1994).

	Memory Usage	Training Time
ANN	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
C-SVC	$\mathcal{O}(m^2)$	$\mathcal{O}(m^3)$
Fuzzy Boxes	$\mathcal{O}(c^n)$	$\mathcal{O}(cn)$

Table 4.1: Worst-case memory usage and training time complexity for ML models $n =$ features, $m =$ training instances, $c =$ box splits

Equation 4.2 can be used to find the model size (in MiB) given the number of input features (f). Memory use for the FB model can be calculated from the total number of boxes in the model, which is directly related to the minimum and maximum number of split points (Generalised in Equation 4.1). Due to the

exponential nature of the memory requirements in relation to the input features, the space required for an FB model quickly grows beyond the scope of embedded systems. Total memory requirements will be dependent on the datatype used to store the *positive* and *total* counts for each box.

As an example, if we assume the use of 2× 16-bit values per box and a split range of 0–20, we can see that with 5 features, the model only uses 62.6 MiB (Equation 4.3), while with 6 features it uses over 1 GiB (Equation 4.4). Attempting to build a FB model with these parameters over 7 input features results in a model space requirement of over 21 GiB (Equation 4.5), far outside the feasibility of current embedded hardware.

$$\sum_{S=S_{\mathrm{MIN}}}^{S_{\mathrm{MAX}}} (S+1)^{f} \tag{4.1}$$

$$\sum_{S=0}^{20} \frac{4\,(S+1)^{f}}{1024^{2}} \tag{4.2}$$

$$\sum_{S=0}^{20} \frac{4\,(S+1)^{5}}{1024^{2}} = 62.627\,\mathrm{MiB} \tag{4.3}$$

$$\sum_{S=0}^{20} \frac{4\,(S+1)^{6}}{1024^{3}} = 1.126\,\mathrm{GiB} \tag{4.4}$$

$$\sum_{S=0}^{20} \frac{4\,(S+1)^{7}}{1024^{3}} = 21.154\,\mathrm{GiB} \tag{4.5}$$

Training time however has a complexity of $\mathcal{O}(cn)$, which is comparable to the established algorithms and provides hope for this algorithm to be optimised and potentially implemented for stream-learning applications.

It should be noted that the complexity values given are worst-case. In real-world scenarios, SVM algorithms such as the C-Support Vector Classifier (C-SVC) can operate with a much lower time complexity since the training time is dependent on which of the training instances are used for support vectors.

Figure 4.1 shows the effect of the `depth` parameter on accuracy for both the `even` and `percentile` splitting methods. It was expected that the `even` splitting method would provide worse accuracy results for low `depth` values because it does not take instance density into account like `percentile` splitting does, and is therefore more sensitive to outlier values affecting the split positions. We can see that the accuracy value converges at around 5 box splits, since at this depth the boxes are fine-grained enough to negate any difference in splitting method.

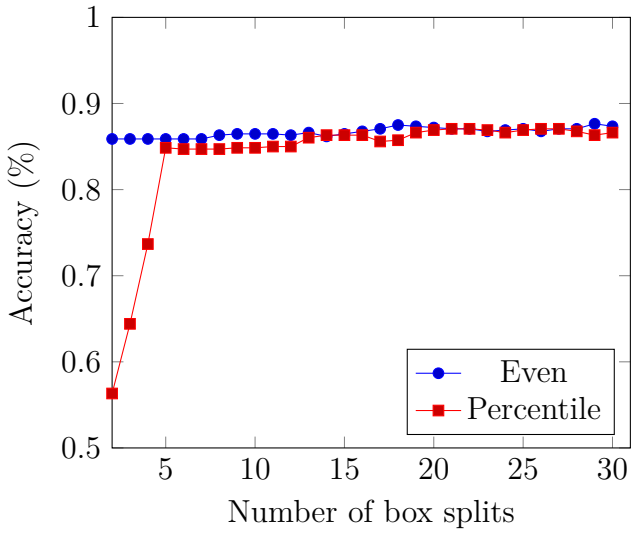


Figure 4.1: Split count effect on prediction accuracy for FB algorithms

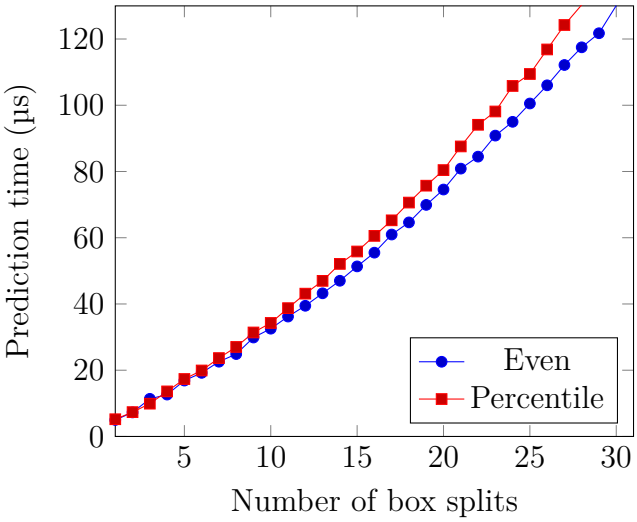


Figure 4.2: Split count effect on prediction timings for Fuzzy Boxes algorithms

Since the *number* of boxes does not change between the splitting methods, the only difference in prediction time comes from the operations required to calculate the split points (Algorithm 1). Calculation of percentiles is slower than the calculation of even splits, and this difference becomes more apparent as the number of boxes grows (Figure 4.2).

One of the current limitations of FB is that it is a binary classifier and has not been designed with multiclass classification or regression in mind. This makes it too responsive for *direct* control over fertiliser equipment, however this could be alleviated by applying damping or smoothing to the output.

4.2 Model Parameters

After performing a 10-fold cross-validation hyper-parameter search, model parameters were found for the SVM algorithm (Table 4.2) and back-propagation ANN algorithm (Table 4.3).

Parameter	Value
C	2048
Gamma	0.0078125

Table 4.2: Model parameters for the SVM algorithm

Parameter	Value
Hidden Layers	7
Learning Rate	0.007
Internal Activation Function	Sigmoid
Output Activation Function	Binary Step

Table 4.3: Model parameters for the fully-connected BP ANN

In order to find suitable parameters for the FB algorithm, a different approach was taken due to the fact that the algorithm performance generally improves with an increasing number of box splits. A 10-fold cross-validation method was again followed, however the final parameter values were not simply those with the highest accuracy.

One of the primary claimed benefits of the FB algorithm is the ability to tune the model not only to accuracy, but also to timing performance. As shown in Figure 4.2, prediction time increases in a non-linear manner as the number of splits increases.

This was expected due to the exponential memory complexity of the model, which requires a corresponding number of operations in order to find the n-dimensional position of values. Three depth values were chosen for comparison to the existing algorithms in order to demonstrate the scalable nature of the algorithm. A "shallow" value of 5, a "mid" value of 10, and a "deep" value of 20 were chosen (Table 4.4).

4.3 Model Accuracy

Comparing the `even` and `percentile` splitting methods of the FB model shows not much accuracy difference between the two. The `percentile` method offers a more robust model since it can handle outliers more easily than the even method.

Parameter	Value
Max Depth (shallow)	5
Max Depth (mid)	10
Max Depth (deep)	20

Table 4.4: Model parameters for the FB algorithm

More importantly, it can be seen from Figure 4.3 that the the `percentile` method produces fewer false-positives and more true-negatives for this data set. This is extremely important for (potentially automated) systems which may be delivering harmful chemicals on a mass scale to crops. The ability to correctly decide when *not to* apply fertiliser is arguably much more important than the ability to decide when to apply fertiliser.

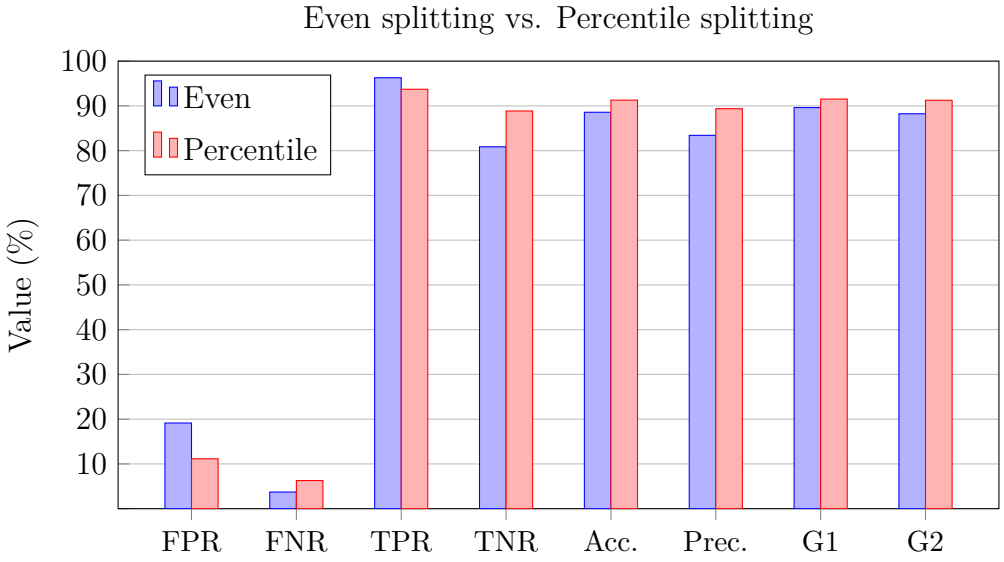


Figure 4.3: Performance metrics for the different Fuzzy Box splitting methods

The SVM and ANN models also showed high accuracy for this data set, with all algorithms achieving over 84% in both accuracy and precision metrics. These results show that all tested models are suitable for making high-accuracy binary predictions of Nitrogen (N)-requirements, and shows promise for producing optimised models from larger data sets in the future.

Comparing the FB algorithm to the existing algorithms, it can be seen that FB produced higher-accuracy models when the *depth* parameter was sufficiently increased. The `percentile` splitting method consistently produced better accuracy and precision metrics, however the `even` splitting method has a slightly better $F1_{score}$. These close results suggest that the difference between these algorithms' predictive accuracy may be statistically insignificant, though no formal

significance tests were performed as part of this work. As such, each of the algorithms tested here could potentially be considered for real-time prediction within Precision Agriculture (PA) contexts.

Algorithm	Accuracy	Precision	$F1_{score}$	G_{mean}
ANN	85.37	85.37	76.21	84.08
C-SVC	85.77	85.77	76.06	83.38
FB_5 (Even)	84.85	84.85	86.36	84.13
FB_{10} (Even)	84.85	84.85	86.39	84.09
FB_{20} (Even)	86.91	86.91	**87.76**	86.64
FB_5 (Percentile)	84.88	84.88	85.88	85.88
FB_{10} (Percentile)	86.47	86.47	86.67	86.46
FB_{20} (Percentile)	**87.21**	**87.21**	87.45	**87.18**

Table 4.5: Accuracy metrics comparison

These metrics provide an optimistic look at the potential for FB models for use in Variable-Rate Fertiliser (VRF).

4.4 Timing Results

Raw timing results are presented for the various ML algorithms executed on a 2.2 GHz Core i7 (Table 4.6) and a Raspberry Pi with BCM2837 chipset (Table 4.7). These results show that the algorithms themselves are more than capable of predicting at high enough speeds to be used in aircraft fertiliser delivery. Spatial Resolution for these tests is defined as the distance covered during prediction time and assumes an aircraft moving at 60 m/s, which is a standard speed for the widely used AirTractor AT-502B crop spraying aircraft. Even when operating on the low powered BCM2837 chipset, a spatial resolution of less than 3.1 cm per prediction can be achieved by all tested algorithms.

	Training (ms)	Prediction (µs)	Spatial Resolution (mm)
ANN	846.46	2.09	0.1
C-SVC	97.88	19.76	1.2
FB5-Even	64.88	15.87	1.0
FB5-Perc	82.76	16	1.0
FB10-Even	130.89	32.27	1.9
FB10-Perc	182.3	30.69	1.8
FB20-Even	251.28	69.61	4.2
FB20-Perc	442.95	64.92	3.9

Table 4.6: Timing and Spatial Resolutions (mm/prediction) for Core i7

	Training (ms)	Prediction (µs)	Spatial Resolution (mm)
ANN	12449.45	16.64	1.0
C-SVC	786.56	261.72	15.7
FB5-Even	513.88	112.64	6.8
FB5-Perc	670.95	108.6	6.5
FB10-Even	1038.44	234.16	14.0
FB10-Perc	1606.43	218.64	13.1
FB20-Even	1986.48	502.77	30.1
FB20-Perc	3841.17	469.73	28.2

Table 4.7: Timing and Spatial Resolutions (mm/prediction) for Raspberry Pi

All algorithms performed well during the Hardware-in-the-Loop (HWIL) simulation tests, and timing constraints used within the Logic-Labelled Finite-State Machine (LLSFM) tester were all verified as expected. The initial "offline" timing tests were an important step in *validating* the software design, meaning the software under construction is shown to be a valid solution to the problem of VRF. The HWIL timing tests, on the other hand, are important for *verification* of the solution, meaning the software actually does what is expected.

ANN algorithm was a significantly faster predictor than the other algorithms tested, but this came at the cost of a longer training time. This is not so much of an issue for VRF applications since the model can be trained offline and then transferred to the hardware device, though may become a disadvantage for a future "online-learning" model.

SVM algorithm performed faster for both training and predictions than FB with over 10 box splits, but was a slower predictor than FB with 5 box splits. This was true for both the *even* splitting method, and *percentile* splitting method.

By far the slowest predictor was FB with 20 percentile splits, though a possible future modification to the FB model could alter this dramatically.

Chapter 5

Conclusion

The work presented in this thesis shows that Machine Learning (ML) algorithms currently used within Precision Agriculture (PA) are capable of potentially high accuracy binary classification of fertiliser requirements estimation at a specific location within a wheat crop.

This accuracy is predicated on the availability of a high quality remotely sensed data stream which correlated to Nitrogen (N) fertiliser requirements, such as EC_a measurements from the Geonics EM38-MK2. Since these ML tests were performed using data from only one regional area (Warialda, NSW), and all wheat was of the same growing stage (Z30), the results presented here can not be generalised to wider agricultural uses, but provide an optimistic outlook for the future of ML in PA.

The recently presented Fuzzy Boxes (FB) algorithm was implemented in the Python language and then compared to these existing algorithms on the metrics of *accuracy*, *precision*, $F1_{score}$, and G_{mean} in order to determine their performance in predicting both the positive class (*needs fertiliser*), and the negative class (*doesn't need fertiliser*). This new algorithm also provided high-accuracy predictions for the same task and was shown to be competitive with the existing models. FB has the ability to "tune" the speed vs. accuracy trade-off, thus making it competitive in terms of accuracy *or* prediction speed, depending on the model's parameter selection. This makes it a promising candidate for future use within PA in general and for an automated Variable-Rate Fertiliser (VRF) system specifically.

Through profiling of metrics such as system resource utilisation, computational complexity, real-time predictive accuracy, and potential for parallelisation, it is predicted that the FB algorithm can be developed further in order to facilitate a low-energy, embedded system, capable of real-time control over the output of fertiliser distribution systems.

Offline timing tests were conducted on a modern, low powered platform (Raspberry Pi 3B) in order to validate the idea behind executing ML algorithms for use in a dedicated low powered device. Timing tests were also conducted on a modern, high powered platform (Mid 2015 MacBook Pro) for comparison, as well as to quantise the required difference if it was actually found that low powered devices were unfeasible for this task.

After successful validation of the software model, Hardware-in-the-Loop (HWIL) testing was used for verification of timing results. All algorithms successfully passed this verification, showing that a dedicated hardware device designed for a real-time, airborne VRF controller is indeed a feasible goal.

5.1 Future Work

As this work shows, Model-Driven Software Development (MDSD) techniques can be useful for verification and validation of software models and ML algorithms. In the future, it would be beneficial for the PA field to make much more use of these techniques in order to facilitate the design of more robust and reusable PA hardware components.

This includes implementing ML algorithms as Finite-State Machine (FSM) or Logic-Labelled Finite-State Machine (LLSFM) models in order to formally reason about the performance profile of these algorithms. ML implementations could also be created targeting micro controllers and platforms such as Arduino, and new projects such as ArduRTOS should be investigated for suitability to PA.

Artificial Neural Network (ANN) models show promise for this task as the model itself is small and a consistent size. An improvement to FB could be made in order to facilitate this and also to reduce prediction time. Currently, in order to predict the classification of a new instance using FB, it is required to find the instance's "box position" once for every value of S. This makes the prediction time of FB dependant on the values of $S_{\min}$ (usually zero) and $S_{\max}$. Since the maximum number of operations required for each value of S is equal to S itself, the worst-case prediction time complexity is $\mathcal{O}(n^2)$. In order to reduce this complexity, it would be possible to *flatten* the probability values for each layer of S into a single n-dimensional array. By doing this, the worst-case number of operations required to perform a lookup would be constant relative to the maximum value of S, giving a complexity of $\mathcal{O}(n)$. It would also be beneficial to use a data set with a higher degree of features in order to test and compare the algorithms' ability to scale with model complexity.

Another future goal from this work would be the creation of a framework for

PA applications which makes use of ML processing. The creation of this framework could facilitate real-time agricultural robotics development and encourage reuse of software components across PA.

Parallelisation of new and existing ML algorithms such as FB will also be of importance in the future for taking advantage of multi- and many-core devices. FB in particular has the potential for operating well in highly parallel environments due to the fact that each depth layer of each feature in the model is independent. This means that the training for each feature could be performed as a separate process, and that the prediction could take advantage of the generic scatter/gather parallel processing operations.

References

Abeni, L., A. Goel, C. Krasic, J. Snow and J. Walpole. 'A measurement-based analysis of the real-time performance of linux'. In: *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*. IEEE Comput. Soc, 2002, pp. 133–142. ISBN: 978-0-7695-1739-1. (Visited on 11/03/2018).

Agathos, Spiros N., Alexandros Papadogiannakis and Vassilios V. Dimakopoulos. 'Targeting the Parallella'. In: *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*. Ed. by Jesper Larsson Träff, Sascha Hunold and Francesco Versaci. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 662–674. ISBN: 978-3-662-48096-0.

Air Tractor, Inc. *AT-502B - Air Tractor*. 2017. (Visited on 18/06/2017).

Aronson, Rory Landon. *FarmBot: Humanity's open-source automated precision farming machine*. Sept. 2013. (Visited on 11/02/2018).

Australian Bureau of Agricultural and Resource Economics. *Australian Crop Report*. Tech. rep. 180. Australian Bureau of Agricultural and Resource Economics, Dec. 2016, p. 29.

Australian Government and Queensland Government. *Reef 2050 Long-Term Sustainability Plan*. Tech. rep. Australian Government Department of Environment, 2015.

Barry, D. Andrew, Jean-Luc Liardon, Philippe Paccaud, Pascal Klaus, Nawaaz S. Gujja Shaik, Abolfazl Irani Rahaghi et al. 'A low-cost, autonomous mobile platform for limnological investigations, supported by high-resolution mesoscale airborne imagery'. In: *PLOS ONE* 14.2 (Feb. 2019), pp. 1–21. DOI: 10.1371/journal.pone.0210562. URL: https://doi.org/10.1371/journal.pone.0210562.

Bell, MJ. *A review of nitrogen use efficiency in sugarcane*. Tech. rep. Sugar Research Australia Limited, Dec. 2014.

Bengtsson, Johan, Kim Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. 'UPPAAL — a tool suite for automatic verification of real-time systems'. In:

Hybrid Systems III. Ed. by Rajeev Alur, Thomas A. Henzinger, Eduardo D. Sontag, G. Goos, J. Hartmanis and J. Leeuwen. Vol. 1066. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 232–243. ISBN: 978-3-540-61155-4. (Visited on 18/02/2018).

Biello, David. 'Fertilizer runoff overwhelms streams and rivers–creating vast "Dead Zones"'. In: *Scientific American* 3.14 (2008), p. 2008.

Billington, David, Vladimir Estivill-Castro, René Hexel and Andrew Rock. 'Modelling behaviour requirements for automatic interpretation, simulation and deployment'. In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2010, pp. 204–216.

— 'Non-Monotonic Reasoning For Requirements Engineering: State Diagrams driven by Plausible Logic'. In: *ENASE 2010 - Proceedings of the 5th International Conference on Evaluation of Novel Approaches to Software Engineering* (Jan. 2010).

Booth, Taylor L. *Sequential machines and automata theory*. English. New York [etc.: Wiley, 1967. ISBN: 978-0-471-08848-6.

Bowden, Phillip, Jan Edwards, Nathan Ferguson, Tim McNee, Bill Manning, Karen Roberts et al. *PROCROP Wheat growth & development*. English. Ed. by Julie White and Jan Edwards. NSW Dept. of Primary Industries, 2008, viii, 92 p. : ISBN: 9780734718945.

Bragagnolo, Jardes, Telmo Jorge Carneiro Amado, Rodrigo da Silveira Nicoloso, Antônio Luis Santi, Jackson Ernani Fiorin and Fabiano Tabaldi. 'Optical crop sensor for variable-rate nitrogen fertilization in corn: II - indices of fertilizer efficiency and corn yield'. In: *Revista Brasileira de Ciência do Solo* 37 (2013), pp. 1299–1309. ISSN: 0100-0683.

Brosnan, Tadhg and Da-Wen Sun. 'Inspection and grading of agricultural and food products by computer vision systems—a review'. In: *Computers and Electronics in Agriculture* 36.2 (Nov. 2002), pp. 193–213. ISSN: 0168-1699.

Bryce, R. and R. Kuhn. 'Software Testing'. In: *Computer* 47.2 (Feb. 2014), pp. 21–22. ISSN: 0018-9162.

Capraro, Flavio, Daniel Patino, Santiago Tosetti and Carlos Schugurensky. 'Neural network-based irrigation control for precision agriculture'. In: *IEEE International Conference on Networking, Sensing and Control*. IEEE, 2008, pp. 357–362. ISBN: 1-4244-1685-X.

Cavigelli, Michel A. 'Agriculture and the Nitrogen Cycle'. In: *Ecology* 86.9 (2005), pp. 2548–2550. ISSN: 1939-9170.

Chang, Chih-Chung and Chih-Jen Lin. 'LIBSVM: a library for support vector machines'. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 2.3 (2011), p. 27. ISSN: 2157-6904.

Chilimbi, Trishul M., Suzue, Yutaka, Apacible, Johnson and Kalyanaraman, Karthik. 'Project Adam: Building an Efficient and Scalable Deep Learning Training System'. In: *OSDI*. Vol. 14. 2014, pp. 571–582.

Chiodo, Massimiliano, Paolo Giusto, Attila Jurecska, Harry C. Hsieh, Alberto Sangiovanni-Vincentelli and Luciano Lavagno. 'Hardware-software codesign of embedded systems'. In: *IEEE Micro* 14.4 (1994), pp. 26–36. ISSN: 0272-1732.

Clark, R. L. and R. L. McGuckin. 'Variable Rate Application Technology: An Overview'. English. In: *Precision Agriculture*. Ed. by P.C. Robert, R.H. Rust and W.E. Larson. Madison, WI: American Society of Agronomy, Crop Science Society of America, Soil Science Society of America, 1996, pp. 855–862. ISBN: 978-0-89118-257-3.

Cohen, David. 'Machine Learning meets Precision Agriculture: classification of real-time sensor data streams for fertiliser applicators'. MA thesis. University of New England, 2016.

Crop Circle ACS-210 Product Sheet. Holland Scientific. 2004.

D'Ausilio, Alessandro. 'Arduino: A low-cost multipurpose lab equipment'. en. In: *Behavior Research Methods* 44.2 (June 2012), pp. 305–313. ISSN: 1554-3528.

Dammer, Karl-Heinz, H. Thöle, T. Volk and B. Hau. 'Variable-rate fungicide spraying in real time by combining a plant cover sensor and a decision support system'. In: *Precision Agriculture* 10.5 (Oct. 2009), pp. 431–442. ISSN: 1573-1618.

Dammer, Karl-Heinz. 'Real-time variable-rate herbicide application for weed control in carrots'. In: *Weed Research* 56.3 (June 2016), pp. 237–246. ISSN: 1365-3180.

Department of Health. *Nitrate in Drinking Water*. 2011.

Diacono, Mariangela, Pietro Rubino and Francesco Montemurro. 'Precision nitrogen management of wheat. A review'. In: *Agronomy for Sustainable Development* 33.1 (Jan. 2013), pp. 219–241. ISSN: 1773-0155.

Dietrich, Sven-Thorsten and Daniel Walker. 'The evolution of real-time linux'. In: *7th RTL Workshop*. Citeseer. 2005.

Dimitriadis, S. and C. Goumopoulos. 'Applying Machine Learning to Extract New Knowledge in Precision Agriculture Applications'. In: *2008 Panhellenic Conference on Informatics*. Aug. 2008, pp. 100–104.

Drummond, S., A. Joshi and K. A. Sudduth. 'Application of neural networks: precision farming'. In: *1998 IEEE International Joint Conference on Neural*

Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227). Vol. 1. May 1998, 211–215 vol.1.

Du, Lin, Wei Gong, Shuo Shi, Jian Yang, Jia Sun, Bo Zhu et al. 'Estimation of rice leaf nitrogen contents based on hyperspectral LIDAR'. In: *International Journal of Applied Earth Observation and Geoinformation* 44.Supplement C (Feb. 2016), pp. 136–143. ISSN: 0303-2434.

Eigenberg, Roger A, John A Nienaber, Bryan L Woodbury and Richard B Ferguson. 'Soil conductivity as a measure of soil and crop status—A four-year summary'. In: *Soil Science Society of America Journal* 70.5 (2006), pp. 1600–1611.

Erisman, Jan Willem, James Galloway Mark A. Sutton, Zbigniew Klimont and Wilfried Winiwarter. 'How a century of ammonia synthesis changed the world'. In: *Nature Geoscience* 1.10 (2008), pp. 636–639. ISSN: 1752-0894.

Ernst, Rolf, Jörg Henkel and Thomas Benner. 'Hardware-software cosynthesis for microcontrollers'. In: *IEEE Design & Test of computers* 10.4 (1993), pp. 64–75. ISSN: 0740-7475.

Espejo-Herrera, Nadia, Kenneth P Cantor, Nuria Malats, Debra T Silverman, Adonina Tardón, Reina Garca-Closas et al. 'Nitrate in drinking water and bladder cancer risk in Spain'. In: *Environmental research* 137 (2015), pp. 299–307. ISSN: 0013-9351.

Estivill-Castro, Vladimir and René Hexel. 'Correctness by Construction with Logic-Labeled Finite-State Machines – Comparison with Event-B'. In: *Proceedings of the 2014 23rd Australian Software Engineering Conference*. ASWEC '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 38–47. ISBN: 978-1-4799-3149-1.

— 'Deterministic Executable Models Verified Efficiently at Runtime - An Architecture for Robotic and Embedded Systems'. In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS - Science and Technology Publications, 2017, pp. 29–40. ISBN: 978-989-758-210-3. (Visited on 29/08/2017).

— 'Run-time verification of regularly expressed behavioral properties in robotic systems with logic-labeled finite state machines'. In: *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. IEEE, Dec. 2016, pp. 281–288. ISBN: 978-1-5090-4616-4. (Visited on 29/08/2017).

Estivill-Castro, Vladimir, Rene Hexel and Alberto Ramrez Regalado. 'Architecture for logic programing with arrangements of finite-state machines'. In: *2016 1st CPSWeek Workshop on Declarative Cyber-Physical Systems (DCPS)*. IEEE,

Apr. 2016. DOI: 10.1109/dcps.2016.7588297. URL: https://doi.org/10.1109%2Fdcps.2016.7588297.

Estivill-Castro, Vladimir, Rene Hexel and David A. Rosenblueth. 'Efficient Modelling of Embedded Software Systems and Their Formal Verification'. In: *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01*. APSEC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 428–433. ISBN: 978-0-7695-4922-4. DOI: 10.1109/APSEC.2012.21. URL: https://doi.org/10.1109/APSEC.2012.21.

Estivill-Castro, Vladimir, René Hexel and Josh Stover. 'Modeling, Validation and Continuous Integration of Software Behaviours for Embedded Systems'. In: *Modelling Symposium (EMS), 2015 IEEE European*. IEEE. Institute of Electrical and Electronics Engineers (IEEE), Oct. 2015, pp. 89–95.

— 'Models Testing Models in Continuous Integration of Model-Driven Development'. In: *IASTED Int. Symp. Software Engineering and Applications (SEA 2015), P.* ACTA Press, Oct. 2015, pp. 1–8.

Estivill-Castro, Vladimir and David A. Rosenblueth. 'Model Checking of Transition-Labeled Finite-State Machines'. In: *Software Engineering, Business Continuity, and Education*. Ed. by Tai-hoon Kim, Hojjat Adeli, Haeng-kon Kim, Heau-jo Kang, Kyung Jung Kim, Akingbehin Kiumi et al. Vol. 257. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 61–73. ISBN: 978-3-642-27206-6. (Visited on 18/02/2018).

Falzon, Greg, David W. Lamb and Derek Schneider. 'The dynamic aerial survey algorithm architecture and its potential use in airborne fertilizer applications'. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 5.6 (2012), pp. 1772–1779. ISSN: 1939-1404.

Ferdoush, Sheikh and Xinrong Li. 'Wireless Sensor Network System Design Using Raspberry Pi and Arduino for Environmental Monitoring Applications'. In: *The 9th International Conference on Future Networks and Communications (FNC'14)/The 11th International Conference on Mobile Systems and Pervasive Computing (MobiSPC'14)/Affiliated Workshops* 34.Supplement C (Jan. 2014), pp. 103–110. ISSN: 1877-0509.

Fernandes, Steven Lawrence and Josemin G. Bala. 'Low Power Affordable and Efficient Face Detection in the Presence of Various Noises and Blurring Effects on a Single-Board Computer'. In: *Emerging ICT for Bridging the Future - Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1*. Ed. by Suresh Chandra Satapathy, A. Govardhan, K. Srujan Raju and J. K. Mandal. Cham: Springer International Publishing, 2015, pp. 119–127. ISBN: 978-3-319-13728-5.

Ferreira, João F., Cristian Gherghina, Guanhua He, Shengchao Qin and Wei-Ngan Chin. 'Automated verification of the FreeRTOS scheduler in Hip/Sleek'. en. In: *International Journal on Software Tools for Technology Transfer* 16.4 (Aug. 2014), pp. 381–397. ISSN: 1433-2787.

Flach, Peter. *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press, 2012.

Fraser, Grant, Ken Rohde and Mark Silburn. 'Fertiliser management effects on dissolved inorganic nitrogen in runoff from Australian sugarcane farms'. In: *Environmental Monitoring and Assessment* 189.8 (July 2017), p. 409. ISSN: 1573-2959.

Ganganwar, Vaishali. 'An overview of classification algorithms for imbalanced datasets'. In: *International Journal of Emerging Technology and Advanced Engineering* 2.4 (2012), pp. 42–47.

Geonics, Ltd. *Geonics EM38-MK2 Ground Conductivity Meter*. 2013.

Goel, Anita. *Computer fundamentals*. Pearson Education India, 2010.

Göktoan, Ali Haydar and Salah Sukkarieh. 'Autonomous Remote Sensing of Invasive Species from Robotic Aircraft'. In: *Handbook of Unmanned Aerial Vehicles*. Ed. by Kimon P. Valavanis and George J. Vachtsevanos. Dordrecht: Springer Netherlands, 2015, pp. 2813–2834. ISBN: 978-90-481-9707-1.

Gonzalez-Dugo, V., D. Goldhamer, P. J. Zarco-Tejada and E. Fereres. 'Improving the precision of irrigation in a pistachio farm using an unmanned airborne thermal system'. In: *Irrigation science* 33.1 (2015), pp. 43–52. ISSN: 0342-7188.

Hagan, Martin T., Howard B. Demuth and Mark H. Beale. *Neural network design*. Vol. 20. Pws Pub. Boston, 1996.

Harel, David. 'Statecharts: A Visual Formalism for Complex Systems'. In: *Sci. Comput. Program.* 8.3 (June 1987), pp. 231–274. ISSN: 0167-6423.

Harter, Thomas and Jay R. Lund. *Addressing Nitrate in California's Drinking Water: With a Focus on Tulare Lake Basin and Salinas Valley Groundwater: Report for the State Water Resources Control Board Report to the Legislature*. Center for Watershed Sciences, University of California, Davis, 2012.

Haykin, Simon S. *Neural networks: a comprehensive foundation*. 2nd ed. Upper Saddle River, N.J: Prentice Hall, 1999. ISBN: 978-0-13-273350-2.

Heffernan, Donal, Ciaran MacNamee and Padraig Fogarty. 'Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety standard as a guide for the definition of the monitored properties'. In: *IET Software* 8.5 (2014), pp. 193–203. ISSN: 1751-8814.

Hegde, Gopalakrishna, Siddhartha, Nachiappan Ramasamy and Nachiket Kapre. 'CaffePresso: An Optimized Library for Deep Learning on Embedded Accelerator-based Platforms'. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '16. New York, NY, USA: ACM, 2016, 14:1–14:10. ISBN: 978-1-4503-4482-1.

Hinton, G., L. Deng, D. Yu, G. E. Dahl, A. Mohamed, Jaitly, N. et al. 'Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups'. In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012), pp. 82–97. ISSN: 1053-5888.

Hobson, Richard F. 'Computing Science Hardware Laboratories and the LSI Revolution'. In: *Papers of the SIGCSE/CSA Technical Symposium on Computer Science Education*. SIGCSE '78. Detroit, Michigan: ACM, 1978, pp. 160–170. DOI: 10.1145/990555.990610. URL: http://doi.acm.org/10.1145/990555.990610.

Holland Scientific, Inc. *Raptor ACS-225LR*. 2011. (Visited on 06/10/2017).

Ibrahim, Dogan. *Microcontroller based applied digital control*. John Wiley & Sons, 2006.

Jones, C. L., P. R. Weckler, N. O. Maness, R. Jayasekara, M. L. Stone and D. Chrz. 'Remote Sensing to Estimate Chlorophyll Concentration in Spinach Using Multi-Spectral Plant Reflectance'. In: *Transactions of the American Society of Agricultural and Biological Engineers (ASABE)* 50.6 (2007), pp. 2267–2273. DOI: 10.13031/2013.24079. URL: https://doi.org/10.13031%2F2013.24079.

Jørgensen, J. R. and R. N. Jørgensen. 'Uniformity of wheat yield and quality using sensor assisted application of nitrogen'. In: *Precision Agriculture* 8.1 (Apr. 2007), pp. 63–73. ISSN: 1573-1618.

Katz, J.J., R.C. Dougherty and L.J. Boucher. '7 - Infrared and Nuclear Magnetic Resonance Spectroscopy of Chlorophyll**Based on work performed under the auspices of the U.S. Atomic Energy Commission.' In: *The Chlorophylls*. Ed. by LEO P. VERNON and GILBERT R. SEELY. Academic Press, 1966, pp. 185–251. ISBN: 978-1-4832-3289-8. DOI: https://doi.org/10.1016/B978-1-4832-3289-8.50013-8. URL: http://www.sciencedirect.com/science/article/pii/B9781483232898500138.

Kim, Y., R. G. Evans and W. M. Iversen. 'Remote Sensing and Control of an Irrigation System Using a Distributed Wireless Sensor Network'. In: *IEEE Transactions on Instrumentation and Measurement* 57.7 (July 2008), pp. 1379–1387. ISSN: 0018-9456.

Kitchen, N. R., D. F. Hughes, K. A. Sudduth and Stuart J. Birrell. 'Comparison of variable rate to single rate nitrogen fertilizer application: corn production and residual soil NO3-N'. In: *Site-specific management for agricultural systems*. American Society of Agronomy, Crop Science Society of America, Soil Science Society of America, 1995, pp. 427–441. ISBN: 0-89118-260-8.

Koch, B., R. Khosla, W. M. Frasier, D. G. Westfall and D. Inman. 'Economic Feasibility of Variable-Rate Nitrogen Application Utilizing Site-Specific Management Zones.' English. In: *Agronomy Journal* 96.6 (2004), pp. 1572–1580.

Kochlán, M., M. Hodo, L. echovi, J. Kapitulk and M. Jureka. 'WSN for traffic monitoring using Raspberry Pi board'. In: *2014 Federated Conference on Computer Science and Information Systems*. Sept. 2014, pp. 1023–1026.

Lamb, D. W., D. A. Schneider and J. N. Stanley. 'Combination active optical and passive thermal infrared sensor for low-level airborne crop sensing'. In: *Precision agriculture* 15.5 (2014), pp. 523–531. ISSN: 1385-2256.

Lamb, D. W., D. A. Schneider, M. G. Trotter, M. T. Schaefer and I. J. Yule. 'Extended-altitude, aerial mapping of crop NDVI using an active optical sensor: A case study using a Raptor™ sensor over wheat'. In: *Computers and electronics in agriculture* 77.1 (2011), pp. 69–73. ISSN: 0168-1699.

Lambert, David C. *Python-ELM: Extreme Learning Machine implementation in Python*. GitHub, 2013.

LeCun, Yann, Yoshua Bengio and Geoffrey Hinton. 'Deep learning'. In: *Nature* 521 (May 2015), p. 436.

Livingstone, David J. *Artificial Neural Networks: Methods and Applications (Methods in Molecular Biology, Vol. 458)*. 1st Edition. Methods in Molecular Biology volume 458. Springer, 2008.

Maes, RE, GGM Dedene et al. 'Reframing the Zachman information system architecture framework'. In: *AmsterdamTinbergen Institute* (1996).

Mayfield, A. H. and S. P. Trengove. 'Grain yield and protein responses in wheat using the N-Sensor for variable rate N application'. In: *Crop and Pasture Science* 60.9 (2009), pp. 818–823.

McCasland, Margaret, Nancy M. Trautmann and Robert J. Wagenet. *Nitrate: Health effects in drinking water*. 1985. URL: `https://psep.cce.cornell.edu/facts-slides-self/facts/nit-heef-grw85.aspx`.

Milioto, Andres, Philipp Lottes and Cyrill Stachniss. 'Real-time blob-wise sugar beets vs weeds classification for monitoring fields using convolutional neural networks'. In: *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 4 (2017), p. 41.

Monson, Robert J. and Norman A. Bauer. *Variable rate application system*. Nov. 1995.

Muñoz-Huerta, Rafael F., Ramon G. Guevara-Gonzalez, Luis M. Contreras-Medina, Irineo Torres-Pacheco, Juan Prado-Olivarez and Rosalia V. Ocampo-Velazquez. 'A review of methods for sensing the nitrogen status in plants: advantages, disadvantages and recent advances'. In: *Sensors* 13.8 (2013), pp. 10823–10843.

Murray, RI and IJ Yule. 'Developing variable rate application technology: economic impact for farm owners and topdressing operators'. In: *New Zealand Journal of Agricultural Research* 50.1 (2007), pp. 65–72.

Mutanga, Onisimo and Andrew K. Skidmore. 'Red edge shift and biochemical content in grass canopies'. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 62.1 (2007), pp. 34–42. ISSN: 0924-2716. DOI: https://doi.org/10.1016/j.isprsjprs.2007.02.001. URL: http://www.sciencedirect.com/science/article/pii/S0924271607000068.

NetBSD Foundation. *Announcing NetBSD 5.0*. 29 Apr. 2009. URL: https://web.archive.org/web/20180622111449/https://www.netbsd.org/releases/formal-5/NetBSD-5.0.html.

Nguyen, Huu-Quoc, Ton Thi Kim Loan, Bui Dinh Mao and Eui-Nam Huh. 'Low cost real-time system monitoring using Raspberry Pi'. In: *2015 Seventh International Conference on Ubiquitous and Future Networks*. July 2015, pp. 857–859.

Nishihara, Robert, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith et al. 'Real-Time Machine Learning: The Missing Pieces'. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. New York, NY, USA: ACM, 2017, pp. 106–110. ISBN: 978-1-4503-5068-6.

Nissen, Steffen. 'Implementation of a fast artificial neural network library (fann)'. In: *Report, Department of Computer Science University of Copenhagen (DIKU)* 31 (2003), p. 29.

Nomi, Taiga. *tiny-dnn: header only, dependency-free deep learning framework in C++14*. GitHub, 2012.

NVIDIA Corporation. *NVIDIA TESLA V100 GPU ACCELERATOR*. Mar. 2018. URL: https://web.archive.org/web/20180815052540/https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf.

Olofsson, Andreas. 'Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip'. In: *arXiv:1610.01832 [cs]* (Oct. 2016). URL: http://arxiv.org/abs/1610.01832 (visited on 30/11/2018).

Olofsson, Andreas, Tomas Nordström and Zain Ul-Abdin. 'Kickstarting high-performance energy-efficient manycore architectures with Epiphany'. In: *48th Asilomar Conference on Signals, Systems and Computers*. IEEE, 2014, pp. 1719–1726. ISBN: 1-4799-8295-4.

Olofsson, Andreas, Roman Trogan, Oleg Raikhman and Lexington Adapteva. 'A 1024-core 70 GFLOP/W floating point manycore microprocessor'. In: *Poster on 15th Workshop on High Performance Embedded Computing HPEC2011*. 2011.

Ortmeyer, C. 'Then and now: a brief history of single board computers'. In: *Electron. Des. Uncovered* 6 (2014), pp. 1–11.

Owechko, Yuri and S Shams. 'Comparison of neural network and genetic algorithms for a resource allocation problem'. In: *1994 IEEE International Conference on Neural Networks*. Jan. 1994, 4655–4660 vol.7. ISBN: 0-7803-1901-X. DOI: 10.1109/ICNN.1994.375027.

Pantazi, X. E., D. Moshou, T. Alexandridis, R. L. Whetton and A. M. Mouazen. 'Wheat yield prediction using machine learning and advanced sensing techniques'. In: *Computers and Electronics in Agriculture* 121 (Feb. 2016), pp. 57–65. ISSN: 0168-1699.

Pedersen, S. M., S. Fountas, H. Have and B. S. Blackmore. 'Agricultural robots—system analysis and economic feasibility'. In: *Precision Agriculture* 7.4 (Sept. 2006), pp. 295–308. ISSN: 1573-1618.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel et al. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

Queensland Government. *Environmental Protection Act 1994*. Oct. 1994.

Qureshi, W. S., A. Payne, K. B. Walsh, R. Linker, O. Cohen and M. N. Dailey. 'Machine vision for counting fruit on mango tree canopies'. In: *Precision Agriculture* 18.2 (Apr. 2017), pp. 224–244. ISSN: 1573-1618.

Ravishankara, A. R., John S. Daniel and Robert W. Portmann. 'Nitrous Oxide (N_2O): The Dominant Ozone-Depleting Substance Emitted in the 21st Century'. In: *Science* 326.5949 (2009), pp. 123–125. ISSN: 0036-8075.

Rhoades, J. D., P. A. C. Raats and R. J. Prather. 'Effects of Liquid-phase Electrical Conductivity, Water Content, and Surface Conductivity on Bulk Soil Electrical Conductivity'. In: *Soil Science Society of America Journal* 40.5 (1976), pp. 651–655.

Robertson, Michael, Peter Carberry and Lisa Brennan. 'Economic benefits of variable rate technology: case studies from Australian grain farms'. In: *Crop and Pasture Science* 60.9 (2009), pp. 799–807.

Robson, Andrew, Muhammad Moshiur Rahman, Gregory Falzon, Niva Kiran Verma, Kasper Johansen, Nicole Robinson et al. 'Evaluating remote sensing technologies for improved yield forecasting and for the measurement of foliar nitrogen concentration in sugarcane'. In: *International Sugar Journal* 118.1416 (2016), pp. 936–942. ISSN: 0020-8841.

Romin, Andreas. 'Locks and raspberries: a comparative study of single-board computers for access control'. MA thesis. Uppsala University, Solid State Electronics, 2016, p. 47.

Rouse Jr., J. W., R. H. Haas, J. A. Schell and D. W. Deering. 'Monitoring Vegetation Systems in the Great Plains with Erts'. In: *NASA Special Publication* 351 (1974), p. 309.

Rowbot Systems. *Rowbot*. English. June 2017. (Visited on 20/06/2017).

Sadgrove, Edmund J., Greg Falzon, David Miron and David Lamb. 'Fast object detection in pastoral landscapes using a Colour Feature Extreme Learning Machine'. In: *Computers and Electronics in Agriculture* 139 (2017), pp. 204–212. ISSN: 0168-1699.

Sametinger, Johannes, Jerzy Rozenblit, Roman Lysecky and Peter Ott. 'Security Challenges for Medical Devices'. In: *Communications of the ACM* 58.4 (Mar. 2015), pp. 74–82. ISSN: 0001-0782.

Sangiovanni-Vincentelli, A. and G. Martin. 'Platform-based design and software design methodology for embedded systems'. In: *IEEE Design & Test of Computers* 18.6 (Dec. 2001), pp. 23–33. ISSN: 0740-7475.

Sawyer, J. E. 'Concepts of Variable Rate Technology with Considerations for Fertilizer Application'. English. In: *Journal of Production Agriculture* 7.2 (1994), pp. 195–201.

Scharf, Peter C., D. Kent Shannon, Harlan L. Palm, Kenneth A. Sudduth, Scott T. Drummond, Newell R. Kitchen et al. 'Sensor-Based Nitrogen Applications Out-Performed Producer-Chosen Rates for Corn in On-Farm Demonstrations'. English. In: *Agronomy Journal* 103.6 (2011), pp. 1683–1691.

Schaul, Tom, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke et al. 'PyBrain'. In: *Journal of Machine Learning Research* 11 (2010), pp. 743–746.

Schmidt, D. C. 'Guest Editor's Introduction: Model-Driven Engineering'. en. In: *Computer* 39.2 (Feb. 2006), pp. 25–31. ISSN: 0018-9162.

Schroeder, B. L., A. W. Wood, P. W. Moody and J. H. Panitz. 'Sustainable nutrient management–delivering the message to the Australian sugar industry'. In: *Proc S Afr Sug Technol Ass*. Vol. 79. 2005, p. 206.

Schwartz, Dominique. *The World Today. Great Barrier Reef health can be improved by cane farmers' herbicide, fertiliser run-off reduction*. Sydney: ABC Radio, 22 Feb. 2016. URL: https://www.abc.net.au/radio/programs/worldtoday/great-barrier-reef-health-can-be-improved-by-cane/7189362.

Selic, Bran. 'Real-Time Object-Oriented Modeling'. In: *IFAC Proceedings Volumes* 29.5 (1996), pp. 1–6. ISSN: 1474-6670". DOI: 10.1016/S1474-6670(17)46346-4. URL: http://www.sciencedirect.com/science/article/pii/S1474667017463464.

Shamshiri, Redmond Ramin, Ibrahim A Hameed, Lenka Pitonakova, Cornelia Weltzien, Siva K Balasundram, Ian J Yule et al. 'Simulation software and virtual environments for acceleration of agricultural robotics: Features highlights and performance comparison'. In: *International Journal of Agricultural and Biological Engineering* 11.4 (2018), pp. 15–31.

Shi, Wenzhe, Jose Caballero, Ferenc Huszar, Johannes Totz, Andrew P. Aitken, Rob Bishop et al. 'Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network'. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

Simonyan, Karen and Andrew Zisserman. 'Very Deep Convolutional Networks for Large-Scale Image Recognition'. In: *CoRR* abs/1409.1556 (2014).

Snoeck, M. and G. Dedene. 'Experiences with object oriented model-driven development'. In: *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering*. July 1997, pp. 143–153. DOI: 10.1109/STEP.1997.615473.

Sousa, Paulo Baltarejo, Nuno Pereira and Eduardo Tovar. 'Enhancing the Real-time Capabilities of the Linux Kernel'. In: *SIGBED Rev.* 9.4 (Nov. 2012), pp. 45–48. ISSN: 1551-3688.

Srivastava, Ashok N. and Johann Schumann. 'Software health management: a necessity for safety critical systems'. In: *Innovations in Systems and Software Engineering* 9.4 (Dec. 2013), pp. 219–233. ISSN: 1614-5054.

Stover, Joshua. 'Model checking of Finite-State Machines which implement feedback control loops in embedded systems'. Honours Thesis. Griffith University, 2015.

Stover, Joshua, Greg Falzon, Troy Jensen and Bernard Schroeder. 'Hardware and Embedded Algorithms for Real Time Variable Rate Fertiliser Applications'. In: *The International Tri-Conference for Precision Agriculture*. Hamilton, New Zealand, Oct. 2017.

SWARM FARM. *SwarmFarm Robotics*. en-US. 2018. (Visited on 11/01/2018).

The European Commision. *Periodic Reporting for period 1 - ASTERIX*. ASTERIX Report Summary 198304. The European Commision, May 2017, p. 3. (Visited on 28/01/2018).

Thorburn, P. J., A. J. Webster, I. M. Biggs, J. S. Biggs, S. E. Park and M. F. Spillman. 'Towards innovative management of nitrogen fertiliser for a sustainable sugar industry'. In: *Proceedings of the Australian Society of Sugar Cane Technologists Conference*. Vol. 29. May 2007, pp. 85–96.

Thorburn, P. J., A. J. Webster and J. S. Biggs. 'Nitrogen balances in sugarcane farming systems as affected by nitrogen fertiliser applications'. In: *Proceedings of the 2008 Conference of the Australian Society of Sugar Cane Technologists Held at Townsville, Queensland, Australia*. Vol. 29. 2008, pp. 357–358.

Thorburn, Peter J., Sarah E. Park and I. M. Biggs. 'Nitrogen fertiliser management in the Australian sugar industry: strategic opportunities for improved efficiency'. In: *Proceedings of the Australian Society of Sugar Cane Technologists Conference*. PK Editorial Services Pty Ltd. Townsville, Queensland, Australia: PK Editorial Services; 1999, May 2003, pp. 22–22.

Topalidou-Kyniazopoulou, Angeliki, Nikolaos I. Spanoudakis and Michail G. Lagoudakis. 'A CASE Tool for Robot Behavior Development'. In: *RoboCup 2012: Robot Soccer World Cup XVI*. Ed. by Xiaoping Chen, Peter Stone, Luis Enrique Sucar and Tijn Zant. Vol. 7500. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 225–236. ISBN: 978-3-642-39249-8. (Visited on 26/02/2018).

Townsend, Alan R, Robert W Howarth, Fakhri A Bazzaz, Mary S Booth, Cory C Cleveland, Sharon K Collinge et al. 'Human health effects of a changing global nitrogen cycle'. In: *Frontiers in Ecology and the Environment* 1.5 (2003), pp. 240–246. ISSN: 1540-9309.

trends.google.com. *Google Trends*. 2018. URL: `https://trends.google.com/trends/explore?date=all&q=%2Fm%2F0djmwv,%2Fm%2F0gmg36g,%2Fm%2F02ytr,%2Fm%2F0pk4w,%2Fm%2F0196qx`.

Trotter, M. G., D. W. Lamb, G. E. Donald and D. A. Schneider. 'Evaluating an active optical sensor for quantifying and mapping green herbage mass and growth in a perennial grass pasture'. In: *Crop and Pasture Science* 61.5 (2010), pp. 389–398. ISSN: 1836-5795.

Truell, Michael and Joshua Gruenstein. 'A Universal Robot Control System using Reinforcement Learning with Limited Feedback'. 2016.

Tsang, Ivor W., James T. Kwok and Pak-Ming Cheung. 'Core Vector Machines: Fast SVM Training on Very Large Data Sets'. In: *J. Mach. Learn. Res.* 6 (Dec. 2005), pp. 363–392. ISSN: 1532-4435. URL: `http://dl.acm.org/citation.cfm?id=1046920.1058114`.

Unkovich, Murray, David Herridge, Mark Peoples, George Cadisch, B. Boddey, K. Giller et al. *Measuring plant-associated nitrogen fixation in agricultural systems.* Australian Centre for International Agricultural Research (ACIAR), 2008. ISBN: 1-921531-26-6.

US Air Force Research Laboratory. *Modeling & Analysis of Multicore Architectures for Embedded SIGINT Applications.* Tech. rep. Air Force Research Lab Rome, NY High Performance Systems Branch, 2015.

Vapnik, Vladimir. *The Nature of Statistical Learning Theory.* Springer Science & Business Media, 2013.

Vladislavs, Dovgalecs. *ELM-C–: Extreme Learning Machine - C++ library.* GitHub, 2013.

Ward, Mary H., Theo M. deKok, Patrick Levallois, Jean Brender, Gabriel Gulis, Bernard T. Nolan et al. 'Workgroup Report: Drinking-Water Nitrate and Health—Recent Findings and Research Needs'. en. In: *Environmental Health Perspectives* 113.11 (June 2005), pp. 1607–1614. ISSN: 0091-6765. (Visited on 18/11/2018).

Wolf, Wayne H. 'Hardware-software co-design of embedded systems'. In: *Proceedings of the IEEE* 82.7 (1994), pp. 967–989. ISSN: 0018-9219.

Wood, A. W., B. L. Schroeder, A. P. Hurney, B. Salter and J. H. Panitz. 'Research aimed at enhancing nitrogen management guidelines for the SIX EASY STEPS program'. In: *Proc. Aust. Sugar Cane Technol.* Vol. 30. 2008, pp. 362–363.

Wood, S. K., D. H. Akehurst, O. Uzenkov, W. G. J. Howells and K. D. McDonald-Maier. 'A Model-Driven Development Approach to Mapping UML State Diagrams to Synthesizable VHDL'. In: *IEEE Transactions on Computers* 57.10 (Oct. 2008), pp. 1357–1371. ISSN: 0018-9340.

Yang, Chun-Chieh, Shiv O. Prasher and Guy R. Mehuys. 'An artificial neural network to estimate soil temperature'. In: *Canadian Journal of Soil Science* 77.3 (Aug. 1997), pp. 421–429. ISSN: 0008-4271.

Yu, Dong and Li Deng. *Automatic Speech Recognition: A deep learning approach.* Signals and Communication Technology. London: Springer London, 2015. ISBN: 978-1-4471-5778-6. (Visited on 10/02/2018).

Zachman, J. A. 'A framework for information systems architecture'. In: *IBM Systems Journal* 26 (3 1987). DOI: 10.1147/sj.263.0276. URL: http://gen.lib.rus.ec/scimag/index.php?s=10.1147/sj.263.0276.

Appendix A

Source Code

Listing 1: Fuzzy Boxes: ___init___.py

```
from .feature import FuzzyBoxFeature as fb_feature
from .model import Ratio as fb_ratio
from .model import FuzzyBoxModel as fb_model
from .instance import FuzzyBoxInstance as fb_instance
from .instance import FuzzyBoxInstanceValue as fb_value

FuzzyBoxFeature = fb_feature
FuzzyBoxRatio = fb_ratio
FuzzyBoxModel = fb_model
FuzzyBoxInstance = fb_instance
FuzzyBoxInstanceValue = fb_value
```

Listing 2: Fuzzy Boxes: exceptions.py

```
class FuzzyBoxException(Exception):
    """Standard Exception for errors raised by the fuzzyboxes module"""


class FuzzyBoxDataError(FuzzyBoxException):
    """Error in the structure or formatting of FuzzyBox data"""


class FuzzyBoxDataTypeError(FuzzyBoxException):
    """Incorrect data type"""


class FuzzyBoxFeatureError(FuzzyBoxException):
    """Something went wrong with a FuzzyBoxFeature instance"""
    def __init__(self, feature, msg):
        msg = "Error in feature [%s]: %s" % (feature.name, msg)
        super(FuzzyBoxFeatureError, self).__init__(msg)
        self.feature = feature


class FuzzyBoxModelError(FuzzyBoxException):
    """Something went wrong relating to an instance of FuzzyBoxModel"""
```

Listing 3: Fuzzy Boxes: feature.py

```
class FuzzyBoxFeature(object):
    def __init__(self, name, values=None):
        """
        Feature component of a FuzzyBoxes model

        Args:
            name (str):     human-readable name of the feature

        Attributes:
            split_points:   indexed list containing splitpoint arrays for boxes with N=(i+1)
            values:         feature values which were set during initialization
        """
        self.name = name
        self.split_points = []
        self._values = values

    values = property(lambda s: s._values)

    def __repr__(self):
        return "<Feature %r: name=%s splits=%r>" % (id(self), self.name, len(self.split_points))
```

Listing 4: Fuzzy Boxes: functions.py

```
#!/usr/bin/env python
# vim:fileencoding=utf-8
import math
import csv
from .instance import FuzzyBoxInstance


def percentile(values, n):
    """
    Calculate and return the nth percentile of the array values given

    Args:
            values  (list)      input array of values
            n       (float)     Percentile to calculate, must be in range 0-100 (inclusive)
    """

    if not values:
        return None

    values = sorted(values)

    if n > 100 or n < 0:
        raise ValueError("Percentile argument (n) must be in range 0-100")

    n /= 100.0

    k = (len(values)-1) * n
    f = math.floor(k)
```

```
29      c = math.ceil(k)
30      if f == c:
31          return values[int(k)]
32      else:
33          return values[int(f)] * (c-k) + values[int(c)] * (k-f)
34
35
36  def initialise_array(dimensions, value=0, level=0):
37      """
38      Create a multidimensional array (list) and initialise all values
39
40      Args:
41              dimensions  (seq)   iterable containing the size of the new array
42              value       (any)   value to initialise each element of the array, may be a callable
43
44      Example:
45
46          initialise_array([2, 2, 2], lambda: object())
47              This will return an array or size 2x2x2. All elements will be initialised as new
    ↪  objects
48
49      """
50      if level >= len(dimensions):
51          return value() if callable(value) else value
52      else:
53          return [initialise_array(dimensions, value, level + 1) for i in range(dimensions[level])]
54
55
56  def read_csv(filename, fieldnames=None):
57      """
58      Generator wrapper around the standard csv python library.
59      This will read a csv file and yield a list object. Header row is discarded"
60      """
61      with open(filename, 'r') as f:
62          sample = f.read(1024)
63          f.seek(0)
64          dialect = csv.Sniffer().sniff(sample)
65          reader = csv.reader(f, dialect)
66          if csv.Sniffer().has_header(sample):
67              next(reader)
68          for row in reader:
69              if len(row) < 2:
70                  raise Exception("Cannot convert from CSV row to FuzzyBoxInstance: %r" % row)
71              row_values = list(map(float, row[1:]))
72              row_class = bool(int(row[0]))
73              yield FuzzyBoxInstance(values=row_values, classification=row_class)
74
75
76  if __name__ == '__main__':
77      import os
78      import tempfile
79      import random
80      import unittest
81      import numpy as np
82
83      class UnitTestPercentile(unittest.TestCase):
84
```

```
        def setUp(self):
            pass

        def testErrorConditions(self):
            data = [random.randint(0, 200) for _ in range(24)]
            self.assertIsNone(percentile(False, 0.5))
            self.assertRaises(ValueError, lambda: percentile(data, -1))
            self.assertRaises(ValueError, lambda: percentile(data, 101))

        def testAgainstNumpy(self):
            percentiles = [x/100.0 for x in range(10001)]
            random.seed()
            data = [random.randint(0, 200) for _ in range(24)]
            for p in percentiles:
                self.assertEqual(percentile(data, p), np.percentile(data, p))

        def testBasicMath(self):
            data = range(10)
            self.assertEqual(percentile(data, 0), 0)
            self.assertEqual(percentile(data, 50), 4.5)
            self.assertEqual(percentile(data, 100), 9)
            data = range(1, 11)
            self.assertEqual(percentile(data, 0), 1.0)
            self.assertEqual(percentile(data, 50), 5.5)
            self.assertEqual(percentile(data, 100), 10.0)

    class UnitTestEmptyArray(unittest.TestCase):
        def setUp(self):
            pass

        def testDimensions(self):
            value = 77
            cube3 = initialise_array([3, 3, 3], value)
            self.assertEqual(len(cube3), 3)
            self.assertEqual(len(cube3[0]), 3)
            self.assertEqual(len(cube3[0][0]), 3)
            self.assertEqual(cube3[0][0][0], value)
            n = 0
            for i in cube3:
                for j in i:
                    for k in j:
                        n += 1
                        self.assertEqual(k, value)
            self.assertEqual(n, 27)
            for i, _ in enumerate(cube3):
                for j, _ in enumerate(cube3[i]):
                    for k, _ in enumerate(cube3[i][j]):
                        self.assertEqual(cube3[i][j][k], value)
            self.assertRaises(IndexError, lambda: cube3[3])
            self.assertRaises(IndexError, lambda: cube3[2][3])
            self.assertRaises(IndexError, lambda: cube3[2][2][3])

        def testSingle(self):
            value = 129
            single = initialise_array([10], value)
            self.assertEqual(len(single), 10)
            self.assertEqual(single[9], value)
```

```
            self.assertRaises(IndexError, lambda: single[10])

        def testObject(self):
            dims = [7, 7, 7]
            objArray = initialise_array(dims, value=lambda: object())
            objs = []
            for i, _ in enumerate(objArray):
                for j, _ in enumerate(objArray[i]):
                    for k, _ in enumerate(objArray[i][j]):
                        objs.append(id(objArray[i][j][k]))
            self.assertEqual(len(objs), reduce(lambda a, b: a*b, dims))
            self.assertEqual(len(set(objs)), reduce(lambda a, b: a*b, dims))

    class UnitTestInput(unittest.TestCase):
        def setUp(self):
            self.csvfiles = {}
            self.csvfiles['plain'] = tempfile.NamedTemporaryFile(delete=False)
            self.csvfiles['header'] = tempfile.NamedTemporaryFile(delete=False)
            with open(self.csvfiles['plain'].name, 'w') as f:
                self._write_csv(f)
            with open(self.csvfiles['header'].name, 'w') as f:
                self._write_csv_header(f)
                self._write_csv(f)

        def tearDown(self):
            for i, csvfile in self.csvfiles.items():
                if os.path.isfile(csvfile.name):
                    os.remove(csvfile)

        def _write_csv_header(self, f):
            f.write(', '.join([chr(c) for c in range(65, 75)]))
            f.write("\n")

        def _write_csv(self, f):
            for x in range(1, 5):
                f.write(','.join(str(n) for n in range(10*x, 10*x+10)))
                f.write("\n")

        def testTestFiles(self):
            for label in self.csvfiles:
                self.assertTrue(os.path.isfile(self.csvfiles[label].name))

        def testInstanceReader(self):
            for i, csvfile in self.csvfiles.items():
                for row in read_csv(csvfile.name):
                    self.assertIsInstance(row, FuzzyBoxInstance)

    unittest.main()
```

Listing 5: Fuzzy Boxes: graphics.py

```
from .exceptions import FuzzyBoxModelError


def scatter(model, feature_x=None, feature_y=None, labels=None):
    if not model.isTrained:
        raise FuzzyBoxModelError("Model has not been trained")
    import matplotlib.pyplot as pyplot

    if feature_x is None:
        feature_x = model.training_features[0]
    if feature_y is None:
        feature_y = model.training_features[-1]

    fig = pyplot.figure(1, figsize=(18, 9))
    x_min = min(feature_x.values)
    x_max = max(feature_x.values)
    x_buffer = (x_max - x_min) * 0.05
    x1 = [x for i, x in enumerate(feature_x.values) if model.training_classes[i]]
    x2 = [x for i, x in enumerate(feature_x.values) if not model.training_classes[i]]
    y_min = min(feature_y.values)
    y_max = max(feature_y.values)
    y_buffer = (y_max - y_min) * 0.05
    y1 = [y for i, y in enumerate(feature_y.values) if model.training_classes[i]]
    y2 = [y for i, y in enumerate(feature_y.values) if not model.training_classes[i]]

    for S in range(min(model.Smax, 8)):
        ax = fig.add_subplot(240 + S + 1, adjustable='box-forced', aspect='auto')
        ax.set_title("S = %d" % S)
        scatterPositive = ax.scatter(x1, y1, c='green', alpha=0.5, label='Positive')
        scatterNegative = ax.scatter(x2, y2, c='red', alpha=0.5, label='Negative')
        ax.axis([x_min - x_buffer, x_max + x_buffer, y_min - y_buffer, y_max + y_buffer])
        ax.grid(True)
        ax.yaxis.set_ticks(feature_y.split_points[S])
        ax.xaxis.set_ticks(feature_x.split_points[S])
        pyplot.setp(ax.get_xticklabels(), rotation=30, ha='right')
        ax.tick_params(axis='both', which='major', labelsize=8)
        ax.tick_params(axis='both', which='minor', labelsize=8)
    if labels is not None and len(labels) == 2:
        fig.figlegend((scatterPositive, scatterNegative), labels, 'lower left', ncol=2)
    fig.subplots_adjust(0.03, 0.10, 0.97, 0.95, 0.2, 0.2)

    return fig


def heatmap(model, feature_x=None, feature_y=None, S=None):
    if not model.isTrained:
        raise FuzzyBoxModelError("Model has not been trained")
    import matplotlib as mpl
    import matplotlib.pyplot as pyplot
    from matplotlib.ticker import AutoMinorLocator

    if feature_x is None:
        feature_x = model.training_features[0]
```

```
    if feature_y is None:
        feature_y = model.training_features[-1]
    if S is None:
        S = -1

    fig, ax = pyplot.subplots()
    data = []
    for row in model.box_weights[S]:
        float_row = []
        for w in reversed(row):
            float_row.append(w.value)
        data.append(float_row)

    def _rotateMatrix(M):
        return zip(*M[::-1])


    data = _rotateMatrix(_rotateMatrix(_rotateMatrix(data)))
    ax.pcolor(data, cmap=pyplot.cm.Blues, alpha=1)
    ax.axis([0, S + 1, 0, S + 1])
    ax.set_xticks(range(S + 2))
    ax.set_xticklabels([])
    ax.set_yticks(range(S + 2))
    ax.set_yticklabels([])

    fig.suptitle("Fuzzy Box Model")
    ax.set_xlabel("RNDVI")
    ax.set_ylabel("EM38")

    ax.text(0.00, -0.04, '0.125',
        verticalalignment='bottom', horizontalalignment='left',
        transform=ax.transAxes, color='black', fontsize=10)
    ax.text(1.00, -0.04, '0.711',
        verticalalignment='bottom', horizontalalignment='left',
        transform=ax.transAxes, color='black', fontsize=10)

    ax.text(0.00, 0.00, '36.4',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes, color='black', fontsize=10)
    ax.text(0.00, 1.00, '103.0',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes, color='black', fontsize=10)

    ax.grid(which='both', axis='both')

    return fig
```

Listing 6: Fuzzy Boxes: instance.py

```
class FuzzyBoxInstance(object):
    def __init__(self, values=[], classification=False):
        """
        Instance object for the Fuzzy Boxes model. Similar to a dataframe row,
        stores a value for each feature, as well as a classification value.

        Args:
            values:         iterable containing a single for each feature
            classification: value indicating the classification of this instance.
        """
        self.values = values
        self.classification = classification

    def __repr__(self):
        return "%s: %s" % (str(self.values), str(self.classification))

    def __eq__(self, other):
        return self.values == other.values

    def __neq__(self, other):
        return not self.__eq__(other)

    def __len__(self):
        return len(self.values)

    def get_position(self, S):
        """Return the position tuple for S based on the position vectors for each value"""
        if len(self.values) > 0 and all(isinstance(v, FuzzyBoxInstanceValue) for v in
        ↪  self.values):
            return tuple([self.values[fi].positions[S] for fi in range(len(self.values))])


class FuzzyBoxInstanceValue(float):
    def __new__(cls, value=0):
        i = float.__new__(cls, value)
        i.positions = []
        return i
```

Listing 7: Fuzzy Boxes: model.py

```
import copy
from .instance import FuzzyBoxInstance, FuzzyBoxInstanceValue
from .feature import FuzzyBoxFeature
from .exceptions import FuzzyBoxException, FuzzyBoxFeatureError
from .functions import percentile, initialise_array


class Ratio(object):
    def __init__(self):
        """Ratio of positive instance to negative instances.
            This holds the fuzzy weighting for each multidimensional 'box'

        Attributes:
            total:      (int)   number of total instances added
            positive:   (int)   number of positive instance added
            value:      (float) floating point representation of the ratio
        """
        self.total = 0
        self.positive = 0
        self._value_changed = False
        self._value = -1

    @property
    def value(self):
        """Get the floating point representation of the"""
        if self._value_changed:
            self._value = self.positive / float(self.total)
        return -1 if self.total == 0 else self._value

    def add(self, positive):
        self.total += 1
        self.positive += 1 if positive else 0
        self._value_changed = True

    def __repr__(self):
        value = self.value
        if value >= 0:
            return "%3d/%3d (%2.2f)" % (self.positive, self.total, value)
        else:
            return "%3d/%3d   --- " % (self.positive, self.total)

    def __str__(self):
        value = self.value
        if value >= 0:
            return "%0.2f" % self.value
        return "None"

    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return self.total == other.total and self.positive == other.positive
        return False

    def __neq__(self, other):
```

```
54          return not self.__eq__(other)
55
56
57  class FuzzyBoxModel(object):
58      def __init__(self, Smax, method, prior_ratio=None):
59          """Fuzzy Boxes statistical learning model.
60
61          Args:
62              Smax    (int)       Maximum number of split points to consider
63              method  (str)       Boxing method to use for split point calculation.
64                                  Can be 'even' or 'percentile'
65              prior   (float)     Prior ratio used for classifications
66
67          Attributes:
68              box_weights:    ()
69          """
70          self.Smax = Smax
71          self.method = method
72          self.prior_ratio = prior_ratio
73          self.box_weights = []
74
75          self.training_features = []
76          self.training_instances = []
77          self.training_classes = []
78
79          self.number_of_features = 0
80
81      isTrained = property(lambda s: bool(len(s.training_instances) > 0))
82
83      def __repr__(self):
84          return "<FuzzyBoxModel:%s Smax=%d, method=%s, prior=%f>" % (id(self), self.Smax,
            ↪  self.method, self.prior_ratio)
85
86      def train(self, training_features, training_classes, stopwatch):
87          """  Train the model
88
89          Args:
90              features    list[FuzzyBoxFeature]   Features to train the model on
91              classes     list[bool]              Class for each instance of feature values
92
93          """
94          # check data sanity
95          for f in training_features:
96
97              if not isinstance(f, FuzzyBoxFeature):
98                  raise FuzzyBoxException("Expected only feature objects of type FuzzyBoxFeature")
99
100             if f.values is None:
101                 raise FuzzyBoxFeatureError(f, "No training values available")
102
103             if len(f.values) != len(training_classes):
104                 raise FuzzyBoxFeatureError(f, "Instance count mismatch with training_classes
                    ↪  data")
105
106         # retrieve a list of FBInstance objects from the values of the training data
107         def _getFBInstances():
108             for i, _ in enumerate(self.training_classes):
```

```
109                 inst_values = [FuzzyBoxInstanceValue(f.values[i]) for f in self.training_features]
110                 yield FuzzyBoxInstance(inst_values)
111
112         # store training data
113         self.training_features = copy.deepcopy(training_features)
114         self.training_classes = [bool(c) for c in training_classes]
115         self.training_instances = [i for i in _getFBInstances()]
116         self.box_weights = []
117         self.number_of_features = len(training_features)
118
119         self.total_boxes = 0
120         for N in range(1, self.Smax + 2):
121             s_boxes = pow(N - 1, len(self.training_features))
122             self.total_boxes += s_boxes
123             self.box_weights.append(initialise_array([N] * len(self.training_features), lambda:
                ↪  Ratio()))
124
125         if stopwatch is not None:
126             stopwatch.timer_tick('train-internal')
127
128         # guess prior ratio to be equal to training if not given as a parameter
129         if self.prior_ratio is None:
130             self.prior_ratio = sum(self.training_classes) / float(len(self.training_classes))
131
132         # Training Start
133         for feature_index, feature in enumerate(self.training_features):
134
135             # precalculate feature min/max/range for this feature
136             ft_min = min(feature.values)
137             ft_max = max(feature.values)
138             ft_range = ft_max - ft_min
139
140             # S is number of split points
141             for S in range(self.Smax + 1):
142
143                 # calculate split points
144                 splits = []
145                 if self.method == 'even':
146                     for i in range(S):
147                         sp = ft_min + (i + 1) * ft_range / float(S + 1)
148                         splits.append(sp)
149                 elif self.method == 'percentile':
150                     for i in range(S):
151                         perc = (i + 1) / float(S + 1) * 100
152                         sp = percentile(feature.values, perc)
153                         splits.append(sp)
154                 else:
155                     raise Exception("Unknown boxing method: %s" % self.method)
156
157                 feature.split_points.append(splits)
158
159                 # find box position for each instance for this feature / S
160                 for instance in self.training_instances:
161                     box = 0
162                     for boundary in feature.split_points[S]:
163                         if (instance.values[feature_index] <= boundary) or (boundary is None):
164                             break
```

```
                        box += 1
                    instance.values[feature_index].positions.append(box)

        # calculate box weights
        for S in range(self.Smax + 1):
            for i, instance in enumerate(self.training_instances):
                positions = instance.get_position(S)
                box = self.box_weights[S]
                for p in positions:
                    box = box[p]
                box.add(self.training_classes[i])

        if stopwatch is not None:
            stopwatch.timer_tock('train-internal')

    def predict(self, values, stopwatch, max_depth=None):
        """ Predict the classification of the values from the trained model.
            The order of the values is assumed to match the order of the training vectors

            values  (list[float])   values to predict
        """
        if not self.isTrained:
            raise FuzzyBoxException("Model has not been trained")

        if len(values) != len(self.training_features):
            raise FuzzyBoxException("Mismatch in feature count")

        if max_depth is None:
            max_depth = self.Smax

        # convert to fuzzyboxes objects
        values = [FuzzyBoxInstanceValue(v) for v in values]
        instance = FuzzyBoxInstance(values, classification=None)

        value_sum = 0.0
        value_count = 0

        # Range of split points 1-S
        # s_range = [s for s in range(self.Smax + 1)]

        # Range of feature indices 1-F
        # fi_range = [fi for fi in range(self.number_of_features)]

        # Generate empty matrix of box positions size = SxF
        instance_positions = [[None] * self.number_of_features] * (self.Smax + 1)

        # Pull split points out to cut down indirection
        splits = [f.split_points for f in self.training_features]

        S = 0
        S_max = max_depth + 1
        F_max = self.number_of_features
        box = None

        stopwatch.timer_tick('predict-internal')

        while S < S_max:
```

```
            F = 0
            while F < F_max:

                box = 0
                for boundary in splits[F][S]:
                    if (instance.values[F] <= boundary) or (boundary is None):
                        break
                    box += 1
                instance_positions[S][F] = box
                F += 1

            weight = self.box_weights[S]
            for p in instance_positions[S]:
                weight = weight[p]

            if weight.value >= 0:
                value_sum += weight.value
                value_count += 1

            S += 1
        predict_class = None if value_count <= 0 else (value_sum / value_count) > self.prior_ratio
        stopwatch.timer_tock('predict-internal')
        return predict_class

    def save(self):
        pass

    def load(self):
        pass

    def details(self):
        details = {
            'max_boxes': self.Smax + 1,
            'total_boxes': self.total_boxes,
            'boxing_method': self.method,
            'trained': self.isTrained,
            'features': [],
            'box_weights': self.box_weights,
            'training': {},
        }
        for feature in self.training_features:
            f = {
                'name': feature.name,
                'range': (min(feature.values), max(feature.values)),
                'split_points': {},
            }

            for S in range(self.Smax + 1):
                f['split_points'][S] = feature.split_points[S]
            details['features'].append(f)

        for instance in self.training_instances:
            _positions = [str(instance.get_position(S)) for S in range(self.Smax + 1)]
            details['training'][str(instance)] = _positions
        return details
```

Listing 8: Fuzzy Boxes: results.py

```
# vim:fileencoding=utf-8
import math
import time


class Results(object):
    def __init__(self):
        self.tp = 0.0
        self.fp = 0.0
        self.tn = 0.0
        self.fn = 0.0
        self.timing = {
            'training': 0.0,
            'testing': 0.0,
            'intervals': 0,
        }
        self.time_start = 0
        self.time_interval = 0
        self.intervals = 0

    def reset_timer(self):
        self.time_start = time.clock()
        self.time_interval = time.clock()
        self.intervals = 0

    def record_time(self, name):
        self.timing[name] = time.clock() - self.time_start

    def start_interval(self):
        self.interval_time = time.clock()

    def add_interval(self, name):
        self.timing[name] += time.clock() - self.time_interval
        self.time_interval = time.clock()
        self.intervals += 1

    def average_interval(self, name):
        self.timing[name] = self.timing[name] / self.intervals

    def add(self, predicted, actual):
        if predicted is True and actual is True:
            self.tp += 1
        if predicted is False and actual is False:
            self.tn += 1
        if predicted is True and actual is False:
            self.fp += 1
        if predicted is False and actual is True:
            self.fn += 1

    def total(self):
        return self.tp + self.tn + self.fp + self.fn

    def tpr(self):
        return self.tp / (self.tp + self.fn)

```

```
56      def fpr(self):
57          return self.fp / (self.fp + self.tn)
58
59      def tnr(self):
60          return self.tn / (self.tn + self.fp)
61
62      def fnr(self):
63          return self.fn / (self.fn + self.tp)
64
65      def accuracy(self):
66          return (self.tp + self.tn) / (self.total())
67
68      def precision(self):
69          return self.tp / (self.tp + self.fp)
70
71      def gmean1(self):
72          return math.sqrt(self.tpr() * self.precision())
73
74      def gmean2(self):
75          return math.sqrt(self.tpr() * self.tnr())
76
77      def __repr__(self):
78          return str(self.__dict__)
79
80      def __str__(self):
81          out = "Results {\n  tp: %d\n  tn: %d\n  fp: %d\n  fn: %d\n}\n" % (self.tp, self.tn,
            ↪  self.fp, self.fn)
82          out += "TPR: %f\n" % self.tpr()
83          out += "TNR: %f\n" % self.tnr()
84          out += "FPR: %f\n" % self.fpr()
85          out += "FNR: %f\n" % self.fnr()
86          out += "Accuracy:  %f\n" % self.accuracy()
87          out += "Precision: %f\n" % self.precision()
88          out += "Gmean1:    %f\n" % self.gmean1()
89          out += "Gmean2:    %f\n" % self.gmean2()
90          out += "Model Training Time:     %f ms\n" % (self.timing['training'] * 1000)
91          out += "Average Prediction Time: %f us\n" % (self.timing['testing'] * 1000 * 1000)
92          return out
```

Listing 9: Interval Timer class

```
#!/usr/bin/env python
import time
import timeit
import unittest


class IntervalTimer(object):
    """ A collection of named timers for performance measurements

       This class supports repeated tick/tock calls on any given timer,
       allowing for measurement of the mean interval time between
       successive tick/tock calls.


    Attributes:
        timers      Dictionary contining the timers indexed by name as key
    """

    def __init__(self, default_timers=None):
        self.timers = {}
        if default_timers is not None:
            for tname in default_timers:
                self.add_timer(tname)

    def add_timer(self, name):
        """ Create a new named timer """
        self.timers[name] = {}
        self.timer_reset(name)

    def timer_tick(self, name):
        """ Start recording a new interval using the named timer """
        if not name in self.timers:
            self.add_timer(name)
        self.timers[name]['start'] = timeit.default_timer()

    def timer_tock(self, name, reset=False):
        """ Stop measuring the current interval of (and optionally reset) the named timer """
        if reset:
            self.timers[name]['total_time'] = timeit.default_timer() - self.timers[name]['start']
            self.timers[name]['intervals'] = 1
        else:
            self.timers[name]['total_time'] += timeit.default_timer() - self.timers[name]['start']
            self.timers[name]['intervals'] += 1

    def timer_reset(self, name):
        """ Reset a named timer to initial zero settings """
        self.timers[name]['start'] = 0.0
        self.timers[name]['intervals'] = 0
        self.timers[name]['total_time'] = 0.0

    def average_interval(self, name):
        """ Mean interval between all tick/tock calls to a named timer """
        return self.timers[name]['total_time'] / max(1, self.timers[name]['intervals'])
```

```
54
55      def total_time(self, name):
56          """Total time taken by a named timer. Sum of all interval times"""
57          return self.timers[name]['total_time']
58
59      def __len__(self):
60          return len(self.timers.keys())
61
62
63
64  class TestTickTock(unittest.TestCase):
65      def setUp(self):
66          self.timer = IntervalTimer()
67
68      def testConstructor(self):
69          timer = IntervalTimer()
70          self.assertEqual(len(timer), 0)
71          timer = IntervalTimer(['clock1', 'clock2', 'clock3'])
72          self.assertEqual(len(timer), 3)
73
74      def testAutoAddTimer(self):
75          timer = IntervalTimer()
76          self.assertEqual(len(timer), 0)
77          timer.timer_tick('new_clock')
78          self.assertEqual(len(timer), 1)
79
80      def testBadTimerName(self):
81          timer = IntervalTimer()
82          self.assertRaises(KeyError, timer.timer_tock, 'bad_clock')
83          self.assertRaises(KeyError, timer.timer_reset, 'bad_clock')
84          self.assertRaises(KeyError, timer.average_interval, 'bad_clock')
85          self.assertRaises(KeyError, timer.total_time, 'bad_clock')
86
87      def testTickTock(self):
88          timer = IntervalTimer(['test_clock'])
89          timer.timer_tick('test_clock')
90          time.sleep(2.00)
91          timer.timer_tock('test_clock')
92          self.assertLess(abs(2.0 - timer.total_time('test_clock')), 0.01)
93
94      def testIntervalAverage(self):
95          timer = IntervalTimer(['test_clock'])
96          for _ in range(20):
97              timer.timer_tick('test_clock')
98              time.sleep(0.1)
99              timer.timer_tock('test_clock')
100         self.assertLess(abs(0.1 - timer.average_interval('test_clock')), 0.01)
101
102
103
104 if __name__ == '__main__':
105     unittest.main()
106
107
108 # vim:fileencoding=utf-8
```

Listing 10: C-Python interface files

```
# python_interface.py
import cffi

api = None

FFI = cffi.FFI()
FFI.cdef(open('python_interface.h', 'r').read())

# Hold references to objects to prevent garbage collection.
noGCDict = {}

@FFI.callback("int (float, float)")
def python_predict(f1, f2):
    return 1


def fill_api(ptr):
    global api
    api = FFI.cast("struct API*", ptr)
    api.python_predict = python_predict
```

```
/* python_interface.h */
struct API {
    int (*python_predict)(float f1, float f2);
};

int python_setup();

void python_destroy();

int python_predict(float, float);
```

```
/* python_interface.c */
#include <Python.h>
#include <assert.h>
#include <stdio.h>
#include "python_interface.h"

struct API api;

int python_setup(){
    int rc;

    PyObject *pName, *pModule, *py_results;
    PyObject *fill_api;
#define PYVERIFY(exp) if ((exp) == 0) { fprintf(stderr, "%s[%d]: ", __FILE__, __LINE__);
↪  PyErr_Print(); exit(1); }

    Py_SetProgramName("PredictionLLFSM");
    Py_Initialize();
    PyRun_SimpleString("import sys; sys.path.insert(0, '.')");
```

```
19      PYVERIFY(pName = PyString_FromString("python_interface"))
20      PYVERIFY(pModule = PyImport_Import(pName))
21      Py_DECREF(pName);
22      PYVERIFY(fill_api = PyObject_GetAttrString(pModule, "fill_api") )
23      PYVERIFY( py_results = PyObject_CallFunction(fill_api, "k", &api) )
24      assert(py_results == Py_None);
25      if (py_results == Py_None){
26          return 0;
27      } else {
28          return 1;
29      }
30  }
31
32  void python_destroy(){
33      Py_Finalize();
34  }
35
36  int python_predict(float f1, float f2)
37  {
38      printf("predicted: %d\n", api.python_predict(f1, f2));
39      return 0;
40  }
```

Listing 11: Arduino imitation sensor code

```
#include <Wire.h>

#define I2C_ADDRESS 0x42
#define DATASET_SIZE 180

const float EM[] = {
  91.3, 90.6, 83.5, 79, 83.5, 89.5, 80.2, 68.4, 95.2, 86.2, 84.9, 80.6, 88, 80.6, 81.9, 89.6,
  ↪ 84.2, 73.5, 40.2, 85.2, 90, 90, 67.7, 84.4, 84.9, 90, 92.2, 94.8, 83.7, 76.6, 64.5, 69.9,
  ↪ 87, 83.8, 67.9, 67.8, 82.3, 93.5, 89.8, 76.3, 79.7, 76.8, 93.9, 76.8, 87.3, 74, 91.8, 78,
  ↪ 89.6, 90.5, 88.8, 86.1, 96.8, 82.6, 96.8, 82.2, 85.7, 91.4, 86.1, 98.7, 86.8, 83.4, 82.7,
  ↪ 85.2, 77.7, 69.3, 73, 91.5, 80.1, 47.9, 62.8, 86.3, 82.8, 93.5, 89.8, 89.4, 86, 83.5, 85.5,
  ↪ 85.6, 86.1, 89.3, 88.8, 70.9, 89.4, 72.4, 85.8, 87.7, 89.8, 93.6, 84.6, 85, 76.7, 86.5,
  ↪ 89.9, 89.8, 83.6, 76.6, 93.6, 90.9, 90.2, 86.3, 79.7, 55.9, 74, 92.3, 78.6, 70.9, 85.4,
  ↪ 38.1, 85, 85.6, 65.1, 84.3, 86.8, 83.6, 68, 88.4, 80.9, 75.4, 85.6, 63.1, 67.8, 94.6, 37,
  ↪ 77.2, 89, 86.4, 79, 88.5, 85.6, 88.3, 74.9, 89, 89.5, 95.6, 83.9, 81.2, 95.6, 89.4, 77.2,
  ↪ 86.4, 91.4, 90, 88.9, 81.1, 81.2, 80.4, 75.1, 83.1, 86.6, 86.5, 75.9, 82.6, 86.9, 89.6,
  ↪ 88.6, 89.8, 77.6, 81.3, 81.4, 81.5, 80.6, 73.9, 81.3, 90.5, 90.6, 88, 77.7, 87.7, 89.6, 82,
  ↪ 87.5, 76.3, 73, 80.1, 87.7, 94.3, 84.4, 61.5
};

const float NDVI[] = {
  0.348841172, 0.379982441, 0.14986376, 0.315929793, 0.198514909, 0.348761761, 0.332568972,
  ↪ 0.425734098, 0.332956153, 0.239849693, 0.206847361, 0.325738977, 0.370821256, 0.267190227,
  ↪ 0.418527316, 0.449445865, 0.30980261, 0.482496195, 0.256261682, 0.169379571, 0.442922374,
  ↪ 0.283910465, 0.264765358, 0.200633312, 0.492106522, 0.277810037, 0.250658601, 0.186214173,
  ↪ 0.401496764, 0.485765125, 0.317033708, 0.213696576, 0.430020284, 0.234536733, 0.355577623,
  ↪ 0.462733957, 0.364386144, 0.414926, 0.48255814, 0.221381158, 0.259221535, 0.266633712,
  ↪ 0.320709825, 0.363435148, 0.243536546, 0.27946372, 0.411242126, 0.297200589, 0.408376546,
  ↪ 0.256332518, 0.231502443, 0.400198413, 0.394094994, 0.342700548, 0.257295442, 0.331784387,
  ↪ 0.439843137, 0.381010864, 0.256131403, 0.426165601, 0.339614995, 0.32865872, 0.243830491,
  ↪ 0.378115502, 0.273918446, 0.343309859, 0.273073593, 0.403429676, 0.435183785, 0.34729064,
  ↪ 0.375381303, 0.170445956, 0.290877797, 0.268600842, 0.423726023, 0.396775349, 0.2451412,
  ↪ 0.396761943, 0.24335679, 0.239764991, 0.214401653, 0.271585557, 0.380089006, 0.387606519,
  ↪ 0.311972716, 0.195385188, 0.328304664, 0.190647037, 0.203477495, 0.408380521, 0.239566651,
  ↪ 0.320116938, 0.261230271, 0.367591159, 0.432944019, 0.389872691, 0.396120814, 0.233294708,
  ↪ 0.22377021, 0.246907752, 0.173266345, 0.402952904, 0.432299546, 0.212168855, 0.282768659,
  ↪ 0.238687669, 0.377703534, 0.378359615, 0.521881659, 0.133850337, 0.241281714, 0.378583129,
  ↪ 0.272509004, 0.440285205, 0.238383838, 0.304837801, 0.55639211, 0.526236349, 0.400954198,
  ↪ 0.565887446, 0.602822941, 0.310390361, 0.436360729, 0.362593516, 0.131584258, 0.295362405,
  ↪ 0.369048728, 0.319145033, 0.292020191, 0.305255748, 0.397186872, 0.27005597, 0.229500687,
  ↪ 0.345377439, 0.363451866, 0.215808171, 0.216472848, 0.265038961, 0.281539558, 0.35052161,
  ↪ 0.356010901, 0.290839016, 0.389499389, 0.305620719, 0.194415621, 0.348660105, 0.431729907,
  ↪ 0.202021336, 0.368096941, 0.275391612, 0.239079693, 0.394459103, 0.262923622, 0.2854494,
  ↪ 0.364300272, 0.284158764, 0.402255639, 0.188804976, 0.252648241, 0.236873747, 0.260907889,
  ↪ 0.276612053, 0.166584141, 0.206108501, 0.291743875, 0.360708535, 0.2710013, 0.310858995,
  ↪ 0.307394466, 0.358828316, 0.193246241, 0.326904153, 0.195794354, 0.276736889, 0.229441524,
  ↪ 0.345811412, 0.355354391, 0.311704385, 0.280827869, 0.258089368
};

// Keep track of which value we are up to, wrap around once we hit DATASET_SIZE
int value = 0;

// Variable to store the current value between Tx/Rx interrupts
```

```
18  float sensorValue = 0;
19
20  void loop() { /* nothing to do in main loop */ }
21
22  void setup() {
23    Wire.begin(I2C_ADDRESS);
24    Wire.onReceive(rxEvent);
25    Wire.onRequest(reqEvent);
26  }
27
28  /* UART Rx interrupt handler */
29  void rxEvent(int count){
30    while (Wire.available()){
31      int cmd = (int)Wire.read();
32      switch (cmd){
33        case 0:
34          value++;
35          if (value >= DATASET_SIZE){
36            value %= DATASET_SIZE;
37          }
38          break;
39        case 1:
40          sensorValue = EM[value];
41          break;
42        case 2:
43          sensorValue = NDVI[value];
44          break;
45      }
46    }
47  }
48
49  /* UART Tx interrupt handler */
50  void reqEvent(){
51    Wire.write((uint8_t*) &sensorValue, sizeof(sensorValue));
52  }
```

Listing 12: Main entry point script to run Fuzzy Boxes model tests

```
#!/usr/bin/env python2
from __future__ import print_function
import sys
import random
import argparse
import time
import errno
import cProfile
import pstats
from datetime import datetime

from fuzzyboxes import FuzzyBoxModel, FuzzyBoxFeature, FuzzyBoxInstance
from fuzzyboxes import graphics

from util.functions import read_csv
from util import Results, IntervalTimer

try:
    from util.i2c import I2C
    I2C_AVAILABLE = True
except ImportError:
    print("WARNING: No support for I2C on this system", file=sys.stderr)
    I2C_AVAILABLE = False


status_codes = {
    0: "OK",
    1: "File not found",
}


def eprint(*args, **kwargs):
    print(*args, file=sys.stderr, **kwargs)


def _exit(code):
    print("exited with code: %d, %s" % (code, status_codes[code]))
    exit(code)


def read_instances(filename, value_columns):
    for row in read_csv(filename, value_columns=value_columns):
        row_values = list(map(float, row['values']))
        row_class = bool(int(row['labels'][0]))
        yield FuzzyBoxInstance(values=row_values, classification=row_class)


DEFAULTS = {
    'features': 2,
    'instances': 25,
    'smax': 20,
    'method': 'even',
    'prior': 0.5,
```

```
54      'i2c_address': 0x42,
55  }
56
57
58  def parse_args():
59      parser = argparse.ArgumentParser(description='Run data through the the FuzzyBox algorithm')
60
61      actions = parser.add_mutually_exclusive_group(required=True)
62      actions.add_argument(
63          '--train', dest='train_file', metavar='FILE', help='Train the model using CSV values from
            ↪  FILE')
64      parser.add_argument(
65          '--test', dest='test_file', metavar='FILE', help='Test the accuracy and timing of the
            ↪  model using CSV values from FILE')
66      parser.add_argument(
67          '--read-i2c', dest='read_i2c', action='store_true', help='Read incoming values from IIC
            ↪  interface in a loop until Ctrl-C is pressed')
68      actions.add_argument(
69          '--simulate', action='store_true', help='Simulate training data using random values')
70
71      options = parser.add_argument_group('options')
72      options.add_argument(
73          '--smax', dest='smax', type=int, default=DEFAULTS['smax'], help='Maximum number of split
            ↪  points')
74      options.add_argument(
75          '--method', dest='method', default=DEFAULTS['method'], choices=['even', 'percentile'],
            ↪  help='Boxing method to use while training')
76      options.add_argument(
77          '--prior', dest='prior', type=float, help='Prior ratio to use for incoming data samples.
            ↪  defaults to the class distribution ratio of the training set')
78      options.add_argument(
79          '--scatter', dest='scatter', action='store_true', help='Produce scatter plot for boxes
            ↪  1-8, using features 1,2')
80      options.add_argument(
81          '--heatmap', dest='heatmap', action='store_true', help='Produce heatmap of the model @
            ↪  S_max split points')
82      options.add_argument(
83          '--details', dest='details', action='store_true', help='Print details of the model after
            ↪  training')
84      options.add_argument(
85          '--csv-columns', dest='csv_columns', nargs='+', type=int, default=[], metavar='COL',
            ↪  help='Filter CSV columns by column number (index 0 is actually second column. first
            ↪  column is assumed to be label)')
86      options.add_argument(
87          '--i2c-address', dest='i2c_address', type=int, default=DEFAULTS['i2c_address'],
            ↪  metavar='X', help='I2C address of the sensor playback device')
88      options.add_argument(
89          '--profile', dest='profile', action='store_true', help='Produce a profile file for the
            ↪  current execution settings')
90
91      sim_options = parser.add_argument_group('simulation options')
92      sim_options.add_argument(
93          '--features', dest='features', type=int, default=DEFAULTS['features'], metavar='F',
            ↪  help='Number of training features to simulate')
94      sim_options.add_argument(
95          '--instances', dest='instances', type=int, default=DEFAULTS['instances'], metavar='I',
            ↪  help='Number of training features to simulate')
```

```
96
97      return parser.parse_args()
98
99
100 def main(args):
101
102     training_features = []
103     training_classes = []
104
105     stopwatch = IntervalTimer()
106     results = Results()
107
108     fbm = FuzzyBoxModel(args.smax, args.method, prior_ratio=args.prior)
109
110     # train the model with simulated (random) data
111     if args.simulate:
112         print("> Training model using simulated data")
113         for i in range(args.features):
114             values = [random.randint(0, 300) for _ in range(args.instances)]
115             training_features.append(FuzzyBoxFeature('feature_%d' % i, values))
116             training_classes = [random.choice([True, False]) for _ in range(args.instances)]
117
118     # train the model from the given CSV file
119     elif args.train_file is not None:
120         try:
121             feature_instances = [
122                 FuzzyBoxInstance(values=list(map(float, row['values'])),
123                                  classification=bool(int(row['labels'][0])))
124                 for row in read_csv(args.train_file)
125             ]
126             feature_instances = [i for i in read_instances(args.train_file, args.csv_columns)]
127             feature_values = []
128
129             # create empty lists for each feature's values
130             for fi in range(len(feature_instances[0])):
131                 feature_values.append([])
132
133             # populate values/classifications lists
134             for instance in feature_instances:
135                 for fi in range(len(instance)):
136                     feature_values[fi].append(instance.values[fi])
137                 training_classes.append(instance.classification)
138
139             # convert values to FuzzyBoxFeature
140             for fi in range(len(feature_values)):
141                 training_features.append(FuzzyBoxFeature('feature_%d' % fi,
                    ↪  values=feature_values[fi]))
142             print("> Training model using data from %s" % args.train_file)
143         except IOError as e:
144             if e.errno == errno.ENOENT:
145                 print("ERROR: Could not find training file: %s" % args.train_file)
146                 _exit(1)
147             else:
148                 raise
149
150     # do the training and record the time taken
151     stopwatch.timer_tick('train')
```

```
152     fbm.train(training_features, training_classes, stopwatch)
153     stopwatch.timer_tock('train')
154 
155     if args.scatter:
156         print("> Generating scatter plot...")
157         filename = 'fuzzybox-scatter-%s-%03d.png' % (args.method, args.smax)
158         graphics.scatter(fbm).savefig(filename)
159         print("Scatter plot saved to %s" % filename)
160 
161     if args.heatmap:
162         print("> Generating heatmap(s)...")
163         for s in range(fbm.Smax):
164             filename = 'fuzzybox-heatmap-%s-S%03d.png' % (args.method, s)
165             graphics.heatmap(fbm, S=s).savefig(filename)
166             print('Heatmap saved to %s' % filename)
167 
168     if args.details:
169         details = fbm.details()
170         print("Max Boxes: %d" % details['max_boxes'])
171         print("Total Boxes: %d" % details['total_boxes'])
172         print("Boxing Method: %s" % details['boxing_method'])
173         print("Trained: %s" % details['trained'])
174         print("Features:")
175         for feature in details['features']:
176             print("\t%s" % feature['name'])
177             print("\t\tRange: %d - %d" % feature['range'])
178             print("\t\tSplit Points:" % feature['split_points'])
179             for S in range(details['max_boxes']):
180                 splits = ', '.join(["%2.2f" % sp for sp in feature['split_points'][S]])
181                 print("\t\t\tS = %d\t%s" % (S, splits))
182         print("Box Weights:")
183         for S in range(1, len(details['box_weights'])):
184             print("\tS = %d" % S)
185             for f, _ in enumerate(details['box_weights'][S]):
186                 print("\t\t%s" % details['box_weights'][S][f])
187         print("Training Instances:")
188         for instance in details['training']:
189             positions = details['training'][instance]
190             print("\t%s" % (instance,))
191             print("\t\t%s" % ", ".join(positions))
192         print("=" * 79)
193 
194     if args.read_i2c:
195         if I2C_AVAILABLE:
196             address = args.i2c_address if args.i2c_address else DEFAULTS['i2c_address']
197             i2c = I2C(address)
198             print("> Reading input from I2C sensor on address 0x%x" % (address,))
199             read_count = 0
200             try:
201                 for f1, f2 in i2c.read_values():
202                     read_count += 1
203                     print("EM=%r, NDVI=%r" % (f1, f2))
204                     stopwatch.timer_tick('i2c_predict')
205                     predicted = fbm.predict([f2, f1], stopwatch)
206                     stopwatch.timer_tock('i2c_predict', reset=True)
207 
208                     print("Predicted classification: %r (took %.2f ms, %.2f ms avg.)\n" % (
```

```
                        predicted,
                        stopwatch.total_time('i2c_predict'),
                        stopwatch.average_interval('predict-internal')))

            except KeyboardInterrupt:
                print("")
                print("I2C read finished. %d values processed" % read_count)

    # test the model
    if args.test_file is not None:
        print("> Testing model using data from %s" % args.test_file)

        for instance in read_instances(args.test_file, args.csv_columns):

            # time the performance of the predict function
            stopwatch.timer_tick('predict')
            predicted = fbm.predict(instance.values, stopwatch)
            stopwatch.timer_tock('predict')

            # add the classification result for accuracy measurements
            results.add(predicted, instance.classification)

        # show the results
        print("=" * 79)
        print(results)
        print("Model Training Time (ms): %f" % (stopwatch.total_time('train') * 1000,))
        print("Internal Model Training Time (ms): %f" % (stopwatch.total_time('train-internal') *
        ↪  1000,))
        print("Average Predict Time (us): %f" % (stopwatch.average_interval('predict') * 1000 *
        ↪  1000,))
        print("Average Internal Predict Time (us): %f" %
        ↪  (stopwatch.average_interval('predict-internal') * 1000 * 1000,))


if __name__ == '__main__':

    args = parse_args()

    if args.profile:
        statsfile = "profile-%s.stats" % datetime.now().strftime("%Y%m%d.%H%M")
        cProfile.run("main(args)", statsfile)
        stats = pstats.Stats(statsfile)
        stats.strip_dirs().sort_stats('time').print_stats(10)
    else:
        main(args)
```

Listing 13: Fuzzy Boxes varying-depth timing tests

```
#!/usr/bin/env python
"""
    splits_vs_time

    This script compares the accuracy and prediction time between the 'even' and
    'percentile' Fuzz Box models.

    Prediction time is tested on a range of depths (1-30) in order to quantify
    the performance penalty for increasing the depth.
"""
from __future__ import print_function
import os
import sys

from fuzzyboxes.model import FuzzyBoxModel, FuzzyBoxFeature
from fuzzyboxes.results import Results
from fuzzyboxes.functions import read_csv
from util import IntervalTimer


ERROR_INSTANCE_LENGTH = 'Mismatch in length of training value and classes'
ERROR_CSV_LENGTH = 'Not enough values to write CSV data! (expected %d, got %d)'
ERROR_FILE_NOT_FOUND = '%s file does not exist (%s)'


def csv_write_line(filepath, csv_values=None, append=True):
    """ Write a single line (row) to the CSV output file """
    if not csv_values:
        csv_values = []
    fflag = 'a' if append else 'w'
    with open(filepath, fflag) as csvfile:
        csvfile.write(','.join(map(str, csv_values))+'\n')


def print_prediction_status(splits, run):
    """ Print a line to display the current split/run number """
    print('\b' * 80, end='')
    print('Testing predict time (depth=%d), run #%d' % (splits, run), end='')
    sys.stdout.flush()


if __name__ == '__main__':

    # PREPARE FILES
    ###############################################################################

    SCRIPT_PATH = os.path.dirname(os.path.realpath(sys.argv[0]))
    TRAIN_FILE = os.path.join(SCRIPT_PATH, 'test', 'train_data.csv')
    TEST_FILE = os.path.join(SCRIPT_PATH, 'test', 'test_data.csv')

    assert os.path.exists(TRAIN_FILE), ERROR_FILE_NOT_FOUND % ('Training', TRAIN_FILE)
    assert os.path.exists(TEST_FILE), ERROR_FILE_NOT_FOUND % ('Testing', TEST_FILE)

```

```
54      # CREATE DATA
55      ###############################################################################
56
57      # All training instances as FBInstance objects
58      TRAINING_INSTANCES = list(read_csv(TRAIN_FILE))
59      TRAINING_INSTANCES = list(read_csv(TRAIN_FILE))
60
61      # Just classes in a seperate list
62      TRAINING_CLASSES = [i.classification for i in TRAINING_INSTANCES]
63
64      # Create FBFeatures, which contain a list of all instance values
65      TRAINING_FEATURES = [
66          FuzzyBoxFeature('feature_%d' % i, values=values)
67          for i, values in enumerate(zip(*[ti.values for ti in TRAINING_INSTANCES]))
68      ]
69
70      # All testing instances as FBInstance objects
71      TESTING_INSTANCES = list(read_csv(TEST_FILE))
72
73      assert len(TRAINING_INSTANCES) == len(TRAINING_CLASSES), ERROR_INSTANCE_LENGTH
74
75      # TRAIN MODELS
76      ###############################################################################
77      print('========== Training ==========')
78      print('  Instances: %d' % len(TRAINING_INSTANCES))
79      print('  Features:  %d' % len(TRAINING_FEATURES))
80
81      S_MAX = 30
82      SPLIT_METHODS = ['even', 'percentile']
83
84      # Create a depth=30 model for both even and percentile splits (prior=0.5)
85      MODELS = {
86          m: FuzzyBoxModel(30, m, prior_ratio=0.5) for m in SPLIT_METHODS
87      }
88
89      # Do the training, adding a timer into the model
90      for method, fbm in MODELS.items():
91          fbm.timer = IntervalTimer(['predict-internal'])
92          fbm.train(TRAINING_FEATURES, TRAINING_CLASSES, fbm.timer)
93          print('> Trained model using %s splits' % method)
94
95      # TEST MODELS
96      ###############################################################################
97
98      CSV_HEADER_FIELDS = [
99          'Model', 'Depth',
100         'TP', 'FP', 'TN', 'FN',
101         'TPR', 'FPR', 'TNR', 'FNR',
102         'Accuracy', 'Precision',
103         'Gmean1', 'Gmean2',
104         'Training Time (ms)', 'Avg. Predict Time (us)',
105     ]
106
107     CSV_FILENAME = 'splits-vs-time.results.csv'
108
109     csv_write_line(CSV_FILENAME, CSV_HEADER_FIELDS, append=False)
110
```

```
111     # Test each model
112     for method, fbm in MODELS.items():
113
114         # Run predictions using smax = [1-30]
115         for depth in range(1, 31):
116
117             # reset results and prediction timer
118             results = Results()
119             fbm.timer.timer_reset('predict-internal')
120
121             # Get accuracy results and an initial prediction time
122             for ti in TESTING_INSTANCES:
123                 predicted = fbm.predict(ti.values, fbm.timer, max_depth=depth)
124                 results.add(predicted, ti.classification)
125
126             predict_time_min = fbm.timer.average_interval('predict-internal')
127
128             # prediction timing tests, run 100 times and take the minimum
129             for r in range(10):
130                 print_prediction_status(depth, r)
131
132                 fbm.timer.timer_reset('predict-internal')
133                 for ti in TESTING_INSTANCES:
134                     _ = fbm.predict(ti.values, fbm.timer, max_depth=depth)
135
136                 predict_time_min = min(
137                     predict_time_min,
138                     fbm.timer.average_interval('predict-internal'))
139
140             # Prepare the CSV row
141             line = [
142                 method, depth,
143                 results.tp, results.fp, results.tn, results.fn,
144                 results.tpr(), results.fpr(), results.tnr(), results.fnr(),
145                 results.accuracy(), results.precision(),
146                 results.gmean1(), results.gmean2(),
147                 fbm.timer.total_time('train-internal') * 1000,
148                 fbm.timer.average_interval('predict-internal') * 1000 * 1000
149             ]
150
151             assert len(line) == len(CSV_HEADER_FIELDS), ERROR_CSV_LENGTH % (
152                 len(CSV_HEADER_FIELDS), len(line))
153
154             csv_write_line(CSV_FILENAME, line)
155
156     print('%sResults saved to %s' % ('\b'*80, CSV_FILENAME))
157
158 # vim:fileencoding=utf-8
```

Listing 14: Fuzzy Boxes varying-depth accuracy tests

```
#!/usr/bin/env python3
# vim:fileencoding=utf-8
import os
import sys
import matplotlib.pyplot as pyplot

from fuzzyboxes.model import FuzzyBoxModel, FuzzyBoxFeature
from fuzzyboxes.functions import read_csv
from fuzzyboxes.results import Results
from util import IntervalTimer


if __name__ == '__main__':
    script_path = os.path.dirname(os.path.realpath(sys.argv[0]))
    train_file = os.path.join(script_path, "test", "oversample_train.csv")
    test_file = os.path.join(script_path, "test", "oversample_test.csv")

    s_max = 30

    timer = IntervalTimer()

    if not os.path.exists(train_file):
        raise Exception("training file does not exist: %s" % train_file)

    if not os.path.exists(test_file):
        raise Exception("testing file does not exist: %s" % test_file)

    feature_instances = [i for i in read_csv(train_file)]
    feature_values = []
    training_features = []
    training_classes = []

    # create training data
    ##############################################################################

    # create empty lists for each feature's values
    for fi in range(len(feature_instances[0])):
        feature_values.append([])

    # populate values/class lists
    for instance in feature_instances:
        for fi in range(len(instance)):
            feature_values[fi].append(instance.values[fi])
        training_classes.append(instance.classification)

    # convert to FuzzyBoxFeature
    for fi in range(len(feature_values)):
        training_features.append(FuzzyBoxFeature("feature_%d" % fi, values=feature_values[fi]))

    # train models
    ##############################################################################
    methods = ["even", "percentile"]
    accuracy = {}
```

```
    models = {method: [FuzzyBoxModel(s, method, prior_ratio=0.5) for s in range(1, s_max + 1)] for
    ↪  method in methods}
    for method in models:
        accuracy[method] = []
        for fbm in models[method]:
            fbm.train(training_features, training_classes, timer)
            r = Results()
            for instance in read_csv(test_file):
                r.add(fbm.predict(instance.values, timer), instance.classification)
            accuracy[method].append(r.accuracy())

    for method in methods:
        print("%s,%s" % (method, ",".join(map(str,accuracy[method]))))

    # plot results
    ############################################################################
    ax_id = 1
    x_values = range(1, s_max + 1)
    fig = pyplot.figure(1)
    for method in methods:
        ax = fig.add_subplot(210 + ax_id)
        ax_id += 1
        ax.axis([1, s_max, 0, 1])
        ax.xaxis.set_ticks([x for x in x_values if x % 2 == 0])
        ax.plot(x_values, accuracy[method])
        ax.set_title(method)
        ax.grid(True, which='both')
    pyplot.show()
```

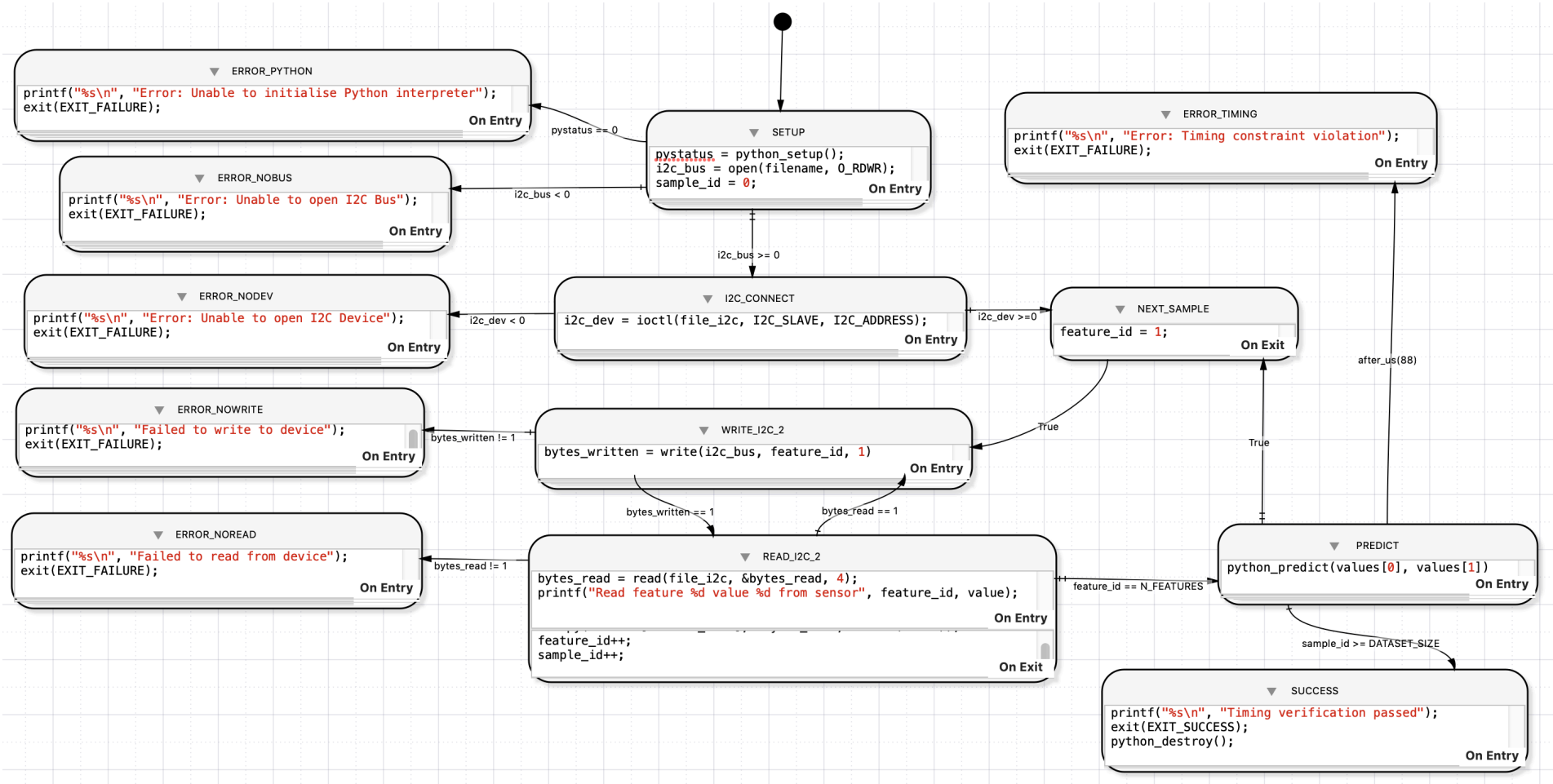


Figure A.1: LLFSM to verify timing constraints of algorithms on a Raspberry Pi